Messaging (& Eventing) Patterns in JavaScript

{ Jim Cowart }

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

Wait -Who is this guy?

* 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

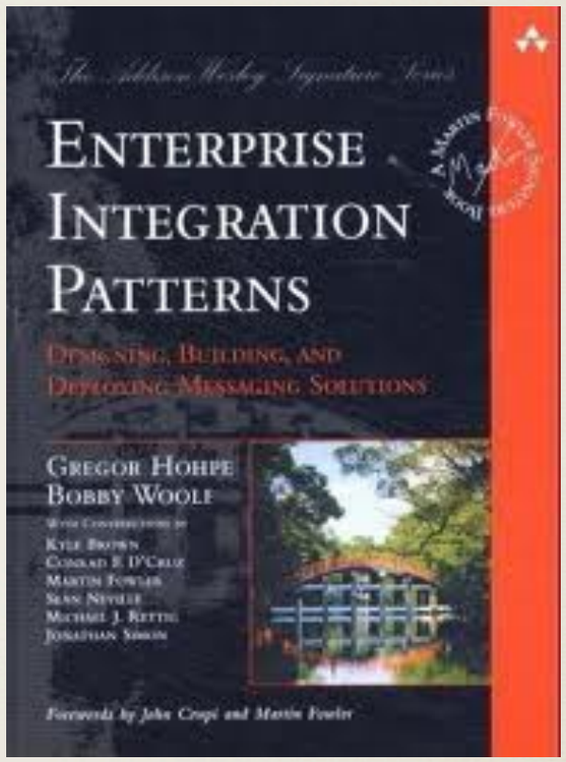
Why are we here?

* Discuss how applying messaging principles in JavaScript:
  * can untangle ‘tightly-coupled-event-handler-spaghetti’ code typical of so many web applications
  * enabled better SoC and testability
  * help write many small applications instead of one large behemoth

Shoulders of Giants

Patterns discussed in this presentation can be found in -

Enterprise Integration Patterns: Designing, Building,and Deploying Messaging Solutions



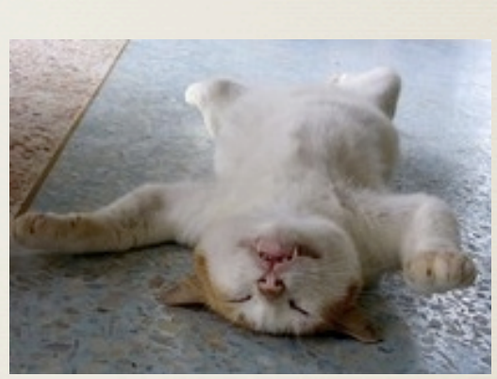
By Gregor Hohpe, Bobby Woolf
Published October 10, 2003

http://www.enterpriseintegrationpatterns.com

Also - big thanks to Kevin Hakanson (@hakanson), for inspiring a lot of the approach in this slide deck.

Arbitrary Distinctions (sort of)

* Many use the term “pub/sub” to describe “event delegation” *and* “messaging” patterns. This kills kittens (& developers’ understanding)



* that’s ok, *technically*...but
  * “event delegation” typically requires direct reference
  * messaging typically involves mediator/broker

so - by ‘eventing’, I mean...

```
// Eventing involving the DOM - we've all seen this
$('#save-btn' ).on("click", function(){
    contactsModel.save();
});

// Eventing inside a Backbone View
var MyCustomView = Backbone.View.extend({
    initialize: function() {
        this.model = new CustomModel();
        // listening for any change on the model
        this.model.on("change", this.render);
        // listening for sync events
        this.model.on("sync", this.updateSaveStatus);
    },
    updateSaveStatus: function() {
        // target specific element for surgical update in DOM
    }
    // more stuff defined here....etc
});
```

so - by ‘eventing’, I mean...

```
// Eventing involving the DOM - we've all seen this
$('#save-btn' ).on("click", function(){
    contactsModel.save();
});
```

