Deep Dive into Asynchronous Patterns in JavaScript

@joeandaverde

TITANIUM SPONSORS

epiq

PAIGE TECHNOLOGIES
INTELLIGENT PAIRING. PERPETUAL SUCCESS.

ADAPTIVE™ SOLUTIONS GROUP

VALOREM CONSULTING

Platinum Sponsors

VinSolutions
Make every connection count.

jack henry & ASSOCIATES INC.®

KU EDWARDS CAMPUS
The University of Kansas

NAIC
National Association of Insurance Commissioners

NIPR
NATIONAL INSURANCE PRODUCER REGISTRY

JET BRAINS

Stormpath

TEKsystems
Our people make IT possible.

stackify

Octopus Deploy

Gold Sponsors

OAKWOOD

KEYHOLE SOFTWARE

Cerner

PocketCake™

Bradford & Galt
CONSULTING SERVICES

GARMIN

dsi

PLURALSIGHT

OBJECT PARTNERS

DST SYSTEMS

Advantage Tech
IT Staffing & Recruiting Services

An affiliate of C&A Industries, Inc.
AUREUS GROUP
Finance | Systems | Executive

Commerce Bank

LRS Consulting Services

Balance Innovations

SPARKPOST

twilio

UNITEDLEX

StrongLoop
An IBM Company

PeopleAdmin

smg service management group®

NERDERY

wirestream™
enabling ideas

RAYGUN

Useagility

TRIPLE-I
AS INDIVIDUAL AS YOU

VML

SAIC
Redefining Ingenuity™

H&R BLOCK

ca technologies

Children's Mercy KANSAS CITY

CENTRIQ TRAINING

mylo™
POWERED BY LOCKTON

What do you mean asynchronous JavaScript?

The execution environment is synchronous!

Indeed, JavaScript does not have a means of performing asynchronous **computation**. (WebWorkers changes this a bit).

There are no constructs for synchronization such as *test and set*. Which is essential for performing parallel computation.

More Info: https://en.wikipedia.org/wiki/Test-and-set

Async in JavaScript really means…

Parallel and **non-blocking IO** (Network access, filesystem access, etc) and **concurrent computation** (running JavaScript).

IO is orders of magnitude slower than computation. By waiting for IO in a non-blocking manor we free up our execution environment to perform logic and manipulate memory.

Relative cost of IO

Computation* is **fast**. IO is slow.

Action	Cost (CPU cycles)
L1 Cache*	3
L2 Cache*	14
RAM*	250
Disk	41,000,000
Network	240,000,000

Borrowed from: http://blog.mixu.net/2011/02/01/understanding-the-node-js-event-loop/

Managing Asynchronous Actions

There are many patterns to choose from

- Callbacks
- Promises
- Generators + Promises
- Async/Await

Callback Functions

simple and familiar

Callbacks Overview

functions to be invoked when IO completes

By convention these functions have the signature:

```
function (err, data) {  }
```

Where `err` is `null` or `undefined` in the event of success.

e.g.

```
fs.readFile('/path/to/file', (err, data) => {
  if (err) {
     return console.error('Error reading file')
  }

  console.log(data)
})
```

Callback Pitfalls

It's easy to forget a `return` after handling an error.

```
fs.readFile('/path/to/file', (err, data) => {
  if (err) { // file does not exist
    /*return*/ console.error('File not found')
  }

  console.log('Found file lets do something destructive!')
})
```

Causing both error and success path's to be executed with possibly undesirable consequences.

