ASYNC PATTERNS & STRATEGIES IN JAVASCRIPT

{ Jim Cowart }

Who am I?

- Jim Cowart (a.k.a. - @ifandelse)
- Chief Architect at appendTo
- I act like I write stuff: http://freshbrewedcode.com/jimcowart
- I write stuff: http://github.com/ifandelse

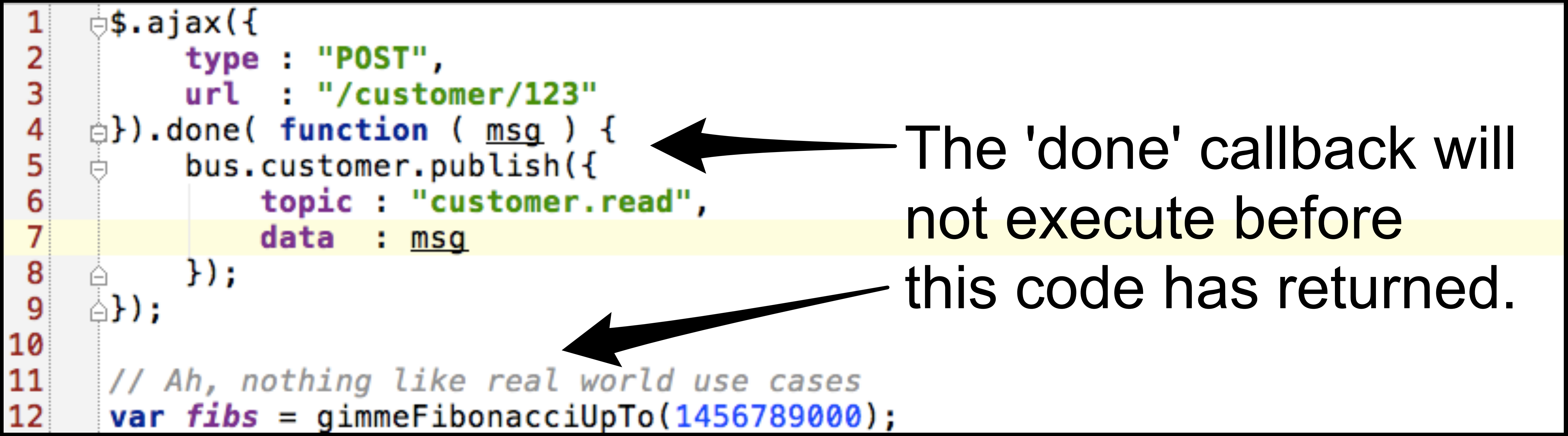
So why are we here?

- “We have a cultural bias towards blocking” code
- Asynchronous is far more than just AJAX response handlers
- What patterns can help us?
- What about good implementations of those patterns in popular libraries?

What does it mean to be asynchronous?

- JavaScript is single-threaded & runs in an event loop
- Events are queued - and will run when the loop is available
- Currently executing code can queue something to run later (but no sooner than the currently executing code has returned)

Obligatory Asynchronous Example



```
$.ajax({
    type : "POST",
    url  : "/customer/123"
}).done( function ( msg ) {
    bus.customer.publish({
        topic : "customer.read",
        data  : msg
    });
});

// Ah, nothing like real world use cases
var fibs = gimmeFibonacciUpTo(1456789000);
```

The 'done' callback will not execute before this code has returned.

"Events can be queued while code is running, but they can't fire until the runtime is free."
- Trevor Burnham (Async JavaScript)

Once you go async, you'll never return...

- Callbacks are the currency of asynchronous code
- Continuation-passing-style vs return values === very different design constraints
- Not safe to assume *how* a 3rd party lib will execute your callback (synchronously? asynchronously?)

Something to keep in mind

“You cannot reduce the complexity of a task beyond a given point. Once you reach that point, you can only shift the burden around.”

Tessler’s Law of Conservation of Complexity

Something to keep in mind

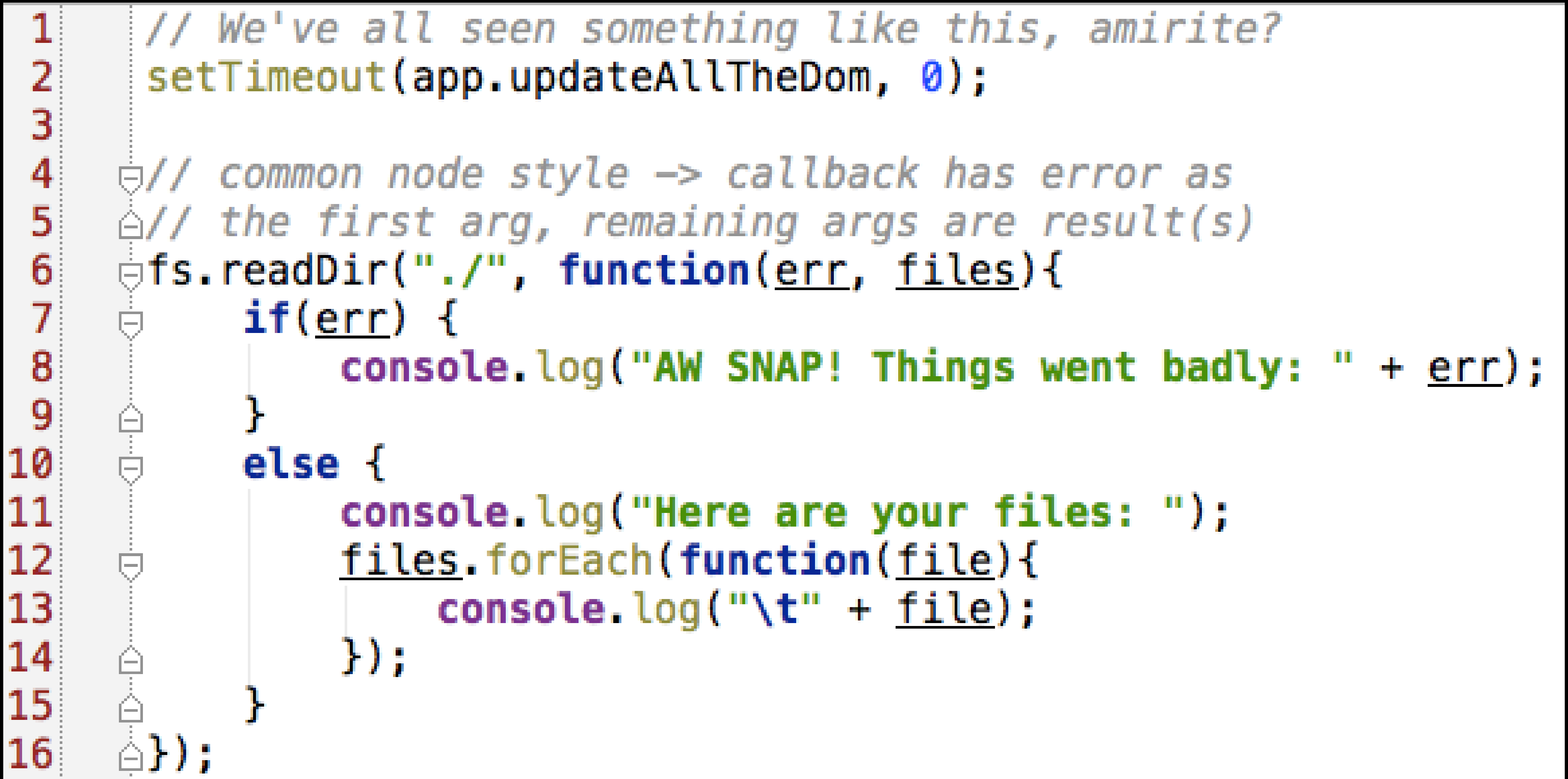
"You cannot reduce the complexity of a task beyond a given point. Once you reach that point, you can only shift the burden around."

Tessler's Law of Conservation of Complexity

So, to *where* are you shifting the burden?

The Future (of upcoming Examples)

