Async Strategies & Patterns in Javascript

{ Jim Cowart }

devLINK Technical Conference
August 29 - 31, 2012 • www.devlink.net

Who am I?

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

WARNING - Opinions
(Tempered by Experience)

Opinions on asynchronous ‘styles’ **abound**
And as we all know....opinions are like...

WARNING - Opinions
(Tempered by Experience)

Opinions on asynchronous ‘styles’ **abound**
And as we all know....opinions are like...

** Belly Buttons ** (We all have them)

Well, except for that creepy-looking guy on “Kyle XY”

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

```
1   $.ajax({
2       type : "POST",
3       url  : "/customer/123"
4   }).done( function ( msg ) {
5       bus.customer.publish({
6           topic : "customer.read",
7           data  : msg
8       });
9   });
10
11  // Ah, nothing like real world use cases
12  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

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

Strategy #1 - Plain Callbacks

```
1   // We've all seen something like this, amirite?
2   setTimeout(app.updateAllTheDom, 0);
3
4   // common node style -> callback has error as
5   // the first arg, remaining args are result(s)
6   fs.readDir("./", function(err, files){
7       if(err) {
8           console.log("AW SNAP! Things went badly: " + err);
9       }
10      else {
11          console.log("Here are your files: ");
12          files.forEach(function(file){
13              console.log("\t" + file);
14          });
15      }
16  });
```

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.

```
1   doc.hangCableOnClockTower(function(err) {
2       if(!err) {
3           marty.getInTimeMachine(delorian, function(err) {
4               if(!err) {
5                   delorian.goTo88Mph(function(err) {
6                       if(!err) {
7                           doc.slideDownCable(function(err) {
8                               if(!err) {
9                                   doc.connectCableOnStreet(function(err) {
10                                      if(!err) {
11                                          lightning.strike(function(err) {
12                                              if(!err) {
13                                                  delorian.touchCable(function(err) {
14                                                      if(!err) {
15                                                          delorian.timeTravel(1985, function(err){
16                                                              console.log(JSON.stringify(results.messages, null, 4));
17                                                          });
18                                                      }
19                                                  });
20                                              }
21                                          });
22                                      }
23                                  });
24                              }
25                          });
26                      }
27                  });
28              }
29          });
30      }
31  });
```

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) {
// MIT-Licensed                doc.slideDownCable(function(err) {
// Copyright 2009                  if(!err) {
// Nicholas C. Zakas,                  doc.connectCableOnStreet(function(err) {
function binarySearch(ls, v){           if(!err) {
  var start = 0,                            lightning.strike(function(err) {
      stop = ls.length - 1,                     if(!err) {
      mid = Math.floor((stop + start)/2);   delorian.touchCable(function(err) {
  while(ls[mid] != v && start < stop){          if(!err) {
    if (v < items[mid]){                            delorian.timeTravel(1985, function(err){
      stop = mid - 1;                                   console.log(JSON.stringify(results.messages, null, 4));
    } 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));
                                                        });
                                                    }
                                                });
                                            }
                                        });
                                    }
                                });
                            }
                        });
                    }
                });
            }
        });
    }
});
```

the
js developer
bermuda triangle

{ Code }

Strategy #1 - Plain Callbacks

PROS:

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

CONS:

- limited ‘visibility’ into the operation
- no built-in way to cancel/ abort
- gets messy when nested (pyramid of doom)

Strategy #1 - Plain Callbacks

Recommendations:

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

Strategy #2 - observing via Events

- EventEmitter (or similar style API):
  - on/once
  - off
  - emit
- Break components into small pieces that listen for events to occur at any time

{ Code }

Strategy #2 - observing via Events

PROS:

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

CONS:

- Can get verbose
- Despite decoupling, observers still require direct reference to observed

Libraries to check out:

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

Strategy #2 - observing via Events

Recommendations:

- Use where nesting would exceed 2 levels
- Use between separate components/ objects inside modules

Strategy #3 - Promises/ 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 - Promises/ 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 - Promises/ 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 not provide ability to resolve/ reject, notify, etc. (no state mutation)

