Bulletproof Node.js coding

<http://stella.laurenzo.org/2011/03/bulletproof-node-js-coding/>

**bulletproof node.js codingmarch 21st, 2011**

I’ve been actively doing node.js coding for about 4 months now. I’m working with a couple of others on a suite of mobile apps and my time has been split between building the Android client (one of my partners drew the long straw and got ios for this round) and building out the node.js based backend. It’s currently a Node.js + CouchDB + Redis server that combines user auth/account management with real-time signaling between connected clients. The core component, the “sessionserver” exposes no HTML ui and is really just a combination of JSON-based services, background agents and client signaling shims that speak in WebSockets and HTTP long polling.

The details of what I’m building aren’t very important to the topic I want to talk about here, but I mention them because, based on my experience, the sessionserver is a pretty core idiomatic node.js usage. This layer contains no view engine, no HTML templating, no sharing code with clients, etc. It is a raw communication crunching engine, representing a pretty pure node.js use case that may be worth some study. Throughout this post, I’ll be posting excerpts from this project instead of contrived examples as much as possible.

When I first started doing node.js coding, my first thought was “Wow! This is insanely powerful but it is really easy to slice your toes off!” It turned out that was also my second, third, and 150th thought as well! Right around the time that I started the third refactoring/rewrite of the sessionserver, I felt like I had gotten a feel for how to write bulletproof code and I thought it would be worth sharing some of the style and conventions I came to adopt. (As an aside, when learning a fundamentally new and different technology, never expect your first or second attempt to be any good straight out of the gate). It was actually kind of funny: Pretty early on, I found that I was crashing my process so much that I wired it up so that it would play a loud door slamming sound on abnormal exit. I heard that sound enough that it kind of got stock in my head and I found myself humming a melody that I’d made up to the steady beat of the slamming door. Seriously, it was that bad.

I’ve long since been a believer that no matter what the language or environment, developing a bulletproof coding style and conventions for how you approach the code is one of the most critical parts of the learning process. We all know there are an infinite number of ways to write the same chunk of logic, and after a fashion, many of them can even be considered good and reasonable. In my opinion, however, the best styles are those that, when followed, make it difficult or impossible to code most common types of bugs. Some of the most powerful features of a language or environment can also be the most deadly when misapplied. A bulletproof style will balance these features so that you get all of the power but it is difficult to abuse. In addition, dangerous, high-octane areas are properly cordoned off as such, and the style will also fill in for some of the inherent weaknesses of the language.

Since we’re talking about JavaScript, involving single-process asynchronous coding meant to serve thousands of connected clients at a time, any little tricks that can make us more reliable at producing good, working code are a huge bonus. I’m not going to spend much time on general JavaScript coding style. Instead, I’m going to focus on what conventions I applied to tame the callback-based asynchronous world that is node.js.

There are several “macro” solutions to writing more robust node.js code:

* [CoffeScript](http://jashkenas.github.com/coffee-script/): Defines a new language (“CoffeScript”) that compiles down to JavaScript.  It’s pretty neat and fills in a lot of the gaps at the language level.
* [node-fibers](https://github.com/laverdet/node-fibers): Adds the concept of “fibers” to node so that asynchronous code can be written in an imperative style.

In addition, I’ve come across some library level patterns that are also good if applied in the right context:

* Promises: There are various promise libraries floating around.  While from before my time, there was promise support in the very early days of node core.  Now its just a pattern people apply if they want to.
* [Tim Caswell’s “Do” library](http://howtonode.org/do-it-fast)

All of these are quite good and worth looking into.  I generally prefer solutions that work with the toolset instead of trying to replace it, and the library solutions certainly fit the bill.  Be careful about picking your core metaphors, however – they will stick with you for the life of your software.

My goals for writing these tidbits down is to share what I’ve learned and to stimulate a conversation about good node.js programming practices.  If you agree or disagree with anything I present, either leave some comments or start a discussion on the [node.js mailing list](http://groups.google.com/group/nodejs).  We all benefit from talking about this stuff more.

Here are the learnings that I’ve taken away from my odyssey with node.js thus far:

1. Return on the last statement
2. Put your callbacks in sequence
3. Define a respond function for complex logic
4. Centralize your exception handling
5. Embrace a functional coding style with futures or promises
6. Differentiate between system interfaces and user interfaces
7. Examine dependencies closely
8. Prefer copying simple, idiomatic code locally
9. Read the source but code to the docs
10. Write good tests

**1. Return on the Last Statement**

This one’s easy but it happens everywhere.  How many times have you done something like this:

|  |  |
| --- | --- |
|  | function doSomething(response, callback) { |
|  | doSomeAsyncCall('abc', 123, function(err, result) { |
|  | if (err) { |
|  | callback(err); |
|  | } |
|  | callback(null, result); |
|  | }); |
|  | } |

[**view raw**](https://gist.github.com/stellaeof/880102/raw/98f77a46894ccebdc25944d6bd188cc0a868c8f0/return-example-bad.js)[**return-example-bad.js**](https://gist.github.com/stellaeof/880102#file-return-example-bad-js) hosted with ❤ by [**GitHub**](https://github.com/)

The problem with this code is that on error, you are calling your callback with the error and the result (most likely null/undefined). This is almost always a violation of the declared API and will cause all manner of badness to happen on error. Making it worse, error paths are notoriously under-tested. You will almost certainly be hearing the door slamming in response to this one. While its easy to spot in a simple function like this, many real world cases are not so obvious. You could choose to just add a “return;” after “callback(err)”, but there is a better way if you can get your eye used to seeing it.

|  |  |
| --- | --- |
|  | function doSomething(response, callback) { |
|  | doSomeAsyncCall('abc', 123, function(err, result) { |
|  | if (err) { |
|  | return callback(err); |
|  | } |
|  | return callback(null, result); |
|  | }); |
|  | } |