Strategy #1 - Plain Callbacks



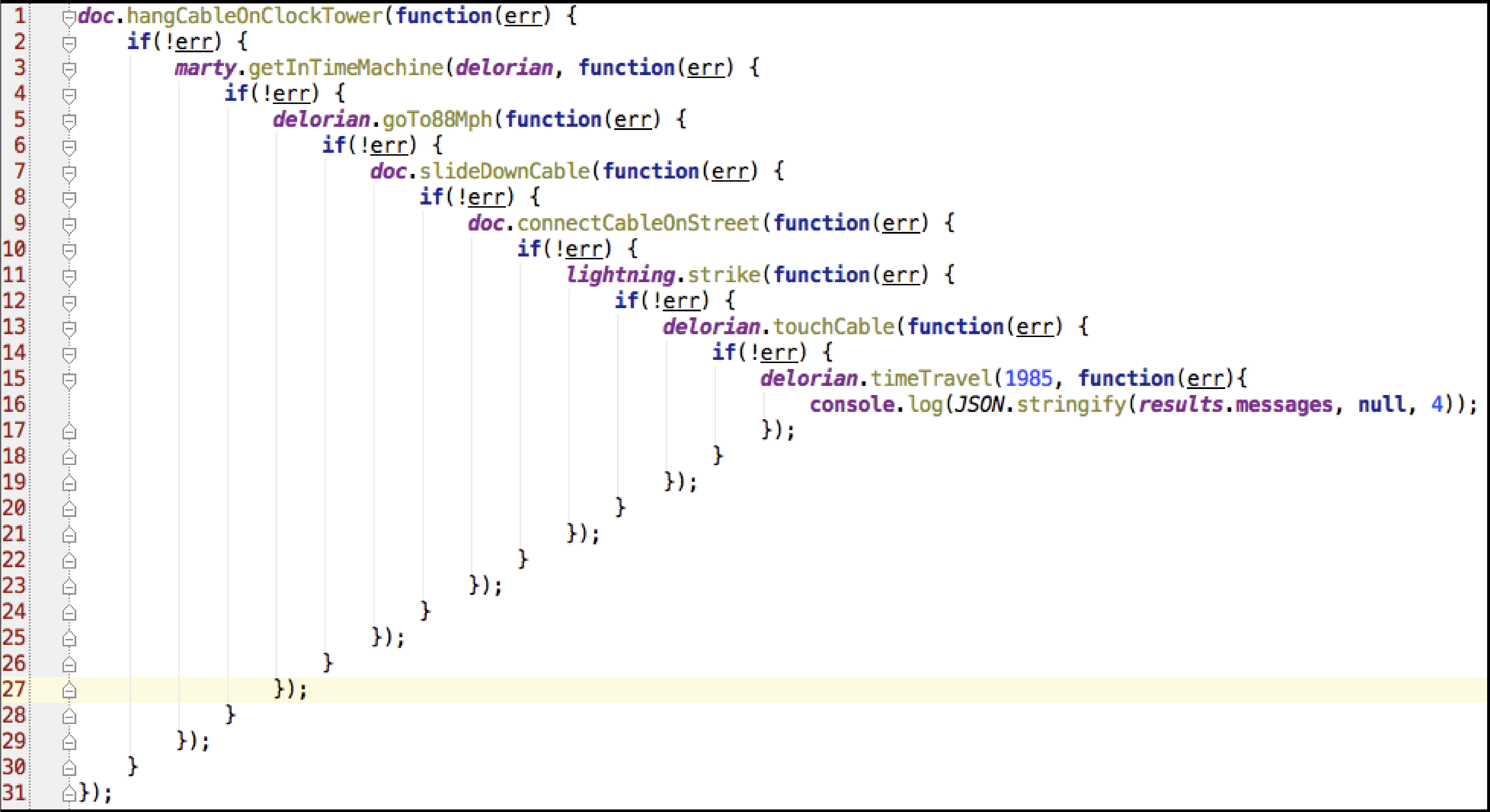
```
// We've all seen something like this, amirite?
setTimeout(app.updateAllTheDom, 0);

// common node style -> callback has error as
// the first arg, remaining args are result(s)
fs.readDir("./", function(err, files){
    if(err) {
        console.log("AW SNAP! Things went badly: " + err);
    }
    else {
        console.log("Here are your files: ");
        files.forEach(function(file){
            console.log("\t" + file);
        });
    }
});
```

It’s simple: Pass a function that will be invoked when the work completes. (Could be synchronous or asynchronous.)

Strategy #1 - Plain Callbacks

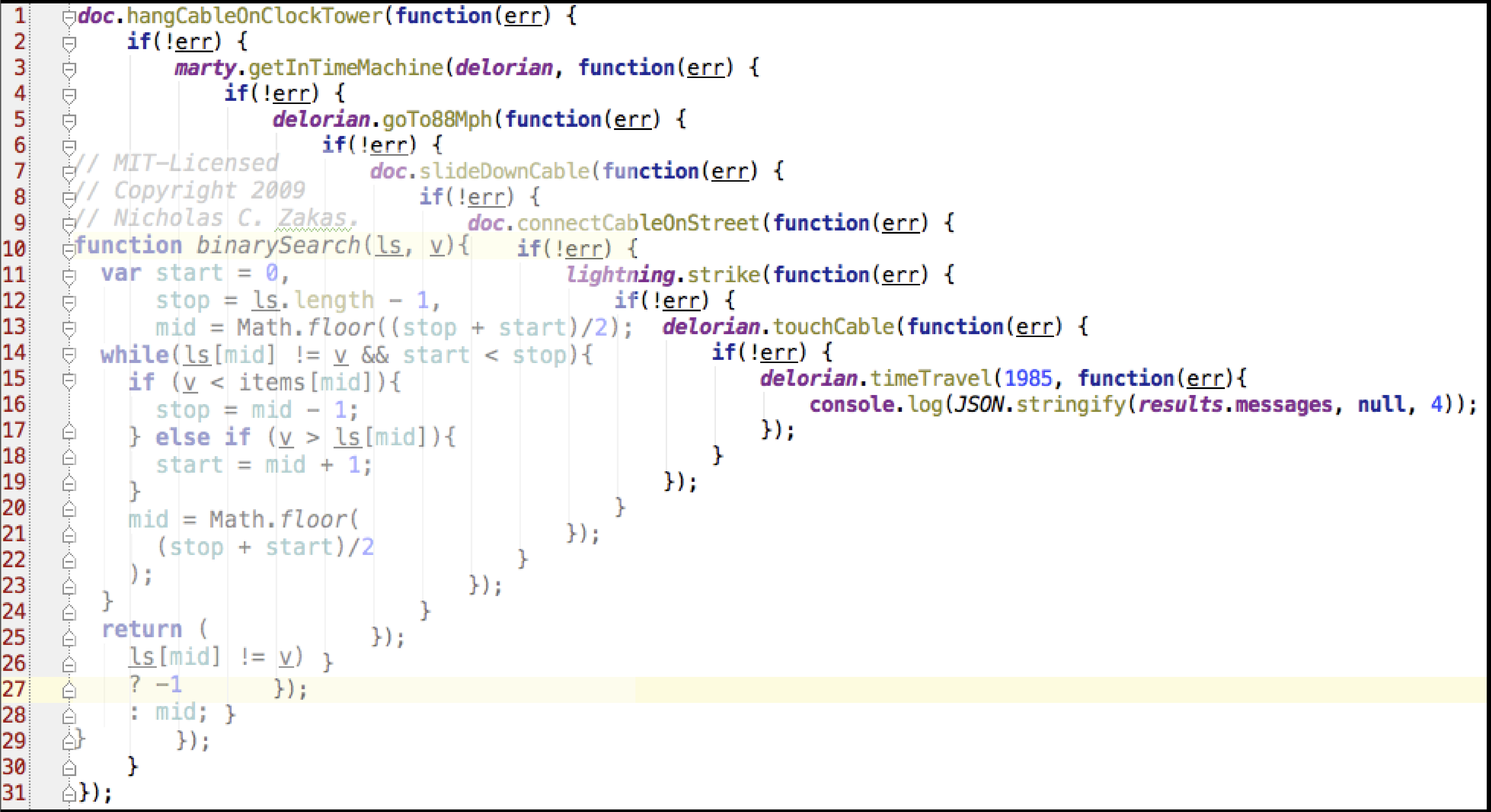
Oh, look! Nested ~~facepalms~~ callbacks.



```
doc.hangCableOnClockTower(function(err) {
    if(!err) {
        marty.getInTimeMachine(delorian, function(err) {
            if(!err) {
                delorian.goTo88Mph(function(err) {
                    if(!err) {
                        doc.slideDownCable(function(err) {
                            if(!err) {
                                doc.connectCableOnStreet(function(err) {
                                    if(!err) {
                                        lightning.strike(function(err) {
                                            if(!err) {
                                                delorian.touchCable(function(err) {
                                                    if(!err) {
                                                        delorian.timeTravel(1985, function(err){
                                                            console.log(JSON.stringify(results.messages, null, 4));
                                                        });
                                                    }
                                                });
                                            }
                                        });
                                    }
                                });
                            }
                        });
                    }
                });
            }
        });
    }
});
```

Strategy #1 - Plain Callbacks

Oh, look! Nested ~~facepalms~~ callbacks.

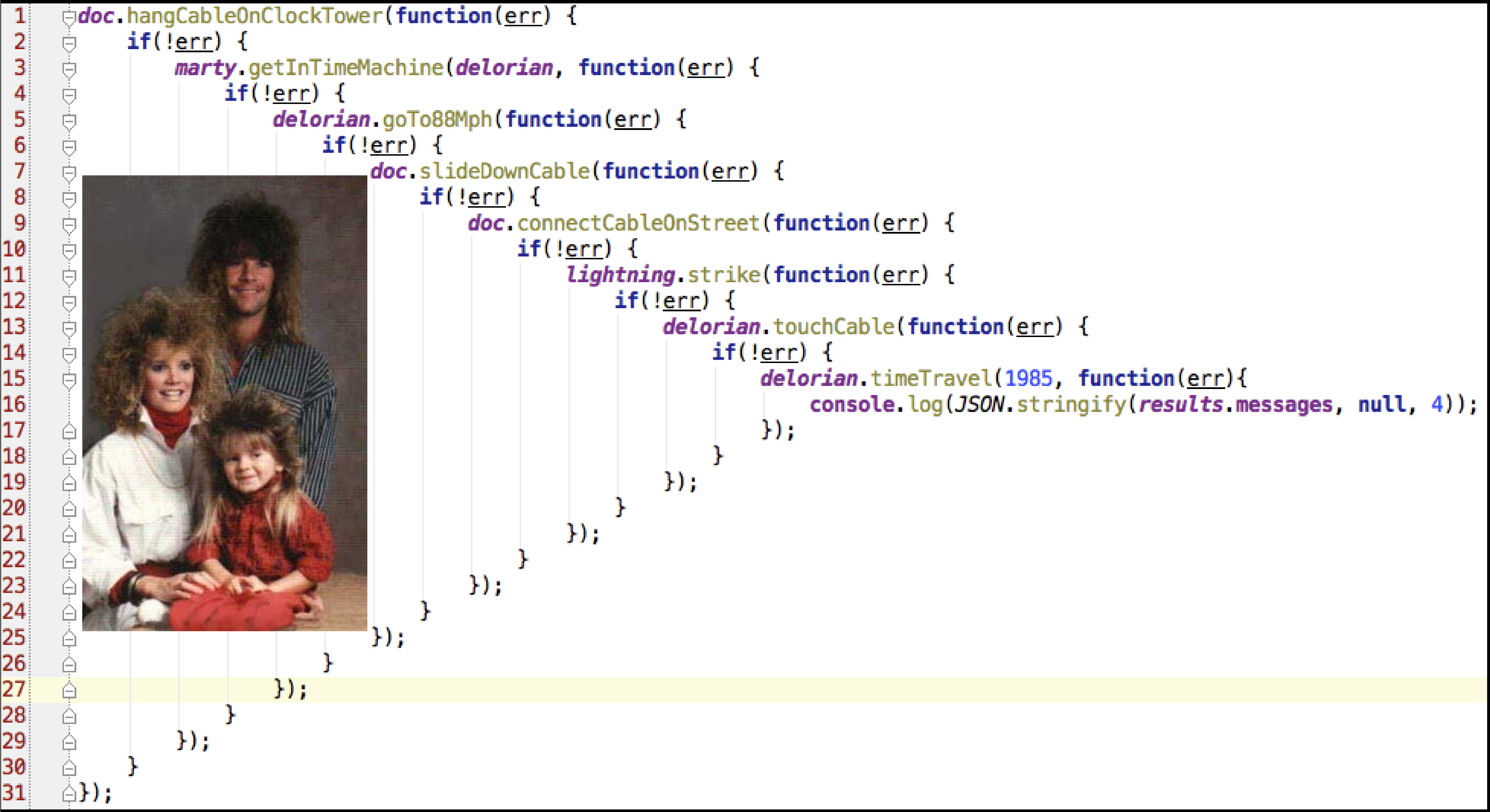


```
doc.hangCableOnClockTower(function(err) {
    if(!err) {
        marty.getInTimeMachine(delorian, function(err) {
            if(!err) {
                delorian.goTo88Mph(function(err) {
                    if(!err) {
                        doc.slideDownCable(function(err) {
                            if(!err) {
                                doc.connectCableOnStreet(function(err) {
                                    if(!err) {
                                        lightning.strike(function(err) {
                                            if(!err) {
                                                delorian.touchCable(function(err) {
                                                    if(!err) {
                                                        delorian.timeTravel(1985, function(err){
                                                            console.log(JSON.stringify(results.messages, null, 4));
                                                        });
                                                    }
                                                });
                                            }
                                        });
                                    }
                                });
                            }
                        });
                    }
                });
            }
        });
    }
});
```

```
// MIT-Licensed
// Copyright 2009
// Nicholas C. Zakas.
function binarySearch(ls, v){
  var start = 0,
      stop = ls.length - 1,
      mid = Math.floor((stop + start)/2);
  while(ls[mid] != v && start < stop){
    if (v < items[mid]){
      stop = mid - 1;
    } else if (v > ls[mid]){
      start = mid + 1;
    }
    mid = Math.floor(
      (stop + start)/2
    );
  }
  return (
    ls[mid] != v)
    ? -1
    : mid;
}
```