Strategy #3 - Promises/ Deferreds

```
1   var getCustomerData = function ( id ) {
2       return $.Deferred(function ( dfd ) {
3           // let's put a 5-second timeout on this thing
4           setTimeout( function () {
5               dfd.reject( "Timeout Fail Whale" );
6           }, 5000 );
7           // our deferred is wrapping a fictional 3rd party call that takes a callback
8           app.data.makeAllTheAjaxCalls( id, function ( err, customer, orders, contacts ) {
9               if ( err ) {
10                  dfd.reject( err );
11              }
12              dfd.resolve( {
13                  customer : customer,
14                  orders : orders,
15                  contacts : contacts
16              } );
17          } );
18      } ).promise();
19  }
20
21  $.when( getCustomerData() ).then(
22      model.update,   // what to do if things succeed
23      app.errorNotice // what to do if things fail
24  );
```

Strategy #3 - Promises/ Deferreds

```
var getCustomerData = function ( id ) {
    return $.Deferred(function ( dfd ) {
        // let's put a 5-second timeout on this thing
        setTimeout( function () {
            dfd.reject( "Timeout Fail Whale" );
        }, 5000 );
        // our deferred is wrapping a fictional 3rd party 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();
}

$.when( getCustomerData() ).then(
    model.update,   // what to do if things succeed
    app.errorNotice // what to do if things fail
);
```

Strategy #3 - Promises/ Deferreds

```
1   var getCustomerData = function ( id ) {
2       return $.Deferred(function ( dfd ) {
3           // let's put a 5-second timeout on this thing
4           setTimeout( function () {
5               dfd.reject( "Timeout Fail Whale" );
6           }, 5000 );
7           // $.Deferred is wrapping a fire-and-forget jQuery call that takes a callback...
8           app.data.makeAllTheAjaxCalls( id, function ( err, customer, orders, contacts ) {
9               if ( err ) {
10                  dfd.reject( err );
11              }
12              dfd.resolve( {
13                  customer : customer,
14                  orders : orders,
15                  contacts : contacts
16              } );
17          } );
18      } ).promise();
19  }
20
21  $.when( getCustomerData() ).then(
22      model.update,   // what to do if things succeed
23      app.errorNotice // what to do if things fail
24  );
```

Strategy #3 - Promises/ Deferreds

```
1   var getCustomerData = function ( id ) {
2       return $.Deferred(function ( dfd ) {
3           // let's put a 5-second timeout on this thing
4           setTimeout( function () {
5               dfd.reject( "Timeout Fail Whale" );
6           }, 5000 );
7           // our deferred is wrapping a fictional 3rd party call that takes a callback
8           app.data.makeAllTheAjaxCalls( id, function ( err, customer, orders, contacts ) {
9               if ( err ) {
10                  dfd.reject( err );
11              }
12              dfd.resolve( {
13                  customer : customer,
14                  orders : orders,
15                  contacts : contacts
16              } );
17          } );
18      } ).promise();
19  }
20
21  $.when( getCustomerData() ).then(
22      model.update,   // what to do if things succeed
23      app.errorNotice // what to do if things fail
24  );
```

Strategy #3 - Promises/ Deferreds

```
var getCustomerData = function ( id ) {
    return $.Deferred(function ( dfd ) {
        // let's put a 5-second timeout on this thing
        setTimeout( function () {
            dfd.reject( "Timeout Fail Whale" );
        }, 5000 );
        // our deferred is wrapping a fictional 3rd party 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();
}

$.when( getCustomerData() ).then(
    model.update,   // what to do if things succeed
    app.errorNotice // what to do if things fail
);
```

{ Code }

Strategy #3 - Promises/ Deferreds

PROS:

- Can make ‘nested callback hell’ significantly more readable
- Results can be cached
- Great for aggregating results of multiple *related* async functions
- Rich API functionality

CONS:

- Returning promises on a public API is a *highly* opinionated constraint on developers
- Results can be cached (yes, this can be a source of facepalms)
- 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 - Promises/ Deferreds

Recommendations:

- Use for aggregating results of async calls that should always resolve together
- Use in place of plain callbacks where progress notifications are needed*
- Use when the 3rd party lib author left you no choice but to use their promises

** assuming you can't use message passing/event delegation instead*

Strategy #4 - Using a local message bus

- Similar to custom events, but no direct reference to observed subject
- best used to adapt existing APIs (i.e. - decoupled alternative to wrapping with deferreds, etc.)
- “Several small apps” that communicate via message passing

{ Code }

Strategy #4 - Using a local 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 - Using a local 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 machine respond 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
    - **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

Strategy #5 - Finite State Machine

Traffic Light FSM

Strategy #5 - Finite State Machine

- 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 require decent amount of boilerplate 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-patterns

Q & A