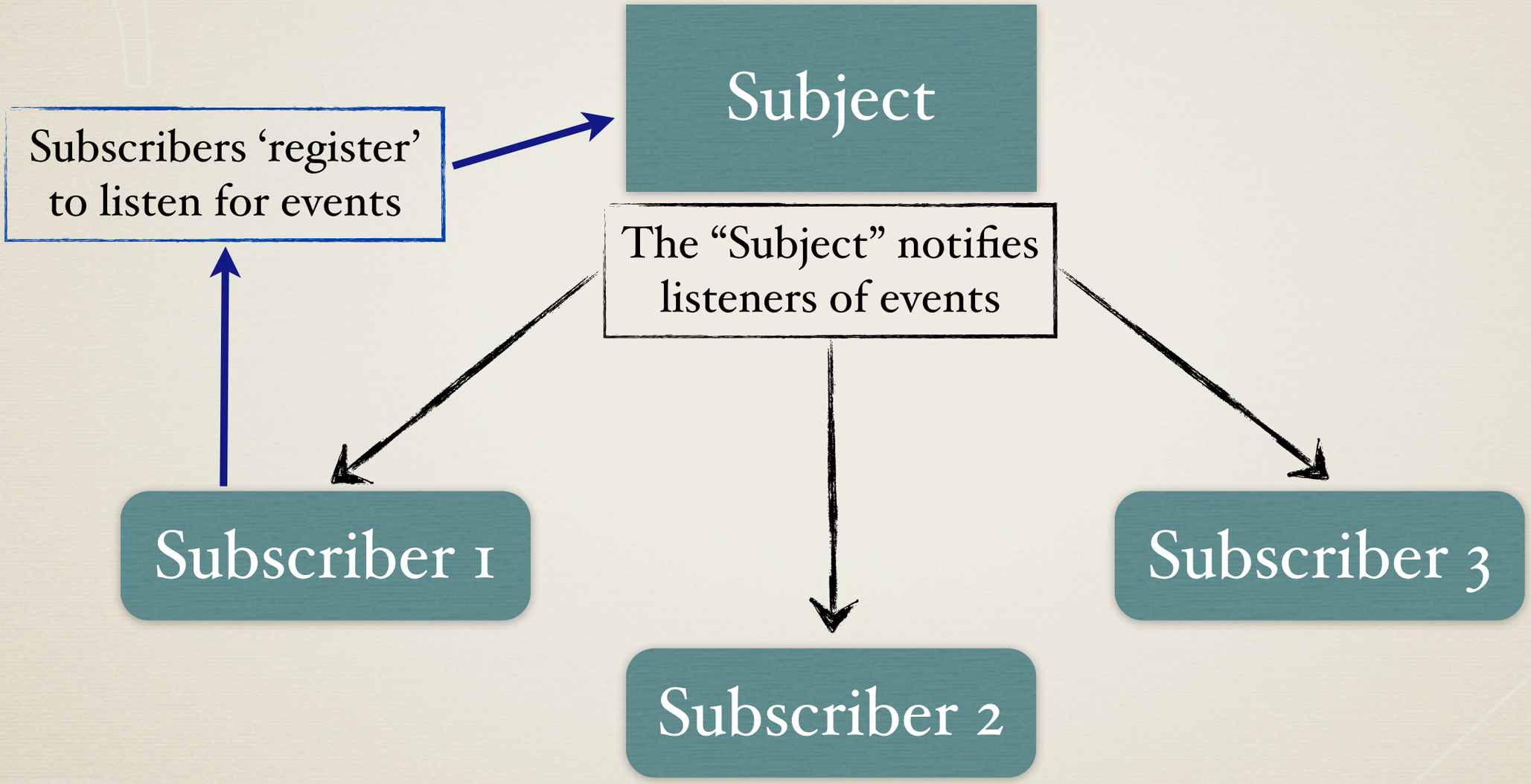
```
// Eventing inside a Backbone View
var MyCustomView = Backbone.View.extend({
    initialize: function() {
        this.model = new CustomModel();
        // listening for any change on the model
        this.model.on("change", this.render);
        // listening for sync events
        this.model.on("sync", this.updateSaveStatus);
    },
    updateSaveStatus: function() {
        // target specific element for surgical update in DOM
    }
    // more stuff defined here....etc
});
```

so - by ‘eventing’, I mean...