Strategy #1 - Plain Callbacks

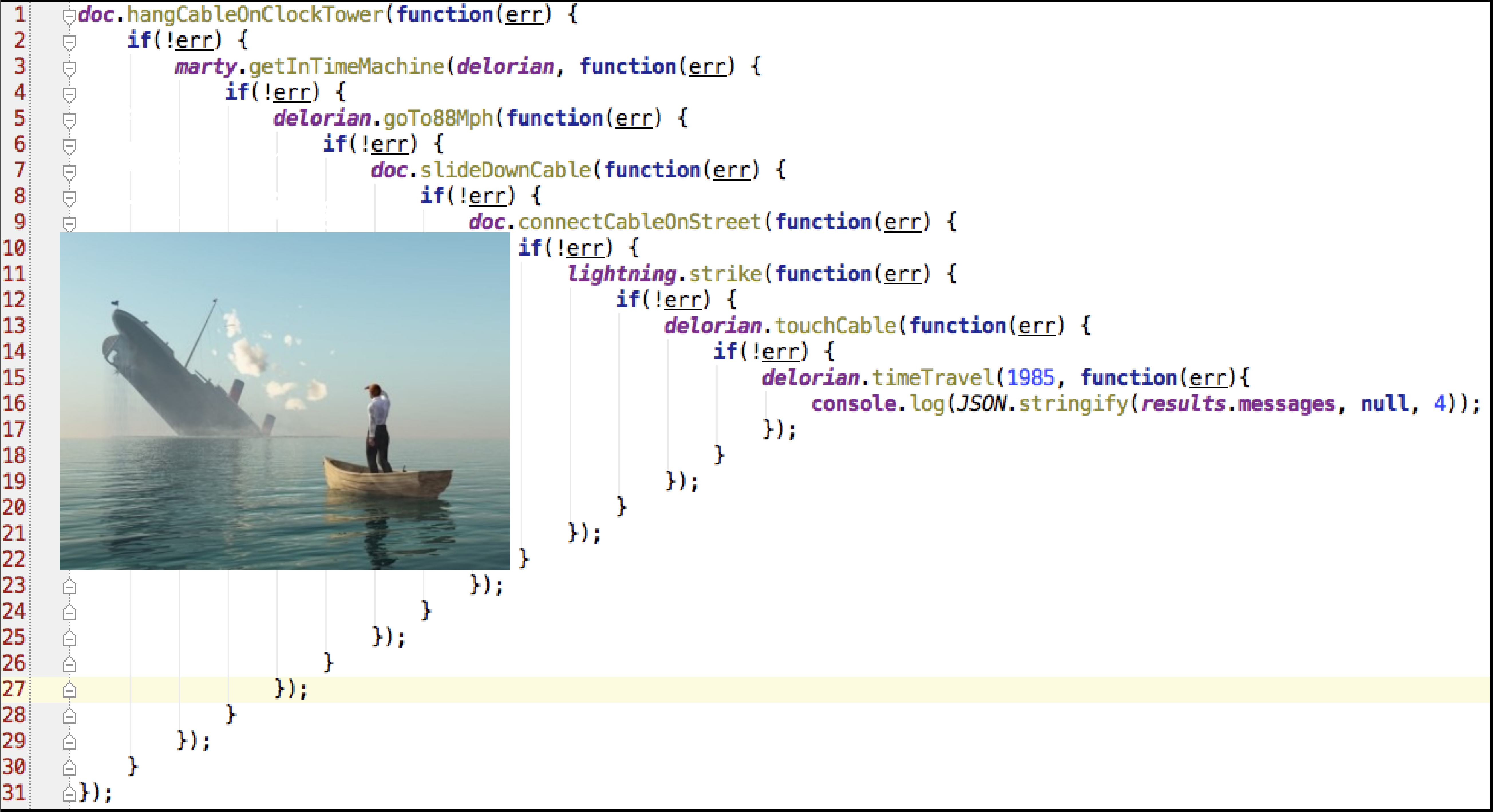
Oh, look! Nested ~~facepalms~~ callbacks.



```
doc.hangCableOnClockTower(function(err) {
    if(!err) {
        marty.getInTimeMachine(delorian, function(err) {
            if(!err) {
                delorian.goTo88Mph(function(err) {
                    if(!err) {
                        doc.slideDownCable(function(err) {
                            if(!err) {
                                doc.connectCableOnStreet(function(err) {
                                    if(!err) {
                                        lightning.strike(function(err) {
                                            if(!err) {
                                                delorian.touchCable(function(err) {
                                                    if(!err) {
                                                        delorian.timeTravel(1985, function(err){
                                                            console.log(JSON.stringify(results.messages, null, 4));
                                                        });
                                                    }
                                                });
                                            }
                                        });
                                    }
                                });
                            }
                        });
                    }
                });
            }
        });
    }
});
```

Strategy #1 - Plain Callbacks

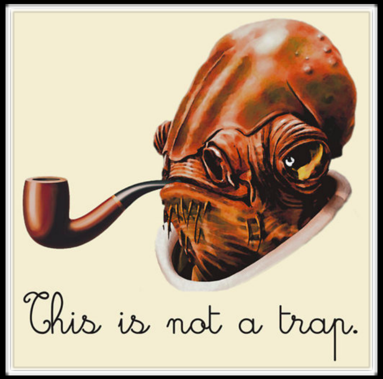
Oh, look! Nested ~~facepalms~~ callbacks.



```
doc.hangCableOnClockTower(function(err) {
    if(!err) {
        marty.getInTimeMachine(delorian, function(err) {
            if(!err) {
                delorian.goTo88Mph(function(err) {
                    if(!err) {
                        doc.slideDownCable(function(err) {
                            if(!err) {
                                doc.connectCableOnStreet(function(err) {
                                    if(!err) {
                                        lightning.strike(function(err) {
                                            if(!err) {
                                                delorian.touchCable(function(err) {
                                                    if(!err) {
                                                        delorian.timeTravel(1985, function(err){
                                                            console.log(JSON.stringify(results.messages, null, 4));
                                                        });
                                                    }
                                                });
                                            }
                                        });
                                    }
                                });
                            }
                        });
                    }
                });
            }
        });
    }
});
```

{ Code }

Strategy #1 - Plain Callbacks



PROS:

- Simple
- No extra libs required
- works well for stand-alone concerns

CONS:

- hardens tight coupling
- limited 'visibility' into the operation
- gets complex when nested

Strategy #1 - Plain Callbacks

Recommendations:

- Use for concerns that go 1 or 2 levels deep (at most)
- Use in ‘public API’ (less opinionated that other options)*

Strategy #2 - Events

- EventEmitter (or similar style API):
  - on(“SomeEvent”, callback [, context])
  - off(“SomeEvent” [, callback [, context]])
  - emit(“SomeEvent”, [arg1, arg2, etc...])
- Break components into small pieces that listen for events to occur at any time

{ Code }

Strategy #2 - Events

PROS:

- Better decoupling
- More testable than nested callbacks
- Better at coordinating evented workflow

CONS:

- Despite decoupling, observers still require direct reference to observed

Strategy #2 - Events

PROS:

- Better decoupling
- More testable than nested callbacks
- Better at coordinating evented workflow

CONS:

- Despite decoupling, observers still require direct reference to observed

Libraries to check out:

- jQuery (custom events)
- EventEmitter (node & browser)
- EventEmitter2
- Backbone.Events

Strategy #2 - Events

Recommendations:

- Use where nesting would exceed 2 levels
- Emitting is (usually) superior to continuations. Use in place of plain callbacks where possible
- Use between separate components/

Strategy #3 - Deferreds

- What is a ‘deferred’?
  - “a chainable utility object that can register multiple callbacks into callback queues, invoke callback queues, and relay the success or failure state of any synchronous or asynchronous function” (jQuery API docs: http://api.jquery.com/category/deferred-object/)
- register one or more callbacks with
  - done() (invoked when ‘resolved’)
  - fail() (invoked when ‘rejected’)
  - always() (invoked, er, um...*always*)