Callback Pitfalls (2)

It’s easy to invoke the callback multiple times with different parameters.

```
function doThingAsync(cb) {
    fs.readFile('/path/to/file', (err, data) => {
        if (error) {
            cb(err)
        }

        if (data.toString() === 'special') {
            cb(null, 'special data found')
        }

        cb(null, 'not so special')
    })
}

doThingAsync(function (err, data) {
    console.log('I am executed twice!')
})
```

Avoiding Callback Pitfalls

Use return statements anytime you invoke a callback:

```
if (err) {
    return callback(err)
}

return callback(null, data)
```

Ensure exactly once execution:

```
doMyAsyncThing(_.once((err, data) => {
//    this function can only be executed once thanks to lodash's once
})
```

Callback Tips

Increase readability and avoid name collisions with reduced nesting

```
fetchUser(user_id, (err, user) => {
    fetchPosts(user_id, (err2, posts) => {
        fetchSomethingElse(user_id, (err3, other_stuff) => {
            if (err) { /* OOPS! I meant to use err3! */ }
        })
    })
})
```

Try this instead:

```
function onPostsRetrieved(err, posts) { /* ... fetchSomethingElse */ }

function onUserRetrieved(err, user) {
    fetchPosts(user.user_id, onPostsRetrieved)
}

fetchUser(user_id, onUserRetrieved)
```

Callbacks are old news

use a more modern pattern when possible

Promises

A Promise represents an operation that hasn't completed yet, but is expected in the future.

SOURCE: HTTPS://DEVELOPER.MOZILLA.ORG/EN-US/DOCS/WEB/JAVASCRIPT/REFERENCE/GLOBAL_OBJECTS/PROMISE

Run Demo

Promise.resolve

Creates a `Promise` with the specified `fulfillment value`. This value **never changes** meaning subsequent uses of this `Promise` always return the same value.

```
const foo = Promise.resolve('last')

// then's are always evaluated in a subsequent event loop iteration
foo.then(x => {
    console.log('this happens ' + x)
})

foo.then(x => {
    console.log('this also happens ' + x)
})

console.log('This happens first')
```

Run Demo

Promise.reject

Creates a `Promise` with the specified `rejection value`. This value **never changes** and subsequent uses of this `Promise` always result in `catch` being executed.

```
const foo = Promise.reject('rejection!')

foo.then(x => {
    console.log('this never gets executed because the promise was rejected')
})

foo.catch(x => {
    console.log(x)
})

// Same result with multiple handlers
foo.catch(x => {
    console.log('Another ' + x)
})
```

Run Demo

new Promise()

Provides resolve and reject callbacks to easily interface with callback based functions

Here we create a `Promise` to wrap our usage of the callback based `fs.readFile` function.

```
new Promise((resolve, reject) => {
    fs.readFile('/path/to/file', (err, data) => {
        if (err) {
            return reject(err)
        }

        return resolve(data)
    })
})
.then(data => {
    console.log(data)
})
.catch(err => {
    console.log(err)
})
```

Run Demo

Promises are composable

```
const promiseA = Promise.resolve(5)
const promiseB = Promise.resolve(4)

promiseA.then(a => {
    return promiseB.then(b => {
        return a + b
    })
})
.then(result => {
    console.log(result)
})
```