[**view raw**](https://gist.github.com/stellaeof/880102/raw/98f77a46894ccebdc25944d6bd188cc0a868c8f0/return-example-good.js)[**return-example-good.js**](https://gist.github.com/stellaeof/880102#file-return-example-good-js) hosted with ❤ by [**GitHub**](https://github.com/)

Here, I take advantage of the fact that in JavaScript we can return anything (even undefined) and I wrap the terminating action and the intent to leave the function into one statement. I’ve found that once my eye gets used to seeing a “return …” as the last line in any control flow situation, it is much easier for me to visually pick out logic errors like the one above. To make this bulletproof, I’ve just gotten into the habit that if my function has any kind of control flow, I make the last statement of every branch be a return statement that returns whatever it is doing. This is usually a garbage value but the point is to make it appear visually as a “we’re done here” so that the next time you don’t see that pattern alarms are going off in your head.

Look around on github for 20 minutes. I bet you can find instances of this class of error in places that will really make you worried (although you may find fewer – I have emailed authors when I’ve found this pattern after reviewing their code).

**2. Put Your Callbacks in Sequence**

If LISP stands for Lost-In-Stupid-Parenthesis, then node should properly have been an acronym for Burried-In-Incomprehensible-Callbacks. BIIC isn’t as cool as NODE, though, so I imagine we should just start fixing the problem rather than renaming anything. It’s not just a problem of visual clutter — deeply nested functions produce brittle code with hard to find errors. There are other code organization techniques further down, but being able to un-nest your functions in a readable way is core to any functional programming. For this tip, we’ll take a quick look back at programming with Scheme, which is like the mother that JavaScript was separated from at birth. She got her father’s braces and her mother’s lexical scoping. Poor thing. No wonder she’s always in therapy.

When programming Scheme, you start with a simple function. Then you need to do something recursive, so you just code another function inline, and then… sound familiar? After nesting about one deep in Scheme, you almost always end up refactoring your outer function into a let-expression that takes all of those nested functions and puts them in sequence with names. I’ve started doing the equivalent thing with JavaScript and have found that my functions are a lot easier to read and manipulate. For this example, I’m digging out something a little older since most of my newer stuff is using futures and doesn’t exactly match the traditional node callback pattern:

|  |  |
| --- | --- |
|  | /\*\* |
|  | \* Add or update an object in the session and invoke |
|  | \* callback(err) when complete. updates is an array |
|  | \* of {objectId:..., value:...} |
|  | \*/ |
|  | updateObjects: function(updates, callback) { |
|  | if (!Array.isArray(updates)) return callback(new Error('updates must be an array')); |
|  |  |
|  | var self=this, cn=self.cn; |
|  |  |
|  | // Initialize the commands |
|  | var commands=cn.multi(), |
|  | publishChannels={}; |
|  |  |
|  | // First grab a new txid |
|  | return self.\_nextTxId(withTransactionId); |
|  |  |
|  | // -- callbacks |
|  | function withTransactionId(err, txid) { |
|  | if (err) return callback(err); |
|  | updateRest(0, txid); |
|  | } |
|  |  |
|  | function updateRest(index, txid) { |
|  | if (index>=updates.length) { |
|  | // Termination condition. Commit the commands. |
|  | return commitCommands(txid); |
|  | } |
|  |  |
|  | // Validate the record |
|  | var updateRecord=updates[index], |
|  | objectId=updateRecord.objectId, |
|  | object=updateRecord.value; |
|  | if (!objectIdIsValid(objectId)) return callback(new Error('Illegal ObjectId: "' + objectId + '"')); |
|  | if (object===null || 'object'!==typeof(object)) { |
|  | return callback(new Error('Illegal object value "' + object + '"')); |
|  | } |
|  |  |
|  | // Get it as json |
|  | var objectJson; |
|  | try { |
|  | objectJson=JSON.stringify(object); |
|  | } catch (e) { |
|  | return callback(e); |
|  | } |
|  |  |
|  |  |
|  | // Queue commands for updating this session |
|  | commands.hset(self.\_contentsKey(), objectId, objectJson); // Add to contents hash |
|  | commands.zadd(self.\_logKey(), txid, objectId); // Add to the txlog |
|  | commands.zrem(self.\_expungeKey(), objectId); // Undelete previous |
|  | //commands.publish(self.\_updateChannel(), self.\_updateMessage(txid)); // Publish update |
|  | publishChannels[self.\_updateChannel()]=true; |
|  |  |
|  | // Now for each published link, do similar operations |
|  | cn.smembers(self.\_publishKey(objectId), function(err, results) { |
|  | if (err) return callback(err); |
|  | var fqObjectId=objectId + '@' + self.sessionId; |
|  |  |
|  | // Each result is a Buffer containing a session id |
|  | results.forEach(function(result) { |
|  | var tgtSessionId=result.toString(); |
|  | commands.zadd(self.\_logKey(tgtSessionId), txid, fqObjectId); |
|  | commands.zrem(self.\_expungeKey(tgtSessionId), fqObjectId); |
|  | //commands.publish(self.\_updateChannel(tgtSessionId), self.\_updateMessage(txid)); |
|  | publishChannels[self.\_updateChannel(tgtSessionId)]=true; |
|  | }); |
|  |  |
|  | // Loop |
|  | return updateRest(index+1, txid); |
|  | }); |
|  | } |
|  |  |
|  | function commitCommands(txid) { |
|  | // Each channel that was marked to publish to needs a publish |
|  | var publishMessage=self.\_updateMessage(txid); |
|  | Object.keys(publishChannels).forEach(function(channelName) { |
|  | commands.publish(channelName, publishMessage); |
|  | }); |
|  |  |
|  | commands.exec(function(err, replies) { |
|  | return callback(err); |
|  | }); |
|  | } |
|  | }, |