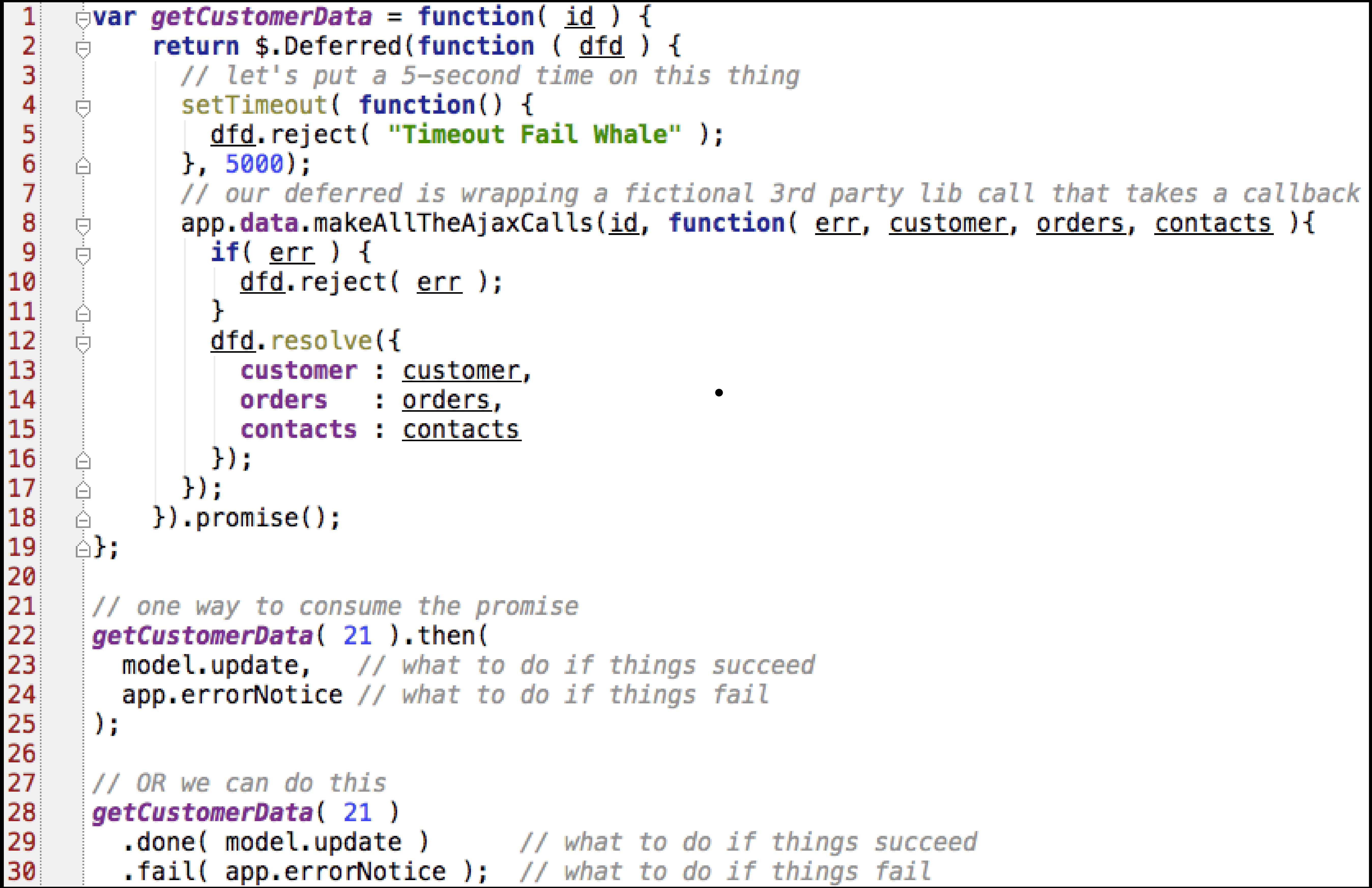
Strategy #3 - Deferreds

- Code that created/owns the deferred calls resolve() or reject()
- callbacks registered after a deferred has resolved/rejected get immediately invoked
- deferreds can be chained via pipe()
- can send progress notifications

Strategy #3 - Deferreds

- But is it a ‘deferred’ or a ‘promise’?
- In jQuery, a deferred can return a promise
- promises:
  - allow callbacks to be registered
  - allow state to only be *examined*
  - do <u>not</u> provide ability to resolve/reject, notify, etc. (no state mutation)

Strategy #3 - Deferreds

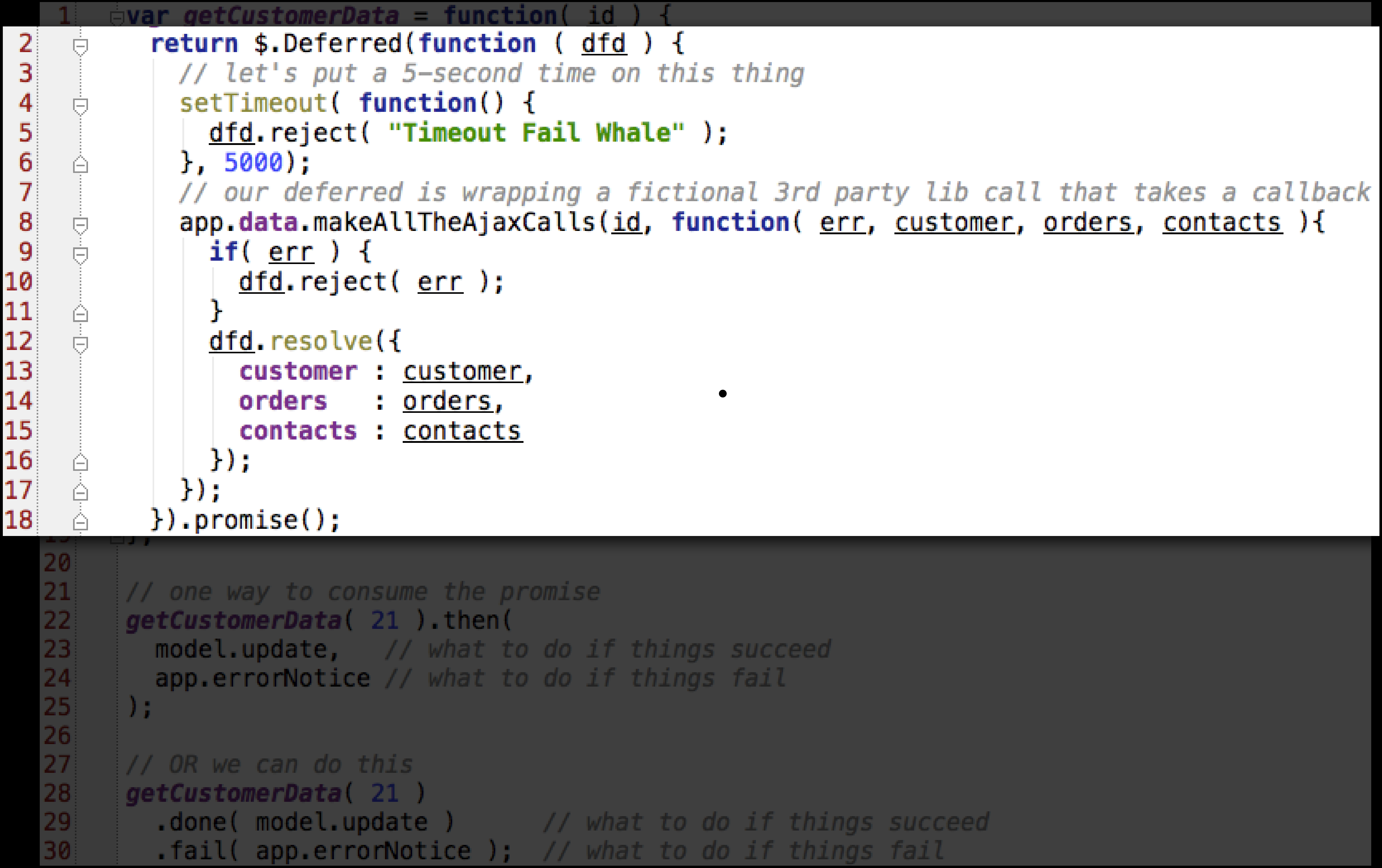


```
var getCustomerData = function( id ) {
    return $.Deferred(function ( dfd ) {
      // let's put a 5-second time on this thing
      setTimeout( function() {
        dfd.reject( "Timeout Fail Whale" );
      }, 5000);
      // our deferred is wrapping a fictional 3rd party lib call that takes a callback
      app.data.makeAllTheAjaxCalls(id, function( err, customer, orders, contacts ){
        if( err ) {
          dfd.reject( err );
        }
        dfd.resolve({
          customer : customer,
          orders   : orders,
          contacts : contacts
        });
      });
    }).promise();
};

// one way to consume the promise
getCustomerData( 21 ).then(
  model.update,   // what to do if things succeed
  app.errorNotice // what to do if things fail
);

// OR we can do this
getCustomerData( 21 )
  .done( model.update )      // what to do if things succeed
  .fail( app.errorNotice );  // what to do if things fail
```