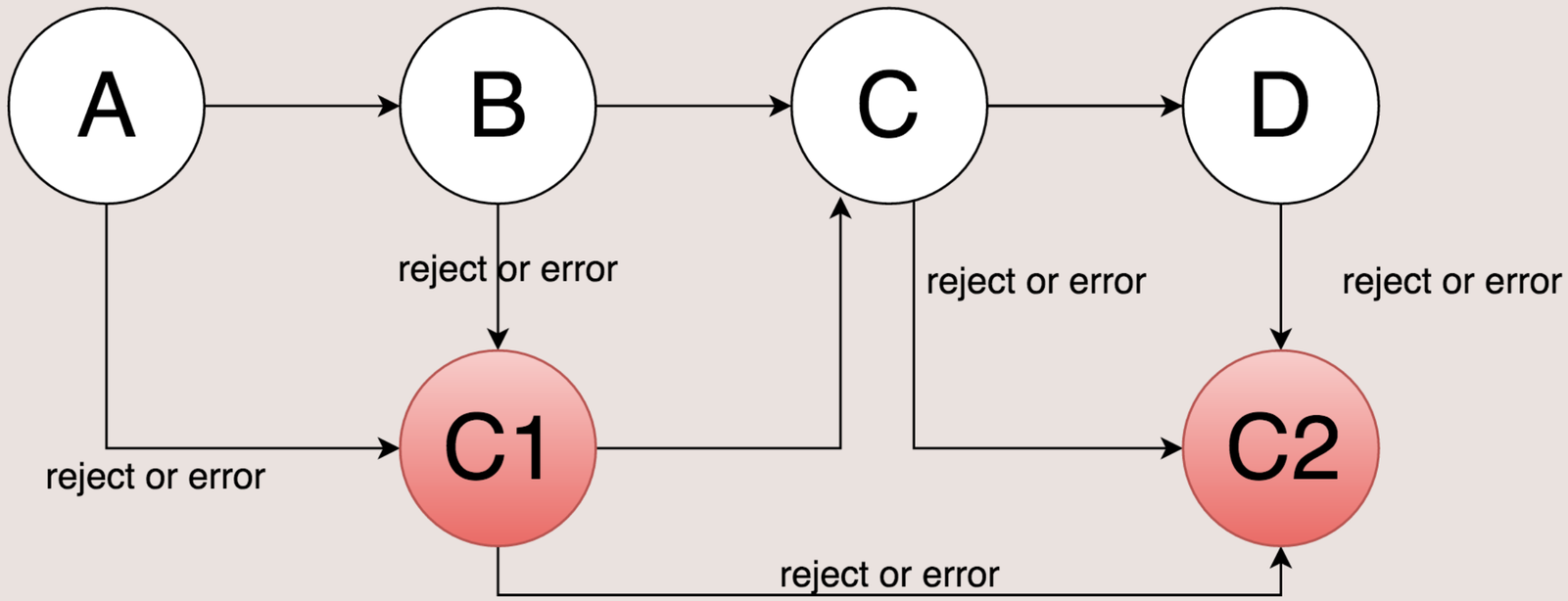
Any value returned from `then` is added to the `Promise chain`.

- `Promise` values are just added to the chain.
- Raw values are wrapped using `Promise.resolve`.
- `throw` or `unhandled error` causes the promise to be rejected.

Run Demo

Promise chain

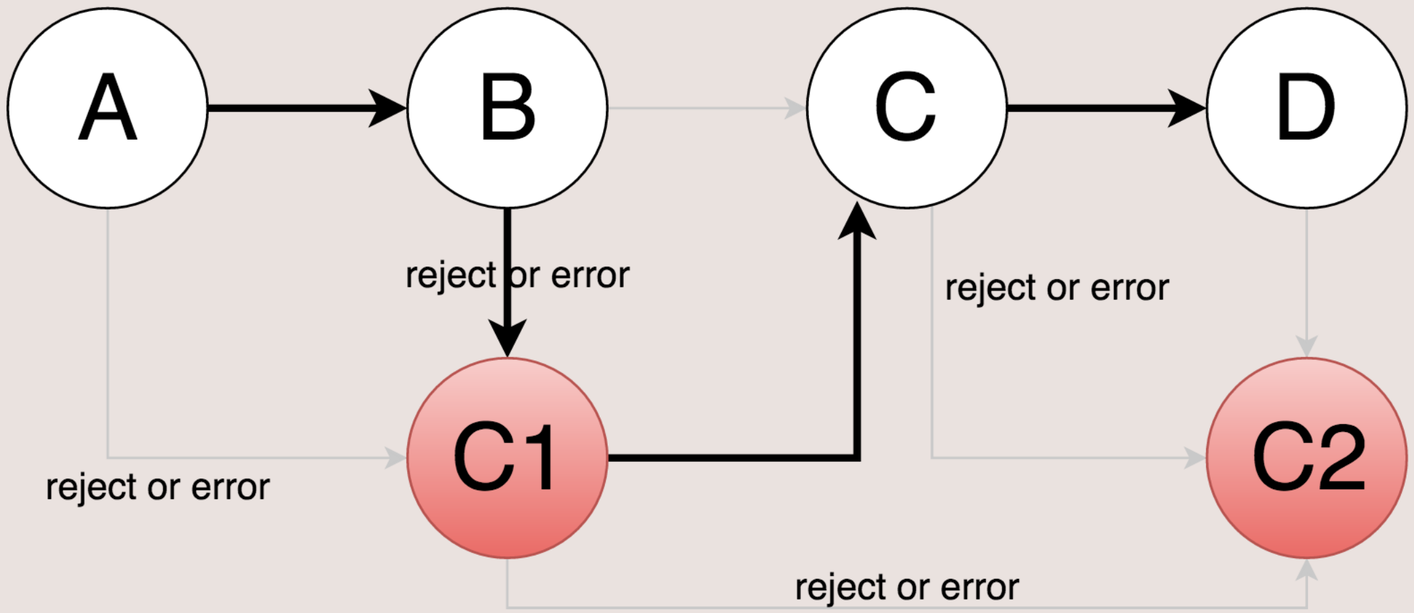
```
promiseA
.then(() => promiseB)
.catch(err => console.log(err)) // C1
.then(() => promiseC)
.then(() => promiseD)
.catch(err => console.log(err)) // C2
```