[**view raw**](https://gist.github.com/stellaeof/880102/raw/98f77a46894ccebdc25944d6bd188cc0a868c8f0/explicit-callbacks-example.js)[**explicit-callbacks-example.js**](https://gist.github.com/stellaeof/880102#file-explicit-callbacks-example-js) hosted with ❤ by [**GitHub**](https://github.com/)

The key thing to note is the first few lines of the function. I do some basic setup and then immediately return by calling an asynchronous function that takes as a callback a function I have defined further below. This uses a trick of JavaScript that may be in bad taste, but I like it: Any functions defined by the form “function name() { }” (as opposed to with a var declaration) are available immediately upon entering the containing function (ie. control does not have to pass through them to define them). This is just some stylistic sugar that let’s you keep your code completely linear: the function starts at the top and proceeds through callbacks in a roughly downward motion. However you actually position your callbacks, however, the key point is that your code will be much more readable and maintainable if you stop using inline callbacks for non-trivial flows and use named callbacks defined in the outer level function instead. I generally stick to the rule that if there is any control flow, recursion or invocation of other asynchronous functions in a callback, it needs to be broken out to be its own named callback. You will also find that once this is done, it becomes trivial to introduce asynchronous recursion to deal with lists and such in a readable fashion.

**3. Define a respond function for complex logic**

If you have a standard node callback-based function with more than two ways to complete (one for errors and one for successful results), consider defining a secondary “respond” function to guard against hard to find situations where your mild-mannered control logic finishes more than precisely once.

|  |  |
| --- | --- |
|  | /\*\* |
|  | \* Sends a message to a queue. If recipient is an array, then multiple messages are put |
|  | \* on the queue. Message can either be a simple string or a compound object of a type |
|  | \* that the given queue understands. Invoke callback(err) on completion. |
|  | \* @param {Object} queue |
|  | \* @param {Object} recipient |
|  | \* @param {Object} message |
|  | \*/ |
|  | sendMessage: function(queue, recipient, message, callback) { |
|  | var self=this; |
|  | var queueType=QUEUE\_TYPES[queue]; |
|  | if (!queueType) { |
|  | return respond('Illegal queue type ' + queueType); |
|  | } |
|  |  |
|  | var queueKey=QUEUE\_KEY\_PREFIX + queueType; |
|  | var notifyChannel=NOTIFY\_CHANNEL\_PREFIX + queueType; |
|  | var commands=self.shared().multi(); |
|  | var wasEmpty=false; |
|  | var depth=0; |
|  |  |
|  | if (!Array.isArray(recipient)) recipient=[recipient]; |
|  | recipient.forEach(function(r) { |
|  | var queueObject={ |
|  | queue: queueType, |
|  | recipient: r, |
|  | timestamp: Date.now(), |
|  | body: message |
|  | }; |
|  | var queueJson=JSON.stringify(queueObject); |
|  | commands.lpush(queueKey, queueJson, function(err, count) { |
|  | if (err) return respond(err); |
|  |  |
|  | count=Number(count); |
|  | depth=count; |
|  | if (count===1) { |
|  | // We transitioned to non-empty |
|  | wasEmpty=true; |
|  | } |
|  | }); |
|  | }); |
|  |  |
|  | // -- send it on |
|  | commands.exec(function(err) { |
|  | if (err) return respond(err); |
|  | if (wasEmpty) { |
|  | self.shared().publish(notifyChannel, 'notempty', function(err) { |
|  | return respond(err); |
|  | }); |
|  | } else { |
|  | return respond(err); |
|  | } |
|  | }); |
|  |  |
|  |  |
|  | function respond(err) { |
|  | if ('function' === typeof callback) { |
|  | callback(err, depth); |
|  | callback = null; |
|  | } |
|  | } |
|  | }, |

[**view raw**](https://gist.github.com/stellaeof/880102/raw/98f77a46894ccebdc25944d6bd188cc0a868c8f0/respond-function-example.js)[**respond-function-example.js**](https://gist.github.com/stellaeof/880102#file-respond-function-example-js) hosted with ❤ by [**GitHub**](https://github.com/)

This is an older example and I don’t generally hold it up as a bastion of good code. In particular, I should have broken out more named callbacks to distinguish between the synchronous and asynchronous parts of the flow. The key thing to note, though, is that there are multiple “ways out” of the function where the callback can be invoked. Instead of adding logic everywhere to determine if we’ve error’ed, responded yet, or determining if the callback is even defined, I use an explicit respond(…) function locally which invokes the callback with the results and then clears it so it won’t be invoked again. An even better solution would have been to add a warning if invoked more than once.

The rule here is in the same vein as those that come before. If your function is simple, keep it simple and don’t add an explicit respond function. However, if the control flow is getting a little dicey (and itself cannot be simplified), protect yourself by making the callback an explicit local respond function.

**4. Centralize your exception handling**

Functional programming in node is a lot of fun, expressive and compact except for one part: exception handling. I don’t really see this talked about that much, but in my opinion the lack of a coherent way of dealing with errors and exceptions is node’s biggest weakness. Node-fibers takes the approach of switching to a completely imperative style to achieve this, but I prefer to stay with a functional style and define a coherent exception handling structure.

I could write an entire post just on this topic (and maybe will one day) but I’ll just cover the high points here. The problem with error codes (which node core is based on) is that for higher level logic, the code that detects the error (ie. the first responder) is almost invariably not in the right position to determine what to do about the error condition. This is where try/catch structures in threaded systems make more sense. Someone up the stack will typically know what to do about the error.