Strategy #3 - Deferreds

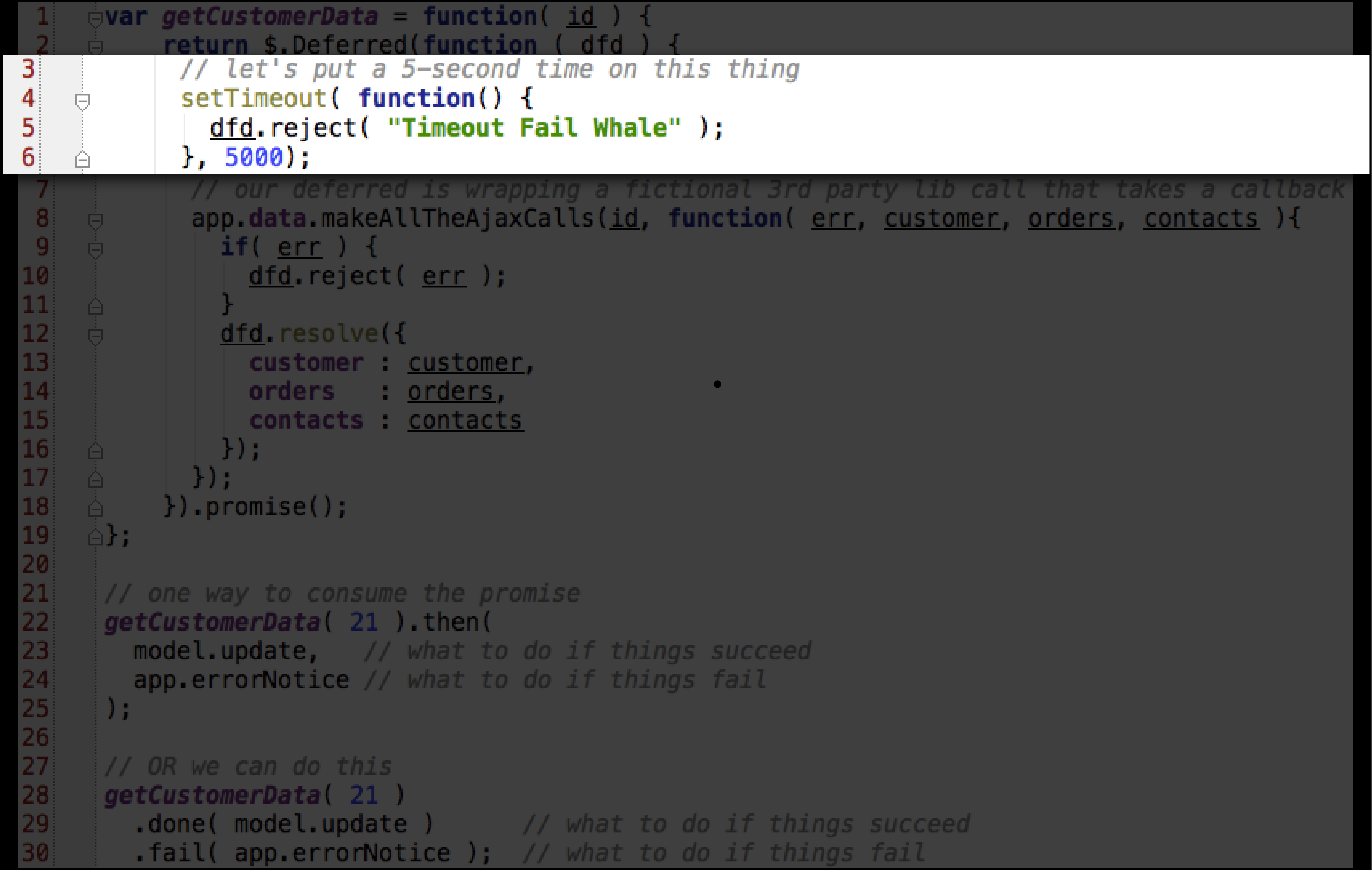


```
var getCustomerData = function( id ) {
  return $.Deferred(function ( dfd ) {
    // let's put a 5-second time on this thing
    setTimeout( function() {
      dfd.reject( "Timeout Fail Whale" );
    }, 5000);
    // our deferred is wrapping a fictional 3rd party lib call that takes a callback
    app.data.makeAllTheAjaxCalls(id, function( err, customer, orders, contacts ){
      if( err ) {
        dfd.reject( err );
      }
      dfd.resolve({
        customer : customer,
        orders   : orders,
        contacts : contacts
      });
    });
  }).promise();
});

// one way to consume the promise
getCustomerData( 21 ).then(
  model.update,   // what to do if things succeed
  app.errorNotice // what to do if things fail
);

// OR we can do this
getCustomerData( 21 )
  .done( model.update )      // what to do if things succeed
  .fail( app.errorNotice );  // what to do if things fail
```

Strategy #3 - Deferreds

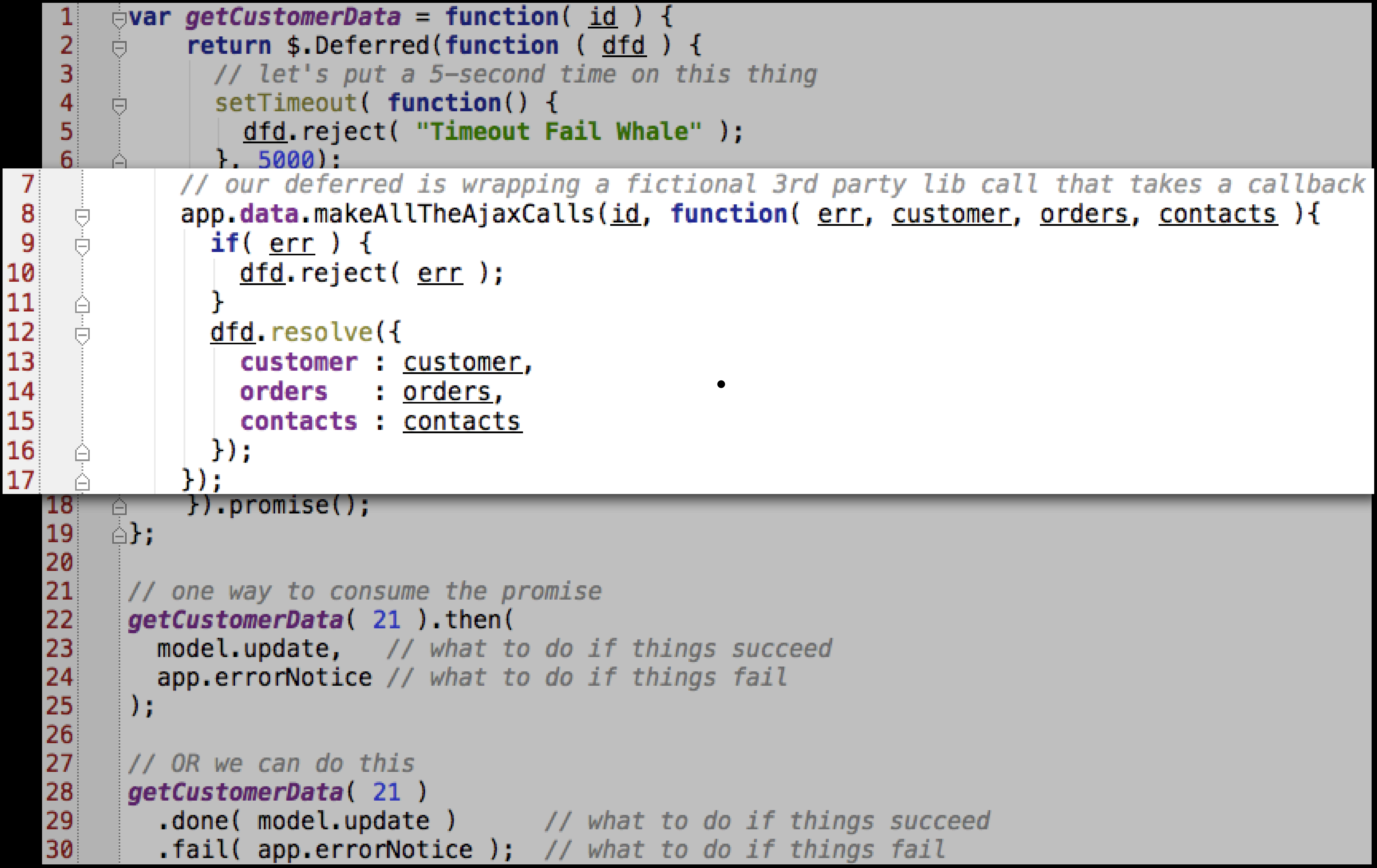


```
var getCustomerData = function( id ) {
    return $.Deferred(function ( dfd ) {
      // let's put a 5-second time on this thing
      setTimeout( function() {
        dfd.reject( "Timeout Fail Whale" );
      }, 5000);
      // our deferred is wrapping a fictional 3rd party lib call that takes a callback
      app.data.makeAllTheAjaxCalls(id, function( err, customer, orders, contacts ){
        if( err ) {
          dfd.reject( err );
        }
        dfd.resolve({
          customer : customer,
          orders   : orders,
          contacts : contacts
        });
      });
    }).promise();
};

// one way to consume the promise
getCustomerData( 21 ).then(
  model.update,   // what to do if things succeed
  app.errorNotice // what to do if things fail
);

// OR we can do this
getCustomerData( 21 )
  .done( model.update )       // what to do if things succeed
  .fail( app.errorNotice );  // what to do if things fail
```