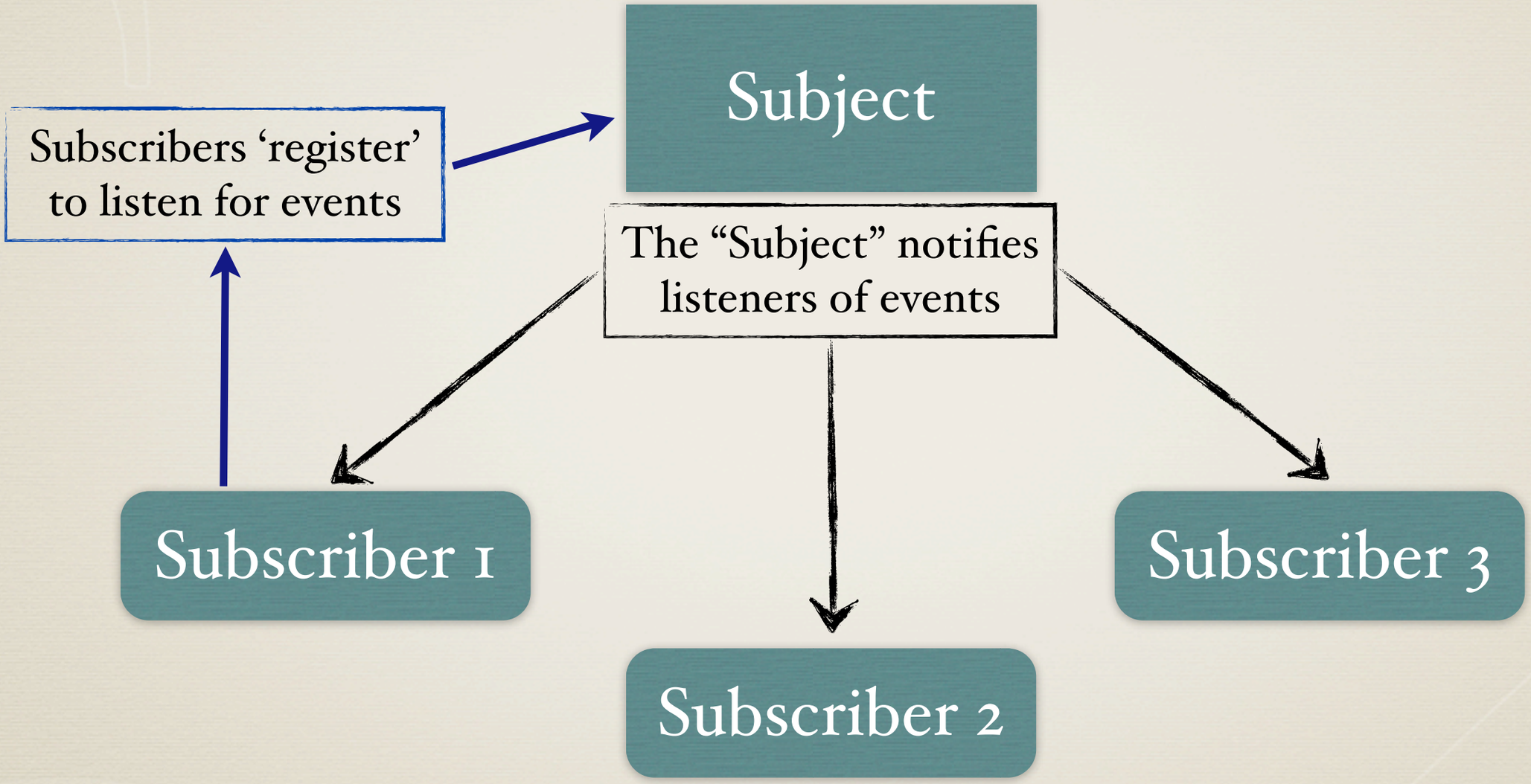
```
// Eventing involving the DOM - we've all seen this
$('#save-btn' ).on("click", function(){
    contactsModel.save();
});
```

```
// Eventing inside a Backbone View
var MyCustomView = Backbone.View.extend({
    initialize: function() {
        this.model = new CustomModel();
        // listening for any change on the model
        this.model.on("change", this.render);
        // listening for sync events
        this.model.on("sync", this.updateSaveStatus);
    },
    updateSaveStatus: function() {
        // target specific element for surgical update in DOM
    }
    // more stuff defined here....etc
});
```