Run Demo

Promise chain - Example Flow

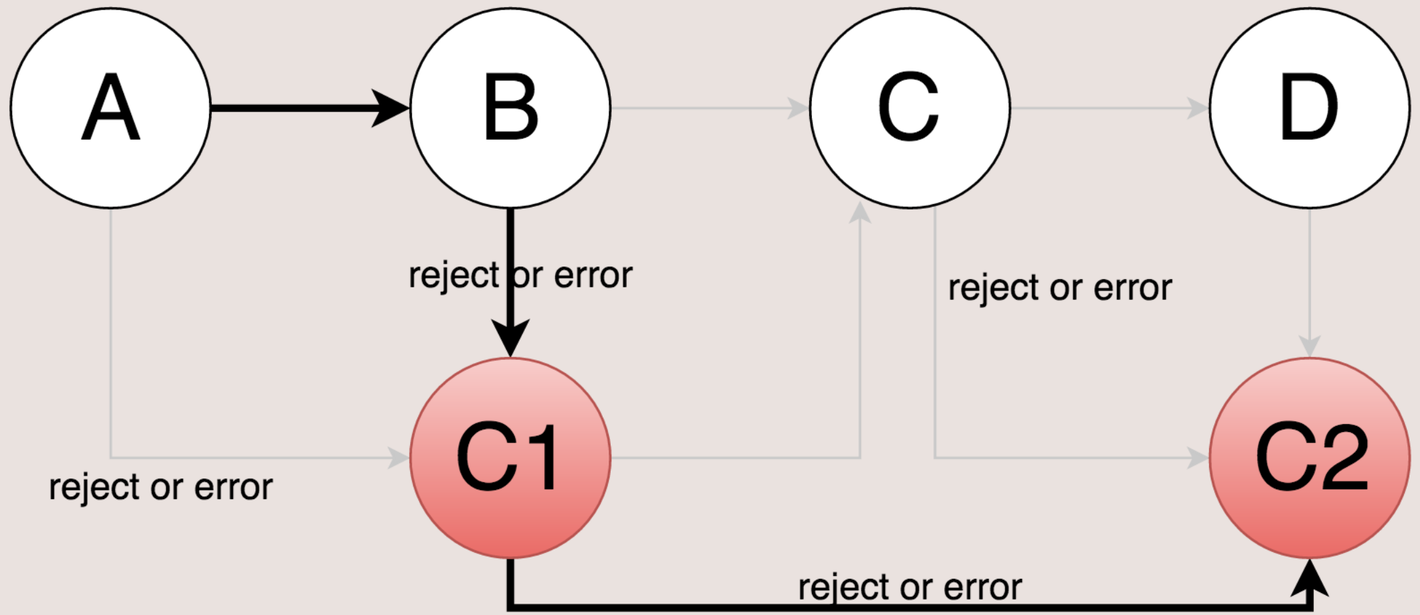
```
promiseA
.then(() => throw new Error('Error')) // Suppose promiseB throws an Error.
.catch(err => console.log(err)) // C1
.then(() => promiseC)
.then(() => promiseD)
.catch(err => console.log(err)) // C2
```