Strategy #3 - Deferreds

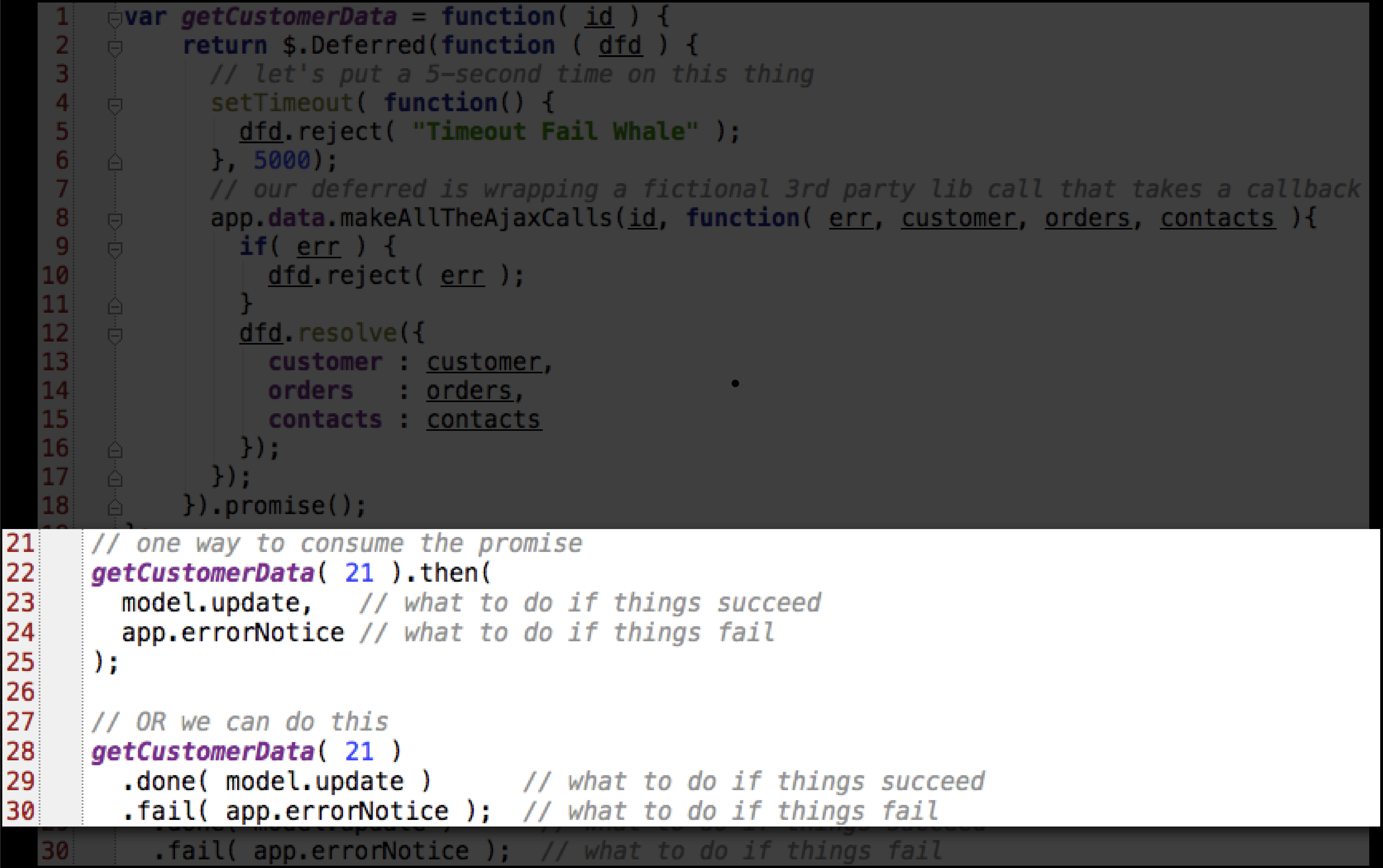


```
var getCustomerData = function( id ) {
    return $.Deferred(function ( dfd ) {
      // let's put a 5-second time on this thing
      setTimeout( function() {
        dfd.reject( "Timeout Fail Whale" );
      }, 5000);
      // our deferred is wrapping a fictional 3rd party lib call that takes a callback
      app.data.makeAllTheAjaxCalls(id, function( err, customer, orders, contacts ){
        if( err ) {
          dfd.reject( err );
        }
        dfd.resolve({
          customer : customer,
          orders   : orders,
          contacts : contacts
        });
      });
    }).promise();
};

// one way to consume the promise
getCustomerData( 21 ).then(
  model.update,   // what to do if things succeed
  app.errorNotice // what to do if things fail
);

// OR we can do this
getCustomerData( 21 )
  .done( model.update )      // what to do if things succeed
  .fail( app.errorNotice );  // what to do if things fail
```

Strategy #3 - Deferreds



```
var getCustomerData = function( id ) {
    return $.Deferred(function ( dfd ) {
      // let's put a 5-second time on this thing
      setTimeout( function() {
        dfd.reject( "Timeout Fail Whale" );
      }, 5000);
      // our deferred is wrapping a fictional 3rd party lib call that takes a callback
      app.data.makeAllTheAjaxCalls(id, function( err, customer, orders, contacts ){
        if( err ) {
          dfd.reject( err );
        }
        dfd.resolve({
          customer : customer,
          orders   : orders,
          contacts : contacts
        });
      });
    }).promise();
```

```
// one way to consume the promise
getCustomerData( 21 ).then(
  model.update,   // what to do if things succeed
  app.errorNotice // what to do if things fail
);

// OR we can do this
getCustomerData( 21 )
  .done( model.update )      // what to do if things succeed
  .fail( app.errorNotice );  // what to do if things fail
```

.fail(app.errorNotice); // what to do if things fail

{ Code }

Strategy #3 - Deferreds

PROS:

- Can flatten ‘nested callback hell’
- Results can be cached*
- Great for aggregating results of multiple *related* async functions

CONS:

- Returning promises on a public API is a *highly* opinionated constraint on developers
- Deferreds often trash the narrative of the code
- Can be *very* difficult to test/debug

Strategy #3 - Deferreds

PROS:

- Can flatten ‘nested callback hell’
- Results can be cached*
- Great for aggregating results of multiple *related* async functions

CONS:

- Returning promises on a public API is a *highly* opinionated constraint on developers
- Deferreds often trash the narrative of the code
- Can be *very* difficult to test/debug

Libraries to check out:

- jQuery 1.5 or greater
- async.js https://github.com/fjakobs/async.js
- Q - https://github.com/kriskowal/q/

Strategy #3 - Deferreds

Recommendations:

- Use for aggregating results of async calls that should always resolve together
- Use when the 3rd party lib author left you no choice but to use their promises

Strategy #4 - Message Bus

- Similar to custom events, but no direct reference to observed subject
- Great option to adapt existing APIs, extending the reach of their events/messages
- “Several small apps” that communicate via message passing

Strategy #4 - Message Bus

- The "bus" is the only common reference
- Typical API includes:
  - subscribe
  - unsubscribe
  - publish
- An 'envelope' is published (unlike event emitting's 0-n args)

{ Code }

Strategy #4 - Message Bus

PROS:

- Clean SoC
- Very testable
- Very extendable

CONS:

- Prone to “boilerplate proliferation”
- Can be difficult to follow

Strategy #4 - Message Bus

PROS:

- Clean SoC
- Very testable
- Very extendable

CONS:

- Prone to “boilerplate proliferation”
- Can be difficult to follow

Libraries to check out:

- postal.js (shameless plug!)
- amplify.js

Strategy #4 - Message Bus

Recommendations:

- Use between modules (wrap existing APIs with message endpoints)
- Use between components that do not (or should not) need a direct reference to each other, but might be interested in data published

Strategy #5 - Finite State Machine

- Exists in one of a finite number of states.
- Responds to input based on the current state.
- Can transition to a different state under defined condition(s)

Strategy #5 - Finite State Machine

100k-foot-view Concepts

- **States** - define states in which machine can exist *(states affect how a machine responds to input/events)*
- **Transitions** - moving from one state to another
- **Input/Events** - behavior (internal or external) that can produce output and/or cause state transitions
- **Rules/Constraints** - used to determine if the machine can transition to new state

Strategy #5 - Finite State Machine

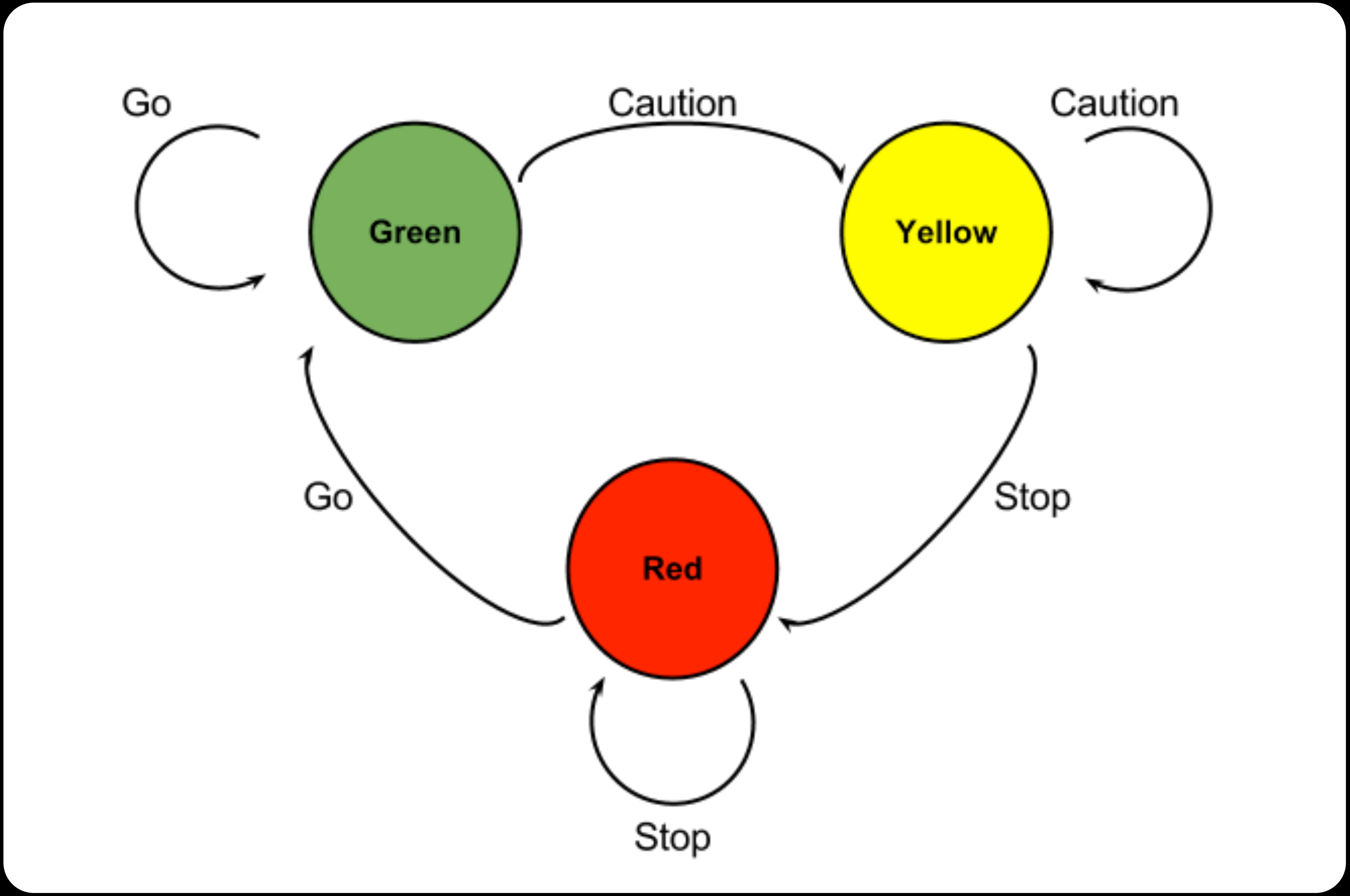
WARNING: FSM Minutia Ahead

Strategy #5 - Finite State Machine

- General Types of FSMs:
  - Acceptor
  - Transducer
    - **Moore machine** - output depends on state (entry actions)
    - **Mealy machine** - output depends on state ***and*** input
- **Deterministic** - only one transition possible for each state
- **Non-deterministic** - zero or more transitions possible from each state