Eventing - in a nutshell



Eventing - in a nutshell



To anyone who recognized this as the "Observer" pattern

Eventing Opinions

- **Advantages**
  - Encourages good SoC
  - Easily add new subscribers
- **Disadvantages**
  - Requires direct reference to subject (tight coupling) & every new subscriber tightens coupling
  - Often widely varying callback signatures
- **Avoid**
  - Assuming the presence of a subscriber/publisher
  - “Mother of all” subscription handlers
  - Mixing business & presentation logic in handlers

Recommendation

Use 'eventing' for:

Internal operations within modules/ components where behavior always ships as a unit

Public event(s) API for a module

and by ‘messaging’, I mean...

```
// product-list.js module
products.on("selected", function(product) {
    bus.products.publish("product.selected", {
        id: product.id,
        name: product.name,
        description: product.description,
        price: product.listPrice,
        qty: product.qtySelected
    });
});

// order.js module
bus.products.subscribe("product.selected", order.addItem);

// summary-view.js module
bus.products.subscribe("product.selected", function(productMsg){
    if(!summary.items[productMsg.id]) {
        summary.items[productMsg.id] = 0;
    }
    summary.items[productMsg.id]++;
    summary.recalculateTotal();
});

// related-products.js module
bus.products.subscribe("product.selected", relatedProds.findSimilar);
```

and by ‘messaging’, I mean...

```
// product-list.js module
products.on("selected", function(product) {
    bus.products.publish("product.selected", {
        id: product.id,
        name: product.name,
        description: product.description,
        price: product.listPrice,
        qty: product.qtySelected
    });
});
```

```
// order.js module
bus.products.subscribe("product.selected", order.addItem);

// summary-view.js module
bus.products.subscribe("product.selected", function(productMsg){
    if(!summary.items[productMsg.id]) {
        summary.items[productMsg.id] = 0;
    }
    summary.items[productMsg.id]++;
    summary.recalculateTotal();
});

// related-products.js module
bus.products.subscribe("product.selected", relatedProds.findSimilar);
```

and by ‘messaging’, I mean...

```
// product-list.js module
products.on("selected", function(product) {
    bus.products.publish("product.selected", {
        id: product.id,
        name: product.name,
        description: product.description,
        price: product.listPrice,
        qty: product.qtySelected
    });
});
```

```
// order.js module
bus.products.subscribe("product.selected", order.addItem);
```

```
// summary-view.js module
bus.products.subscribe("product.selected", function(productMsg){
    if(!summary.items[productMsg.id]) {
        summary.items[productMsg.id] = 0;
    }
    summary.items[productMsg.id]++;
    summary.recalculateTotal();
});

// related-products.js module
bus.products.subscribe("product.selected", relatedProds.findSimilar);
```

and by ‘messaging’, I mean...

```
// product-list.js module
products.on("selected", function(product) {
    bus.products.publish("product.selected", {
        id: product.id,
        name: product.name,
        description: product.description,
        price: product.listPrice,
        qty: product.qtySelected
    });
});

// order.js module
bus.products.subscribe("product.selected", order.addItem);
```

```
// summary-view.js module
bus.products.subscribe("product.selected", function(productMsg){
    if(!summary.items[productMsg.id]) {
        summary.items[productMsg.id] = 0;
    }
    summary.items[productMsg.id]++;
    summary.recalculateTotal();
});
```

```
// related-products.js module
bus.products.subscribe("product.selected", relatedProds.findSimilar);
```