Run Demo

Promise chain - Example Flow 2

```
promiseA
.then(() => throw new Error('Error'))
.catch(err => throw err) // C1
.then(() => promiseC)
.then(() => promiseD)
.catch(err => console.log(err)) // C2
```



Promise.all

waits for all fulfillments (or the first rejection).

```
const userService = require('./services/user')

const users_promise = Promise.all([
    userService.get(123),
    userService.get(555)
])

users_promise.then((users) => {
    const user123 = users[0]
    const user555 = users[1]
    console.log('ALL calls to the userService succeeded!')
})
.catch((error) => {
    console.log('At least one of the calls to the userService failed!')
})
```

Useful when run-order of promises is not important.

Promise.race

receives a collection of promises and resolves or rejects with the value of the first fulfillment

This example illustrates waiting 10 seconds for a service call to complete or timing out.

```
const timeout = new Promise((resolve, reject) => {
    setTimeout(reject, 10000)
})

Promise.race([ timeout, userService.get(123) ])
.then(user => {
    console.log(user)
})
.catch(error => {
    // either error from timeout OR userService.get
    console.log(error)
})
```

Run Demo

Promise Pitfalls

Inadvertently swallowing errors in a catch.

`catch` will either flow you in to the next `then` OR the next `catch` depending on whether the function throws, returns a rejected promise, or succeeds.

```
Promise.reject('original')
.catch(err => {
    return Promise.resolve('default')
})
.then(res => {
    console.log(res) // res -> the value returned from catch
})
```

```
Promise.reject('original')
.then(res => {
    console.log(res) // not executed
})
.catch(err => {
    return Promise.resolve('default')
})
```

Run Demo

Promise Pitfalls (2)

forgetting to return inner promise

```
Promise.resolve('/path/to/file')
.then(filename => {
    /*return*/ new Promise((resolve, reject) => {
        fs.readFile(filename, (err, data) => {
            if(err) { return reject(err) }
            return resolve(data)
        })
    })
})
.then(res => {
    console.log('result: ' + res)
})
```

Flooding the event loop

```
const userService = require('./services/user')

// Imagine if this was a long list of users
const users_to_fetch = [1, 2, 5, 19, ... 9999]

const get_users_promises = _.map(users_to_fetch, (user_id) => {
    return userService.get(user_id)
})

// Note: All promises are already started!
Promise.all(get_users_promises)
```

Run Demo

Avoid flooding the event loop!

use BlueBirds map

```
function timeoutPromise(timeout) {
    return new Promise(resolve => {
        setTimeout(resolve, timeout)
    })
}

const things_to_do = new Array(100)

// Promise.map comes from bluebird (Not available in native implementation) -- finishes in ~2
seconds
Promise.map(things_to_do, () => timeoutPromise(200), { concurrency: 10 })
.then(() => console.log('Batch complete!'))

// As opposed to this which could flood the event loop with large collections
Promise.all(things_to_do.map(() => timeoutPromise(200)))
.then(() => console.log('Flood complete!'))
```

Generators

…functions which can be exited and later re-entered.

SOURCE: HTTPS://DEVELOPER.MOZILLA.ORG/EN-US/DOCS/WEB/JAVASCRIPT/REFERENCE/STATEMENTS/FUNCTION*

Run Demo

Generator example

```
function *oneGenerator() {
    const a = yield 1
    const b = yield 1

    console.log('result 1: ' + a)
    console.log('result 2: ' + b)
}

// Does NOT invoke the function body. Creates a new instance of the generator.
const generator = oneGenerator()

// Invokes the function body for the first time -- the input is ignored
console.log(generator.next('a').value)

console.log(generator.next('b').value)

console.log(generator.next('c').value)
```

Run Demo

Generators don’t block the event loop

this allows us to generate infinite series or defer computation until requested

```
function *myGenerator() {
    let i = 1
    while (true) {
        // suspends execution until the next value is requested
        yield i++
    }
}

const generator = myGenerator()

console.log(generator.next().value)

console.log(generator.next().value)

console.log(generator.next().value)
```

Run Demo

Generators are composable

```
function *oneGenerator() {
    yield 1
    yield* twoGenerator()
    yield 1
}

function *twoGenerator() {
    yield 2
}

const generator = oneGenerator()

console.log(generator.next().value) // run generator to first yield.
console.log(generator.next().value) // now using twoGenerator!
console.log(generator.next().value) // back to oneGenerator
```

Run Demo

How can generators help with async execution?