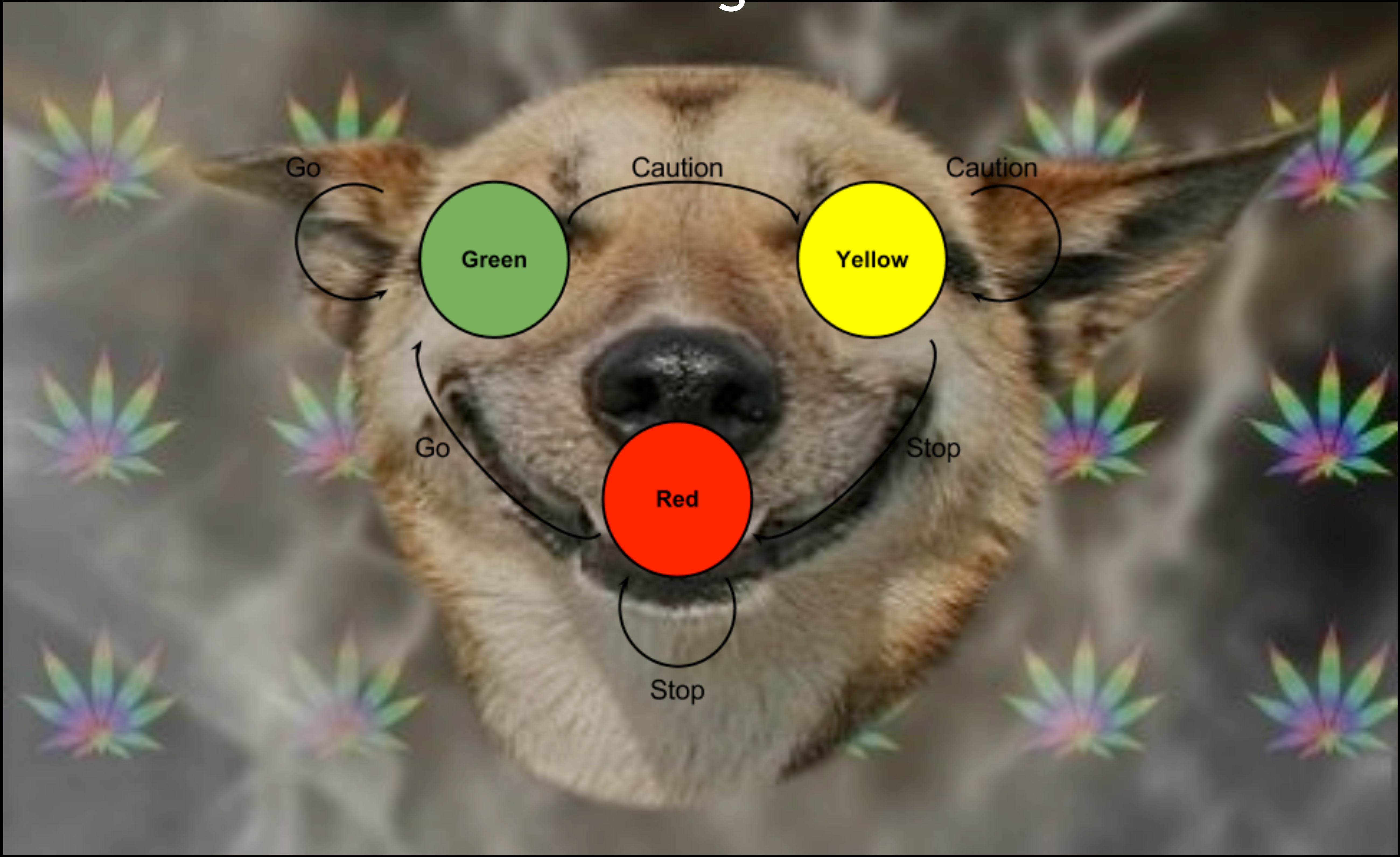
Strategy #5 - Finite State Machine

Traffic Light FSM



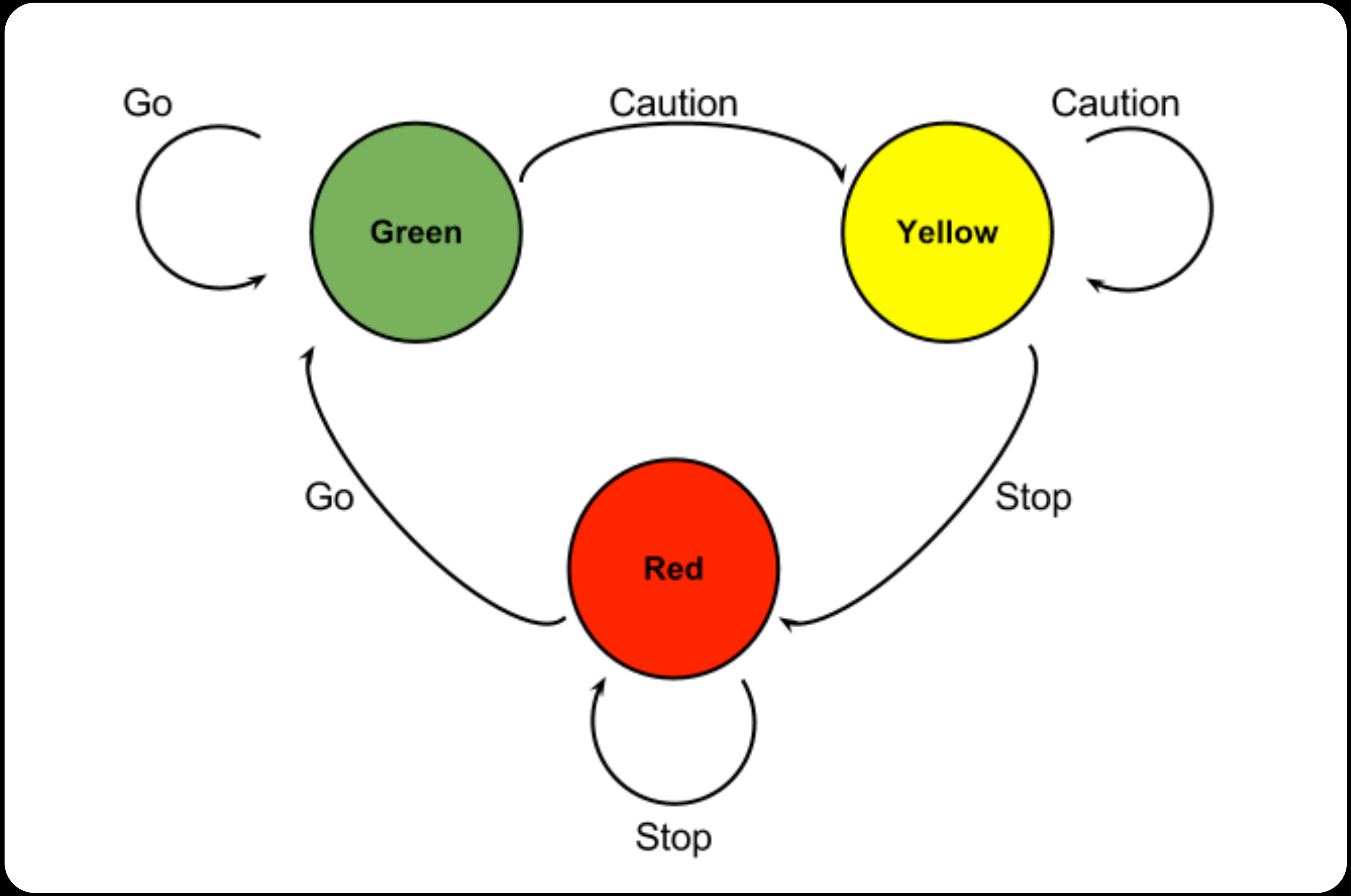
Strategy #5 - Finite State Machine

Traffic Light FSM



Strategy #5 - Finite State Machine

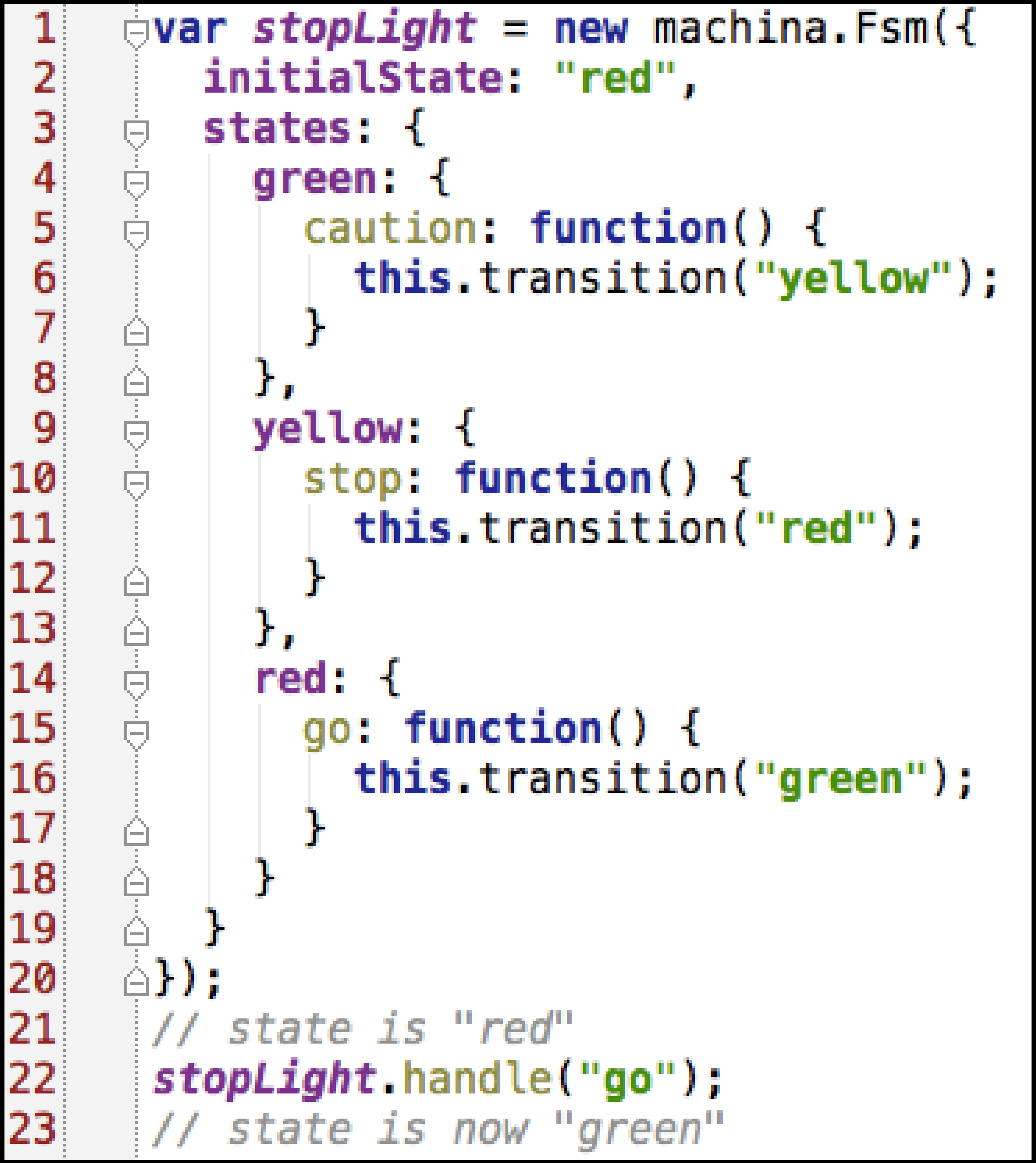
Traffic Light FSM



Strategy #5 - Finite State Machine

- machina.js - helper lib for FSMs in JavaScript
- Using machina.js to drive FSMs:
  - Control is *yours* (low level)
  - Acceptor...Transducer...Franken-FSM
  - Leans towards Mealy, but supports Moore or *both*
  - You determine determinism
    (preferably with determination....*yo, dawg, I hear you like determinism...*)

Strategy #5 - Finite State Machine



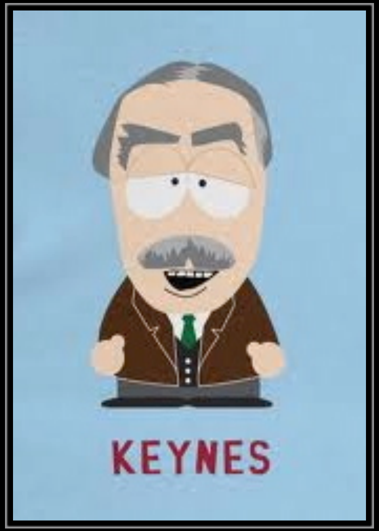
```
var stopLight = new machina.Fsm({
    initialState: "red",
    states: {
        green: {
            caution: function() {
                this.transition("yellow");
            }
        },
        yellow: {
            stop: function() {
                this.transition("red");
            }
        },
        red: {
            go: function() {
                this.transition("green");
            }
        }
    }
});
// state is "red"
stopLight.handle("go");
// state is now "green"
```

{ Code }

Strategy #5 - Finite State Machine

PROS:

- Very useful for coordinating long-running async workflows
- Expressive intent
- Extremely versatile



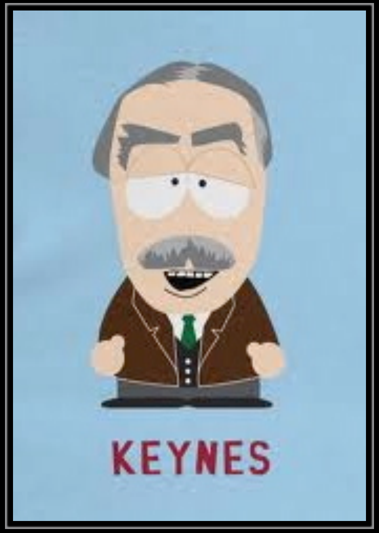
CONS:

- Poorly abstracted FSMs can lead to ‘state handler explosion’ when adding new states/input
- Can involve more lines of code*