and by ‘messaging’, I mean...

```
// product-list.js module
products.on("selected", function(product) {
    bus.products.publish("product.selected", {
        id: product.id,
        name: product.name,
        description: product.description,
        price: product.listPrice,
        qty: product.qtySelected
    });
});

// order.js module
bus.products.subscribe("product.selected", order.addItem);

// summary-view.js module
bus.products.subscribe("product.selected", function(productMsg){
    if(!summary.items[productMsg.id]) {
        summary.items[productMsg.id] = 0;
    }
    summary.items[productMsg.id]++;
    summary.recalculateTotal();
});

// related-products.js module
bus.products.subscribe("product.selected", relatedProds.findSimilar);
```

Messaging - in a nutshell

Messaging - in a nutshell

Publisher 1

Publisher 2

Message Bus

Subscriber 1

Subscribed to “topic.A”
Subscribed to “topic.B”

Subscriber 2

Subscribed to “topic.A”

Messaging - in a nutshell

Publisher 1

Publisher 2

Publishing on
"topic.A"

Message Bus

Receiving on
"topic.A"

Receiving on
"topic.A"

Subscriber 1

Subscriber 2

Subscribed to "topic.A"
Subscribed to "topic.B"

Subscribed to "topic.A"

Messaging - in a nutshell

Publisher 1

Publisher 2

Message Bus

Subscriber 1

Subscribed to “topic.A”
Subscribed to “topic.B”

Subscriber 2

Subscribed to “topic.A”

Messaging - in a nutshell

Publisher 1

Publisher 2

Publishing on
“topic.B”

Message Bus

Receiving on
“topic.B”

Subscriber 1

Subscribed to “topic.A”
Subscribed to “topic.B”

Subscriber 2

Subscribed to “topic.A”

A More 'Real World' Example

AJAX Wrapper

socket.io Wrapper

Message Bus

Sales Module

A More 'Real World' Example

AJAX Wrapper

socket.io Wrapper

Subscribe to
customer.getAllSales

Message Bus

Subscribe to
customer.sales

Subscribe to
order.new.*

Sales Module

A More 'Real World' Example

AJAX Wrapper

socket.io Wrapper

Publish
customer.sales

Message Bus

customer.sales

Sales Module

A More 'Real World' Example

AJAX Wrapper

socket.io Wrapper

Message Bus

Sales Module

A More 'Real World' Example

AJAX Wrapper

socket.io Wrapper

Publishing
order.new.12406

Message Bus

order.new.12406

Sales Module

OMG, MOAR Opinions

* **Advantages**
  * promotes de-coupling/good SoC
  * testability!
  * portability of components/modules
  * enables interoperability
* **Disadvantages**
  * Can lead to loss of clarity if poorly abstracted

Things to Avoid

* Passing behavior and/or otherwise-hidden references as message payload
* Mutating the message (*at least* be clear about message transformation if you must do it)
* Depending on subscriber priority for critical operations
* Message storms (WAT?!)
* 'Littering' your app code with publish/subscribe everywhere (try to abstract inside infrastructure where possible)

Recommendation

Use messaging "at the seams" between modules/components to allow for 'compose-ability' of an application from smaller, interoperable-but-loosely-coupled parts

OK, Great - Now What?