The problem with an asynchronous system like node, however is that every time one of your callbacks or EventEmitter listeners gets invoked, it is often either at the very top level of the event loop or being called by some foreign code that is different from the code that attached the listener (the thing that attached the listener is probably in a better position to deal with the failure than whatever random execution context you ended up in). If you throw an exception in these contexts, it is a good bet that the program will terminate. Since JavaScript has a pretty impressive array of ways that mild-mannered looking statements can throw runtime errors, this problem is worse than in an environment like C, where if I’m careful with my pointers and don’t divide by zero I’m ok. Yes, unit tests help but its kind of like trying to plug all of the holes in a strainer when what you really want is a bowl.

For this tip, you are going to need library support of some kind. What is needed is a way to define a Block with an Error Handler and be able to tear this off and take it with you when your callbacks go into foreign territory. Then when they raise an exception, the exception gets routed back to the Block that was in effect when the callback was sent out to do its master’s bidding. I found that most of the solutions out there munged Futures, Promises, Fibers, etc together with this simple need to define an exception handling Block. The following snippet defines a Block class that fulfills what I’m looking for:

|  |  |
| --- | --- |
|  | /\*\* |
|  | \* Block class is used for routing errors to higher level logic. |
|  | \*/ |
|  | function Block(errback) { |
|  | this.\_parent=Block.current; |
|  | this.\_errback=errback; |
|  | } |
|  | Block.current=null; |
|  |  |
|  | /\*\* |
|  | \* Wrap a function such that any exceptions it generates |
|  | \* are sent to the error callback of the Block that is active |
|  | \* at the time of the call to guard(). If no Block |
|  | \* is active, just returns the function. |
|  | \* |
|  | \* Example: stream.on('end', Block.guard(function() { ... })); |
|  | \*/ |
|  | Block.guard=function(f) { |
|  | if (this.current) return this.current.guard(f); |
|  | else return f; |
|  | }; |
|  |  |
|  | /\*\* |
|  | \* Begins a new Block with two callback functions. The first |
|  | \* is the main part of the block (think 'try body'), the |
|  | \* second is the rescue function/error callback (think 'catch'). |
|  | \* The terminology follows Ruby for no other reason than that |
|  | \* Block, begin and rescue describe an exception handling |
|  | \* paradigm and are not reserved words in JavaScript. |
|  | \*/ |
|  | Block.begin=function(block, rescue) { |
|  | var ec=new Block(rescue); |
|  | return ec.trap(block); |
|  | }; |
|  |  |
|  | /\*\* |
|  | \* Returns a function(err) that can be invoked at any time to raise |
|  | \* an exception against the now current block (or the current context |
|  | \* if no current). Errors are only raised if the err argument is true |
|  | \* so this can be used in both error callbacks and error events. |
|  | \* |
|  | \* Example: request.on('error', Block.errorHandler()) |
|  | \*/ |
|  | Block.errorHandler=function() { |
|  | // Capture the now current Block for later |
|  | var current=this.current; |
|  |  |
|  | return function(err) { |
|  | if (!err) return; |
|  | if (current) return current.raise(err); |
|  | else throw err; |
|  | }; |
|  | }; |
|  |  |
|  | /\*\* |
|  | \* Raises an exception on the Block. If the block has an |
|  | \* error callback, it is given the exception. Otherwise, |
|  | \* raise(...) is called on the parent block. If there is |
|  | \* no parent, the exception is simply raised. |
|  | \* Any nested exceptions from error callbacks will be raised |
|  | \* on the block's parent. |
|  | \*/ |
|  | Block.prototype.raise=function(err) { |
|  | if (this.\_errback) { |
|  | try { |
|  | this.\_errback(err); |
|  | } catch (nestedE) { |
|  | if (this.\_parent) this.\_parent.raise(nestedE); |
|  | else throw nestedE; |
|  | } |
|  | } else { |
|  | if (this.\_parent) this.\_parent.raise(err); |
|  | else throw(err); |
|  | } |
|  | }; |
|  |  |
|  | /\*\* |
|  | \* Executes a callback in the context of this block. Any |
|  | \* errors will be passed to this Block's raise() method. |
|  | \* Returns the value of the callback or undefined on error. |
|  | \*/ |
|  | Block.prototype.trap=function(callback) { |
|  | var origCurrent=Block.current; |
|  | Block.current=this; |
|  | try { |
|  | var ret=callback(); |
|  | Block.current=origCurrent; |
|  | return ret; |
|  | } catch (e) { |
|  | Block.current=origCurrent; |
|  | this.raise(e); |
|  | } |
|  | }; |
|  |  |
|  | /\*\* |
|  | \* Wraps a function and returns a function that routes |
|  | \* errors to this block. This is similar to trap but |
|  | \* returns a new function instead of invoking the callback |
|  | \* immediately. |
|  | \*/ |
|  | Block.prototype.guard=function(f) { |
|  | if (f.\_\_guarded\_\_) return f; |
|  | var self=this; |
|  | var wrapped=function() { |
|  | var origCurrent=Block.current; |
|  | Block.current=self; |
|  | try { |
|  | var ret=f.apply(this, arguments); |
|  | Block.current=origCurrent; |
|  | return ret; |
|  | } catch (e) { |
|  | Block.current=origCurrent; |
|  | self.raise(e); |
|  | } |
|  | }; |
|  | wrapped.\_\_guarded\_\_=true; |
|  | return wrapped; |
|  | }; |

[**view raw**](https://gist.github.com/stellaeof/880102/raw/98f77a46894ccebdc25944d6bd188cc0a868c8f0/control-block.js)[**control-block.js**](https://gist.github.com/stellaeof/880102#file-control-block-js) hosted with ❤ by [**GitHub**](https://github.com/)

(I chose the Block/Rescue terminology not because I really have any fondness for Ruby but because it is an implementation of scoped exception handling that uses words that are not reserved in JavaScript) Here is an example of using a Block to centralize exception handling. In this case, this is a connect based middleware and the provided “next” function from the framework is a perfect exception handler: it returns an appropriate error to the http client. If we had other cleanup that needed to be done, just define your own function(err) {} callback instead and then invoke next(err) when done. You could also use inline functions in the call to Block.begin (making it resemble a try/catch visually), but I chose to use named callbacks here for readability.