By taking advantage of composable generators and the power of `Promises`.

```
Promise.coroutine(function* () {
    yield Promise.delay(500)

    console.log('after first delay')

    yield Promise.delay(1000)

    console.log('after second delay')
})()
```

Run Demo

What's this Promise.coroutine business?

It's the precursor to having `async/await`. Generators don't natively understand `Promises`. Bluebird's `Promise.coroutine` delegates to your generator to provide a `Promise` and then invokes `next` on your generator with the fulfillment value.

```
Promise.coroutine(function *() {
    // the fulfilled promise value is unwrapped and provided as the result
    const result = yield Promise.resolve('foobar')

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

Run Demo

(Generators + Promises) vs Promises

```
Promise.coroutine(function* () {
    try {
        const resultA = yield Promise.resolve('foobar')
        const resultB = yield Promise.resolve('baz')
        const resultC = yield Promise.resolve('super')
        console.log('GENERATORS: ' + resultA + resultB + resultC)
    } catch (error) { console.log(error) }
})()
```

```
Promise.resolve('foobar')
.then(resultA => {
    return Promise.resolve('baz')
    .then(resultB => {
        return Promise.resolve('super')
        .then(resultC => console.log('NESTED: ' + resultA + resultB + resultC))
    })
})
.catch(error => console.log(error))
```

Run Demo

Generators with Parallel Promises

That's easy, use the tools we've learned already.

```
Promise.coroutine(function *() {
    const results = yield Promise.all([
        Promise.resolve(1),
        Promise.resolve(2)
    ])

    console.log('array', results[0], results[1])

    // Alternatively, using some new syntax to destructure the result:
    const [a, b] = yield Promise.all([
        Promise.resolve(1),
        Promise.resolve(2)
    ])

    console.log('destructured', a, b)
})()
```

Generators Summary

Generators can greatly increase the readability of your asynchronous code. My suggestion is to start using them right away! They are available as of node version `0.12` (with —harmony flag) and on by default in `4+`.

async/await

the beauty of coroutine without the boilerplate

COMING IN ES7 - NOT RECOMMENDED FOR USE YET

Run Demo

From generators to async/await

```
Promise.coroutine(function* () {
    const [a, b] = yield Promise.all([
        Promise.resolve(1),
        Promise.resolve(2)
    ])

    console.log(a, b)
})()
```

Using async/await:

```
async function () {
    const [a, b] = await Promise.all([
        Promise.resolve(1),
        Promise.resolve(2)
    ])

    console.log(a, b)
}
```

Debugging Asynchronous JavaScript

Run Demo

Stacktraces are helpful if you have them

But sadly, stacktraces are lost on the event loop boundary.

```
function doMyThing() {
    setTimeout(() => {
        console.log(new Error().stack)
    }, 100)
}

doMyThing()
```

Run Demo

Maintaining your stacktrace

Capture your stack trace before setting up an asynchronous callback if you want to know where things happened. **Warning: used incorrectly, this can be very memory expensive and cause performance issues.**

```
function doMyThing() {
    const stack = new Error().stack
    // Queues the function to be invoked in 1000ms on the event loop.
    setTimeout(() => {
        console.log(stack)
    }, 1000)
}

doMyThing()
```

There are far more sophisticated ways of capturing the stacktrace. However, what's important to know is that stacktraces are lost on the event loop boundary.

Run Demo

Name function parameters