* Eventing options abound:
  * Backbone has Backbone.Events (can be mixed into almost anything)
  * jQuery has custom events
  * EventEmitter
* Message-bus options
  * postal.js (hey, I'm biased)
  * pubsub.js (Peter Higgins) - super lightweight
  * Postman (Aaron Powell)

Message Broker

* a.k.a - “The Bus”
* The broker - or channels it provides - is the only common/shared reference between otherwise de-coupled components using message-passing to communicate
* API commonly provides:
  * publish
  * subscribe
  * unsubscribe
* Other API possibilities include adding/removing wire taps, getting persistent handle to a channel

Channels & Topics

* Topics
  * Routing key
  * Determines recipient(s)
* Channels
  * Logical groupings/partitions of topics
  * The ‘endpoint’ for publishers/subscribers to call
  * Most client side messaging implementations are single channel (i.e. - you only specify topic) [postal.js is multi-channel]

The Message

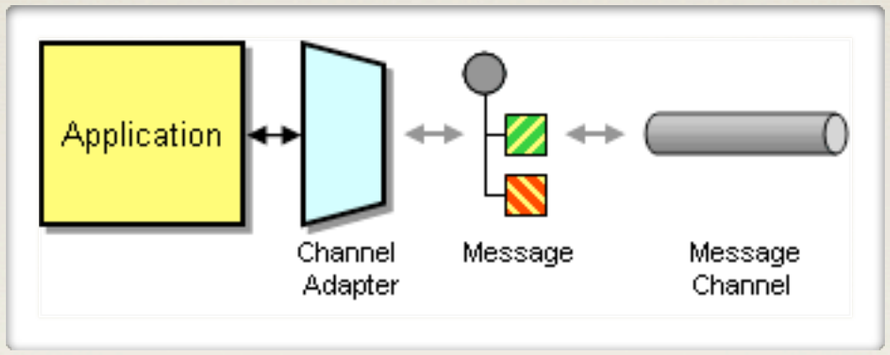
An ‘envelope’ provides a consistent, structured & extendable way to publish & receive data

* Default postal.js envelope:
  * Channel
  * Topic
  * Time Stamp
  * Data
* Easily add other envelope metadata (reply-to, correlation ID, version, etc.)
* Pro tip: Make sure it’s serializable

FINALLY: {Code}

- Channel
- Selective Subscriber
- Message Endpoint
- Message ‘Types’
- Format Indicator
- Message Filter
- Idempotent Receiver
- Correlation Identifier
- Request-Response
- Wire Tap
- Aggregator
- Routing Slip

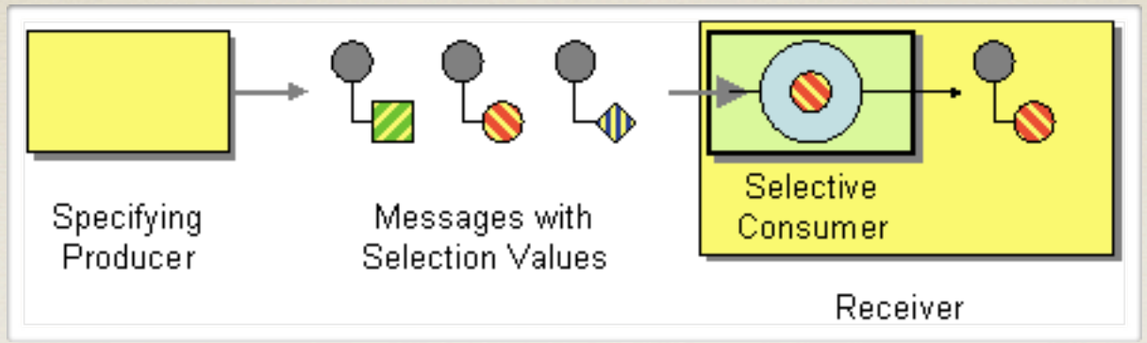
Channels




```
// getting an instance of a postal ChannelDefinition
// here we only provide the topic, which gives us the default channel ("/"):
var postal_sub1 = postal.channel("order.process");

// and here we provide a specific channel name with the topic:
var postal_sub2 = postal.channel("orders", "order.process");
```