|  |  |
| --- | --- |
|  | function handleUserAgent(req, res, next) { |
|  | return Block.begin(process, next); |
|  |  |
|  | function process() { |
|  | return jsonifyRequest(req).force(withRequest); |
|  | } |
|  |  |
|  | function withRequest(requestObj) { |
|  | var r=validators.UserAgentRecord(requestObj, {fix:true}); |
|  | if (!r.valid) { |
|  | res.writeHead(400); |
|  | return res.end('Invalid request object: ' + r.reason); |
|  | } |
|  |  |
|  | var uar=r.object; |
|  | if (uar.token) { |
|  | // Verify |
|  | //return handler.verifyUserAgent(uar); |
|  | throw new Error('verifyUserAgent not yet implemented'); |
|  | } else { |
|  | // Create |
|  | uar.token=null; |
|  | uar.type='auth'; // TODO: Maybe support unauth in the future? |
|  | return handler.createUserAgent(uar).force(withUserAgent); |
|  | } |
|  | } |
|  |  |
|  | function withUserAgent(userAgent) { |
|  | var r=validators.UserAgentRecord(userAgent, {fix:true}); |
|  | return respondJson(r.object, res); |
|  | } |
|  | } |

[**view raw**](https://gist.github.com/stellaeof/880102/raw/98f77a46894ccebdc25944d6bd188cc0a868c8f0/control-block-middleware-example1.js)[**control-block-middleware-example1.js**](https://gist.github.com/stellaeof/880102#file-control-block-middleware-example1-js) hosted with ❤ by [**GitHub**](https://github.com/)

This example also uses a Future class which I’ll cover in a bit. The key thing to keep in mind is that any exception thrown by code in or called by the process() function will be routed to the rescue handler (in this case next). In order to get a callback into the block scope, it should be wrapped by calling Block.guard(originalFunction). This will capture the current Block at the time that Block.guard is called and reestablish it for the duration of any call to originalFunction. The Future class does this internal to the force(…) call, which allows me to rest certain that anything I place as the target of a force(…) will have its exceptions routed appropriately. More on that later.

Here’s an example of explicitly capturing the block in your callbacks. In this case, we are invoking an HTTP request, accumulating the text results and resolving a Future with a constructed CouchResponse object (which does some parsing and other things that could conceivably throw an exception).

|  |  |
| --- | --- |
|  | request: function(options) { |
|  | var future=new Future(); |
|  |  |
|  | var req=http.request(options, function(res) { |
|  | var text=''; |
|  | res.setEncoding('utf8'); |
|  | res.on('data', function(chunk) { |
|  | text+=chunk; |
|  | }); |
|  | res.on('end', Block.guard(function() { |
|  | future.resolve(new CouchResponse(res, text)); |
|  | })); |
|  | res.on('error', Block.errorHandler()); |
|  | }); |
|  | req.on('error', Block.errorHandler()); |
|  | req.end(); |
|  |  |
|  | return future; |
|  | } |

[**view raw**](https://gist.github.com/stellaeof/880102/raw/98f77a46894ccebdc25944d6bd188cc0a868c8f0/control-block-http-example1.js)[**control-block-http-example1.js**](https://gist.github.com/stellaeof/880102#file-control-block-http-example1-js) hosted with ❤ by [**GitHub**](https://github.com/)

There are still a couple of places in this example where an unexpected exception would crash the process:

* directly within the “function(res)” callback
* in the ‘data’ callback

I could have wrapped Block.guard statements around these bits as well but chose not to because it costs a little extra and I am 100% confident that a failure here is a critical breakage and is completely covered by unit tests. The ‘end’ handler, however, does some stuff that I can’t immediately see (and I happen to know it contains a JSON.parse call) so I protect it with guard. Finally, I use the block’s standard errorHandler() callbacks to catch request and response error events. I’ve found that this simple pattern of centralizing exception handling makes it very easy to visually understand where exceptions are going and route them at the levels where it makes sense. You can also nest calls to Block.begin. This is useful in framework code that needs to go off and do some other work in response to something the Block initiated but not intrinsically owned by it.

**5. Embrace a functional coding style with futures or promises**

I actually like node’s callback style a lot for low level stuff — you know for those times when you feel like coding in C is the right thing to be doing and you’re thankful that someone let’s you operate at that level but without malloc/free. For higher level logic/abstractions, though, I prefer something with a bit more functional heritage. A lot of people have used Promises which are just a construct for converting a callback into a return value. You return a Promise instead of invoking a Callback and then you can ask the promise to give you its result. Future’s are similar as far as metaphors go and I prefer them. A Future has two intrinsic operations: resolve and force. Resolve sets the value on the future and force either gets the value if it is immediately available or gives it to you later when it is available. Given the Block based exception handling I illustrated above, my Future class doesn’t really need to think much about capturing and propagating exceptions, so its pretty simple. It does build on the Block by making sure to call Block.guard(…) to wrap any functions that are bound to be invoked as callbacks later by force(…). Here’s the class:

|  |  |
| --- | --- |
|  | /\*\* |
|  | \* A Future class as per the literature on the topic. |
|  | \* The two main operations are force() and resolve(). |
|  | \*/ |
|  | function Future(resolution) { |
|  | this.\_resolved=false; |
|  | this.\_errored=false; |
|  | this.\_resolution=null; |
|  | this.\_pending=null; |
|  | if (arguments.length>0) { |
|  | // Create a resolved future |
|  | this.resolve(resolution); |
|  | } |
|  | } |
|  | /\*\* |
|  | \* Cast an arbitrary value to a Future. If already a Future, |
|  | \* just return it. Otherwise, return a new Future resolved |
|  | \* with the value. |
|  | \*/ |
|  | Future.cast=function(futureOrLiteral) { |
|  | if (futureOrLiteral instanceof Future) return futureOrLiteral; |
|  | else return new Future(futureOrLiteral); |
|  | }; |
|  | Future.prototype={}; |
|  |  |
|  | /\*\* |
|  | \* Invokes the callback(result) upon resolution of the future. |
|  | \* Return true if force executed immediately, false if pended. |
|  | \* |
|  | \* If the callback is pended, it is wrapped with Block.guard |
|  | \* so that any exceptions it throws are routed to the Block |
|  | \* in effect at the time of the call to force(). |
|  | \*/ |
|  | Future.prototype.force=function(callback) { |
|  | if (this.\_resolved) { |
|  | if (callback) callback(this.\_resolution); |
|  | return true; |
|  | } else if (callback) { |
|  | // Pend it. |
|  | var pended=Block.guard(callback); |
|  | if (!this.\_pending) this.\_pending=[pended]; |
|  | else this.\_pending.push(pended); |
|  | return false; |
|  | } else { |
|  | return false; |
|  | } |
|  | }; |
|  |  |
|  | /\*\* |
|  | \* Resolves the future. Any pended force callbacks are |
|  | \* executed immediately. Any future calls to force() will |
|  | \* invoke their callbacks immediately. |
|  | \*/ |
|  | Future.prototype.resolve=function(resolution) { |
|  | if (this.\_resolved) { |
|  | throw new Error('Logic error. Future resolved multiple times.'); |
|  | } |
|  | this.\_resolved=true; |
|  | this.\_resolution=resolution; |
|  |  |
|  | if (this.\_pending) { |
|  | this.\_pending.forEach(function(pended) { |
|  | pended(resolution); |
|  | }); |
|  | } |
|  | }; |
|  |  |
|  | /\*\* |
|  | \* Return a new Future whose resolution is dependent on |
|  | \* the resolution of this future. When this future is |
|  | \* resolved, the transformer callback will be invoked with |
|  | \* its resolution and the callback result will become the |
|  | \* resolution of the new Future returned by this method. |
|  | \*/ |
|  | Future.prototype.chain=function(transformer) { |
|  | var chained=new Future(); |
|  | this.force(function(resolution) { |
|  | chained.resolve(transformer(resolution)); |
|  | }); |
|  | return chained; |
|  | }; |
|  | /\*\* |
|  | \* Forward the resolution from this future to another future. |
|  | \* This future is forced and the resolution is resolved on |
|  | \* the passed future. |
|  | \*/ |
|  | Future.prototype.forward=function(otherFuture) { |
|  | this.force(function(resolution) { |
|  | otherFuture.resolve(resolution); |
|  | }); |
|  | }; |

[**view raw**](https://gist.github.com/stellaeof/880102/raw/98f77a46894ccebdc25944d6bd188cc0a868c8f0/future.js)[**future.js**](https://gist.github.com/stellaeof/880102#file-future-js) hosted with ❤ by [**GitHub**](https://github.com/)

The key advice here is not necessarily to use my Future class, but to use someone’s Future or Promise implementation. I like mine because it is so brain-dead simple and integrates with Block.guard so that when I’m scanning my code and I see a function being passed to a force() call, I can mentally tell myself “this function is safe for exceptions to be thrown from.”

There are examples of using the Future in the previous sections.

**6. Differentiate between system interfaces and user interfaces**

This one is more of philosophical advice about using the right tool for the job. Some things are best done with node’s callback(err)/EventEmitter machinery and sometimes its better to use a higher level abstraction like a Future/Block. Don’t be afraid to use both. I tend to use the lower-level machinery for stuff that is interfacing with the system. For some reason it feels right to me to be passing around error codes in these situations, but this probably has more to do with the time I spent in C hacking on the Linux kernel than anything else. If you’re writing code to be consumed outside of your project, make sure it speaks the callback(err)/EventEmitter pattern since that is the lowest common denominator that every node programmer on the planet is going to intrinsically understand.

**7. Examine dependencies closely**

You can get a *little* cavalier in threaded environments like Java, Ruby or Python when it comes to relying on third party bits. After all you can always just catch Throwable right? Remember that in Node, everything you put into your project and call has the very real potential to kill you. Don’t just run the tests and assume a happy future. Look at the code and make a critical evaluation. If you get the feeling like its playing fast and loose with control flow, it probably is — and it might just kill you. Also, and I mean this with all respect to the node community, do not rely on popularity of a module to assume that others have given its internals a critical evaluation. Remember too that most of the node modules floating around on GitHub started as internal bits for someone else’s project and they have built-in assumptions to those ends.

I don’t mean to be too melodramatic here, but the point is simple: pulling in an external dependency is a lot more like inviting someone into your bed than into your living room. There are lot’s of great things that can come from it, but just be safe about it.

**8. Prefer copying simple, idiomatic code locally**

This runs counter to most of my experience in other environments and it might not hold up over time as the ecosystem evolves. For now, however, I generally prefer to take simple external dependencies, copy them locally and modify vs trying to share. There’s just no reason why we need to have one “copyObject”, “clone”, etc to rule them all. Find one that does what you want, make sure you understand it, stick it in your project and use it with a local require (require(‘./myCoolObjectCopy’)).

**9. Read the source but code to the docs**

The great thing about node is that the code is flayed open for all to see. And with most of the modules out on GitHub, its just a few clicks before you are reading anything. Just remember that all of those interesting bits in the source code are not necessarily part of the public api. Rely on the docs for what you are supposed to be calling. If you see something internally that you think should be part of the public api, email the appropriate people and ask/make a suggestion.