```
function doMyThing() {
    setTimeout(() => {
        console.log('NOT NAMED: ' + new Error().stack)
    }, 50)
}
doMyThing()
```

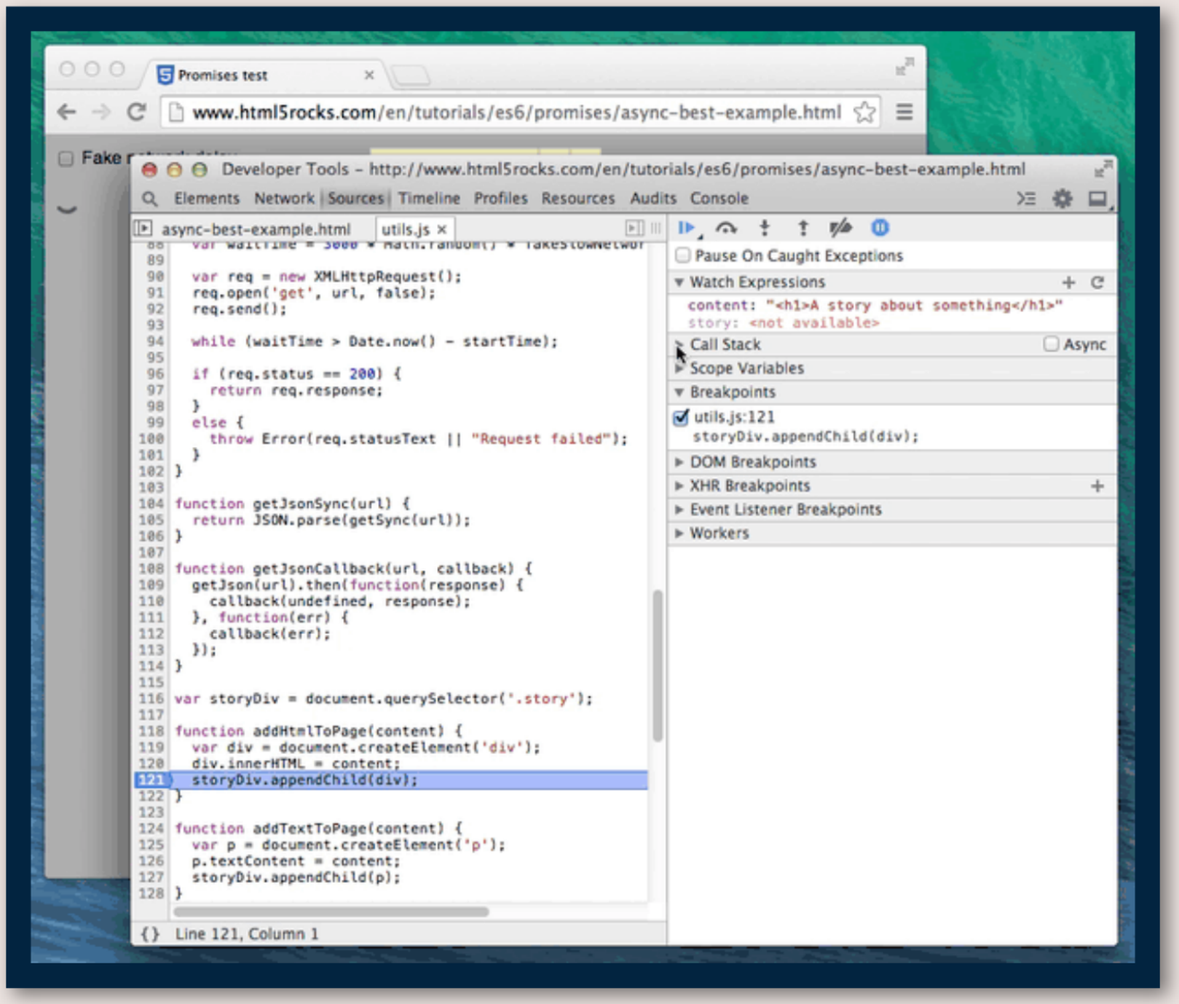
Instead, provide a name for your function parameter to improve stack trace

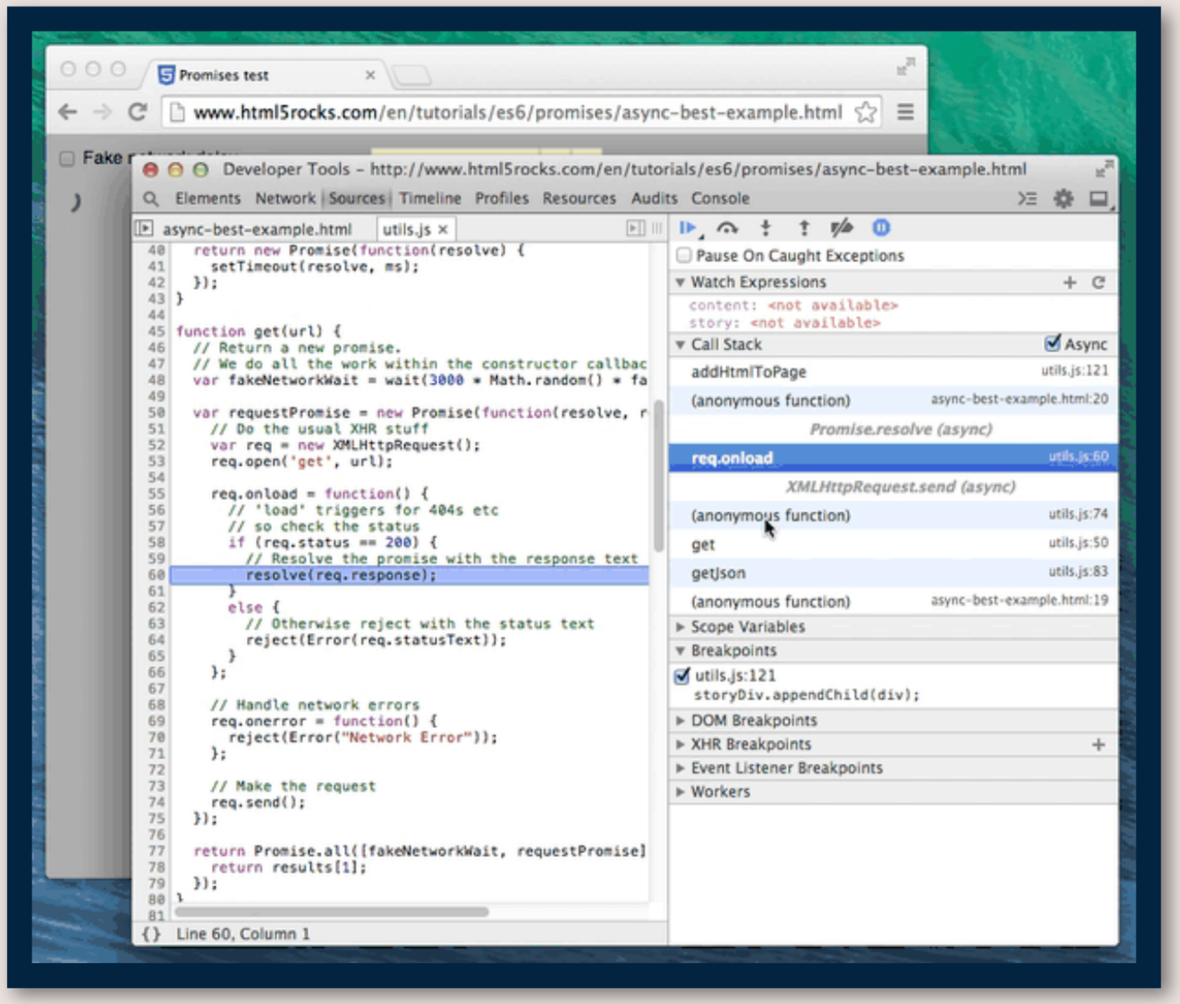
```
function doMyThing() {
    setTimeout(function fooIsMyName() {
        console.log('    NAMED: ' + new Error().stack)
    }, 50)
}

doMyThing()
```

Enable async stacktraces in chrome

source: http://www.html5rocks.com/en/tutorials/developertools/async-call-stack/





Thank you!

@joeandaverde