Strategy #5 - Finite State Machine

PROS:

- Very useful for coordinating long-running async workflows
- Expressive intent
- Extremely versatile



CONS:

- Poorly abstracted FSMs can lead to ‘state handler explosion’ when adding new states/input
- Can involve more lines of code*

Libraries to check out:

- machina.js (shameless plug!)
- state.js - https://github.com/nickfargo/state

Strategy #5 - Finite State Machine

Recommendations:

- look for workflow applications!
- deterministic FSM can help with initialization
- consider an FSM for managing offline/online concerns
- consider an FSM to abstract “enabled/disabled” type concerns

Be kind to your API consumers

- Avoid deeply nested callbacks
- Don’t let your abstractions leak
  - Beware of what you bake into your API
  - Avoid forcing dependencies where possible
  - Consider offering plain callback alternatives alongside more opinionated API approaches

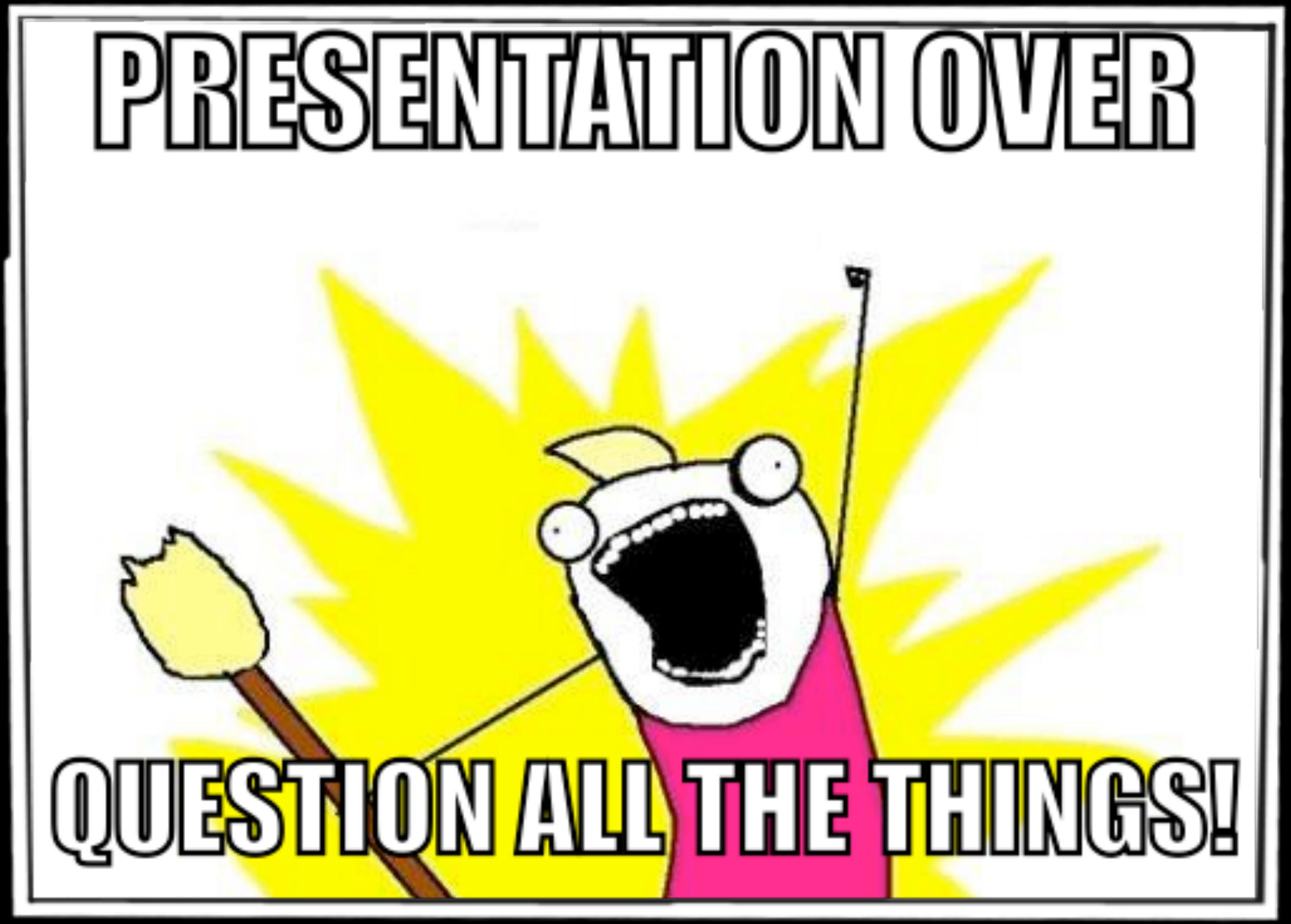
Further Reading

- Async JavaScript
  by Trevor Burnham

  fantastic treatment of jQuery deferreds + async.js

- Finite State Machines:
  - http://blog.markwshead.com/869/state-machines-computer-science/
  - http://machina-js.org/ (shameless plug!)
  - http://www.ibm.com/developerworks/library/wa-finitemach1/
    - (Great further reading suggestions on this one!)
- Other good stuff:
  - http://www.2ality.com/2012/06/continuation-passing-style.html
  - http://www.erichynds.com/jquery/using-deferreds-in-jquery/

Code/Slides for this presentation - http://bit.ly/async-js-patterns



Q & A