**10. Write good tests**

Really, however non-optional they may have been in other environments, they are not optional here. There are quite a few testing frameworks about, but I tend to use [nodeunit](https://github.com/caolan/nodeunit). Here’s a simple one to get you started:

|  |  |
| --- | --- |
|  | var nodeunit=require('nodeunit'); |
|  | var myModuleUnderTest=require('../lib/myModuleUnderTest'); |
|  |  |
|  | module.exports=nodeunit.testCase({ |
|  | 'test for smoke': function(test) { |
|  | test.done(); |
|  | }, |
|  | }); |

[**view raw**](https://gist.github.com/stellaeof/880102/raw/98f77a46894ccebdc25944d6bd188cc0a868c8f0/stub-testcase.js)[**stub-testcase.js**](https://gist.github.com/stellaeof/880102#file-stub-testcase-js) hosted with ❤ by [**GitHub**](https://github.com/)

For some reason, I always include a ‘test for smoke’ that does nothing as my first test. If there’s a parse error or some other setup problem, then its pretty obvious on the console because I’ll see the error and the line that says “test for smoke” ran successfully won’t be there.

Here’s my runtests.js file. I just customize this slightly (to add require paths, etc) and drop it into any project.

|  |  |
| --- | --- |
|  | #!/usr/bin/env node |
|  | require.paths.unshift(\_\_dirname + '/local/lib'); |
|  | process.chdir(\_\_dirname); |
|  |  |
|  | var files=process.argv.slice(2).map(function(v) { |
|  | return v; |
|  | }); |
|  |  |
|  | if (files.length===0) { |
|  | // Just run the whole directory |
|  | files=['local/test', 'test']; |
|  | } |
|  |  |
|  | var reporter=require('nodeunit').reporters.default; |
|  |  |
|  | reporter.run(files); |

[**view raw**](https://gist.github.com/stellaeof/880523/raw/69b8d0d4216d879596d11f2f2435ab37df3ad707/runtests.js)[**runtests.js**](https://gist.github.com/stellaeof/880523#file-runtests-js) hosted with ❤ by [**GitHub**](https://github.com/)

This entry was posted on Monday, March 21st, 2011 at 5:52 pm and is filed under [node.js](http://stella.laurenzo.org/category/blog/node-js/). You can follow any responses to this entry through the [RSS 2.0](http://stella.laurenzo.org/2011/03/bulletproof-node-js-coding/feed/) feed.You can [leave a response](http://stella.laurenzo.org/2011/03/bulletproof-node-js-coding/#respond), or [trackback](http://stella.laurenzo.org/2011/03/bulletproof-node-js-coding/trackback/) from your own site.

**17 responses**

http://1.gravatar.com/avatar/5bed5c28883cb3ba13b8241b06144451?s=32&d=http%3A%2F%2F1.gravatar.com%2Favatar%2Fad516503a11cd5ca435acc9bb6523536%3Fs%3D32&r=G

March 21st, 2011 at 7:32 pm  
***shaun* Says:**

“pulling in an external dependency is a lot more like inviting someone into your bed than into your living room. There are lot’s of great things that can come from it, but just be safe about it.”

Have you been hanging out with Isaac? :)

Nice post. Thank you.

http://0.gravatar.com/avatar/a2a33a305490ee5e6bddad4ba10400d4?s=32&d=http%3A%2F%2F0.gravatar.com%2Favatar%2Fad516503a11cd5ca435acc9bb6523536%3Fs%3D32&r=G

March 21st, 2011 at 10:21 pm  
[***stella***](http://stella.laurenzo.org/)**Says:**

:)  
No, we’ve never met, but I have a feeling we agree on a few things.

http://1.gravatar.com/avatar/f99519309d506e9747afb24ab89bff36?s=32&d=http%3A%2F%2F1.gravatar.com%2Favatar%2Fad516503a11cd5ca435acc9bb6523536%3Fs%3D32&r=G

March 22nd, 2011 at 2:49 am  
[***Guy***](http://www.cach.me/blog)**Says:**

This is a really great post. I like point 2 especially, I have been finding that my nodejs code becomes cluttered very quickly and this is a nice solution. Bookmarked ;) .

http://0.gravatar.com/avatar/4023fba6eaec22ed69367a11735989a7?s=32&d=http%3A%2F%2F0.gravatar.com%2Favatar%2Fad516503a11cd5ca435acc9bb6523536%3Fs%3D32&r=G

March 25th, 2011 at 11:27 am  
[***Elrasguno***](http://attnspan.com/)**Says:**

Hey there,  
Thanks so much for posting this. It’s helpful to see example implementations that have gone beyond “Hello World” and into the numerous rabbit holes that you’re clearly exploring, especially error handling. While I’ve done a lot of defensive coding, it was typically client side, and recovery from errors is trivial. With node, errors make everything very, very dead.

So, if any of you see this, and have a sec (especially you Stella), could you take a quick look at the beginnings of my node project? I’m really just looking for some constructive criticism, because as you each know, there just aren’t \_that\_ many people jumping in here. Here it is: <http://j.mp/epkNv5> The idea is that it’s a generic game server that passes commands on to whatever the game happens to be, and subsequently kicks back responses to whomever the client happens to be. Oh yeah, the game in this case is tic-tac-toe :) Thanks!

http://0.gravatar.com/avatar/814b91f4ca4eb8242c5abe05dcc29058?s=32&d=http%3A%2F%2F0.gravatar.com%2Favatar%2Fad516503a11cd5ca435acc9bb6523536%3Fs%3D32&r=G

March 25th, 2011 at 12:40 pm  
[***Peter Scott***](http://finger-tree.blogspot.com/)**Says:**

Your server got hammered when this was posted to Hacker News, and was timing out for a while there. It looks like you’re using Apache 2.2.14 with a Keep-Alive time of 15 seconds, which would explain the problem; Apache spawned too many threads and fell down. If you ever feel like digging into some server config files, here’s an article on how to fix it:

<http://www.kalzumeus.com/2010/06/19/running-apache-on-a-memory-constrained-vps/>

Thanks for the great article, by the way.

http://0.gravatar.com/avatar/a2a33a305490ee5e6bddad4ba10400d4?s=32&d=http%3A%2F%2F0.gravatar.com%2Favatar%2Fad516503a11cd5ca435acc9bb6523536%3Fs%3D32&r=G

March 25th, 2011 at 1:37 pm  
[***stella***](http://stella.laurenzo.org/)**Says:**

Thanks Peter. That was embarrassing, more so because I know better. I got the disk thrash email from my vps provider while drinking my coffee this morning and it was a quick fix from there. As you say, too many worker threads and other sub-optimal settings. I’m kicking myself because I have always built hand-crafted Apache configs but just set this one up relatively quickly a while back and forgot about it. It’s memory-stable now and I’ll go through the config with a fine-tooth comb this weekend.

http://1.gravatar.com/avatar/fafbca6072d78894a65888eee17302ec?s=32&d=http%3A%2F%2F1.gravatar.com%2Favatar%2Fad516503a11cd5ca435acc9bb6523536%3Fs%3D32&r=G

March 25th, 2011 at 1:53 pm  
***Bob* Says:**

Very nice write-up but reading white bolded text on a black background is really tiring on the eyes.

http://0.gravatar.com/avatar/a2a33a305490ee5e6bddad4ba10400d4?s=32&d=http%3A%2F%2F0.gravatar.com%2Favatar%2Fad516503a11cd5ca435acc9bb6523536%3Fs%3D32&r=G

March 25th, 2011 at 2:05 pm  
[***stella***](http://stella.laurenzo.org/)**Says:**

Thanks. Coming up with a new style has been on my list for a while for the reason you mention, but I haven’t gotten to it. For some reason when I built that style a while back, darkness was appealing to me. :)

http://0.gravatar.com/avatar/c5910cb2d2f81eb43a13e14b2087bc63?s=32&d=http%3A%2F%2F0.gravatar.com%2Favatar%2Fad516503a11cd5ca435acc9bb6523536%3Fs%3D32&r=G

March 25th, 2011 at 2:21 pm  
[***Sufian***](http://sigusrone.com/)**Says:**

There are many situations outside your particular example for #3 where running a function once is useful. I’ve found that extending the Function prototype with a once() method (or a better name for give me the function that forces idempotency on the first invocation) is useful.

Function.prototype.once = function(bindTarget) {  
var done = false;  
return (function() {  
if (!done) {  
this.apply(bindTarget, arguments);  
done = true;  
}  
}).bind(this);  
}

http://0.gravatar.com/avatar/a2a33a305490ee5e6bddad4ba10400d4?s=32&d=http%3A%2F%2F0.gravatar.com%2Favatar%2Fad516503a11cd5ca435acc9bb6523536%3Fs%3D32&r=G

March 25th, 2011 at 2:40 pm  
[***stella***](http://stella.laurenzo.org/)**Says:**

Good discussion on this over at hacker news: <http://news.ycombinator.com/item?id=2368658>

April 2nd, 2011 at 11:29 am  
[***6 Free E-Books and Tutorials for Learning and Mastering Node.js - Finding Out About***](http://www.findingoutabout.com/?p=15430)**Says:**

[...] finally, Bulletproof Node.js Coding is a concise but helpful collection of tips for Node.js [...]

April 2nd, 2011 at 12:01 pm  
[***6 Free E-Books and Tutorials for Learning and Mastering Node.js | JetLib News***](http://jetlib.com/news/2011/04/02/6-free-e-books-and-tutorials-for-learning-and-mastering-node-js/)**Says:**

[...] finally, Bulletproof Node.js Coding is a concise but helpful collection of tips for Node.js [...]

April 2nd, 2011 at 12:02 pm  
[***6 Free E-Books and Tutorials for Learning and Mastering Node.js | SEO College***](http://seocollege.org/news/?p=32483)**Says:**

[...] finally, Bulletproof Node.js Coding is a concise but helpful collection of tips for Node.js programmers. [...]

April 2nd, 2011 at 12:36 pm  
[***6 Free E-Books and Tutorials for Learning and Mastering Node.js***](http://feedproxy.google.com/~r/readwriteweb/~3/LeUahPmL5OY/6-free-e-books-on-nodejs.php)**Says:**

[...] finally, Bulletproof Node.js Coding is a concise but helpful collection of tips for Node.js [...]

April 3rd, 2011 at 6:26 am  
[***6 Free E-Books and Tutorials for Learning and Mastering Node.js | Scripting4U Blog***](http://www.scripting4u.com/2011/04/6-free-e-books-and-tutorials-for-learning-and-mastering-node-js/)**Says:**

[...] finally, Bulletproof Node.js Coding is a concise but helpful collection of tips for Node.js [...]

April 3rd, 2011 at 11:12 am  
[***6 Free E-Books and Tutorials for Learning and Mastering Node.js | IT News Wire***](http://www.itnewswire.info/2011/04/6-free-e-books-and-tutorials-for-learning-and-mastering-node-js.html)**Says:**

[...] finally, Bulletproof Node.js Coding is a concise but helpful collection of tips for Node.js programmers. Related News: 6 Free [...]

http://0.gravatar.com/avatar/aef49113114eb63c39b86ae8fe181d21?s=32&d=http%3A%2F%2F0.gravatar.com%2Favatar%2Fad516503a11cd5ca435acc9bb6523536%3Fs%3D32&r=G

April 3rd, 2011 at 1:58 pm  
[***Jens***](http://blitz.io/)**Says:**

If you want to validate your config, try out <http://blitz.io/> – you can also run performance tests – it’s in beta but already pretty slick…