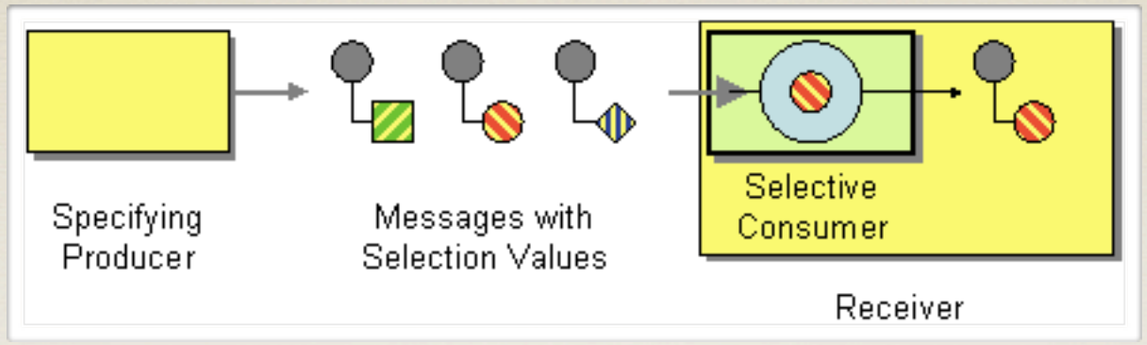
Selective Consumer



Using Channels + Topics

```
// postal.js topic matching supports wildcards
// if you don't have a channel reference already, you can use top-level postal.subscribe
// "#" matches 0-1 'words' (words are period-delimited alpha-numeric sequences in the topic)
// This would match "Customer.firstName.error" & "Address.error"
var validationListener = postal.subscribe( {
    channel  : "validation",
    topic    : "#.error",
    callback : function ( data, envelope ) {
        summary.validationErrors.add( data );
    }
} );
// "*" matches exactly 1 'word'.
// This would match "Customer.ABC123.fetch", as well as "Customer.8675309.fetch"
var readRequestHandler = postal.subscribe( {
    channel  : "data",
    topic    : "Customer.*.fetch",
    callback : function ( data, envelope ) {
        amplify.request(
            "Customer-get",
            { customerId: data.customerId },
            function(resp) {
                postal.publish({
                    channel: "data",
                    topic: envelope.replyTo,
                    data: resp.data
                })
            });
    }
});
```

Selective Consumer

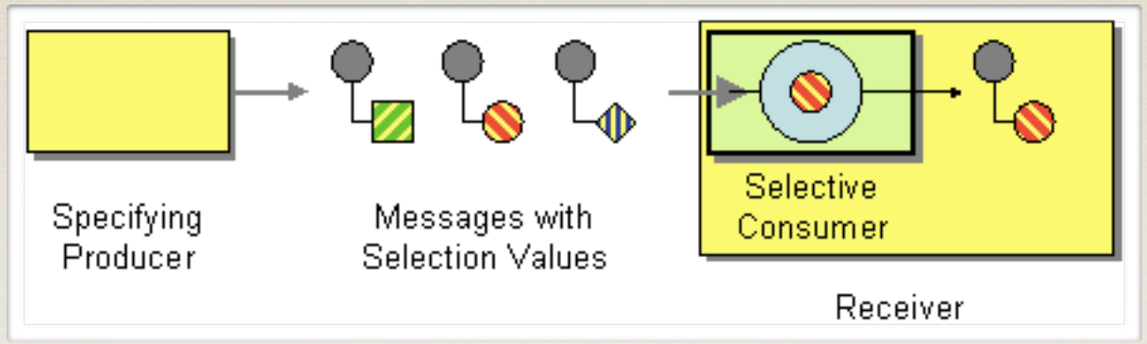


Using Channels + Topics

```
// postal.js topic matching supports wildcards
// if you don't have a channel reference already, you can use top-level postal.subscribe
// "#" matches 0-1 'words' (words are period-delimited alpha-numeric sequences in the topic)
// This would match "Customer.firstName.error" & "Address.error"
var validationListener = postal.subscribe( {
    channel  : "validation",
    topic    : "#.error",
    callback : function ( data, envelope ) {
        summary.validationErrors.add( data );
    }
} );
```

```
var readRequestHandler = postal.subscribe( {
    channel  : "data",
    topic    : "Customer.*.fetch",
    callback : function ( data, envelope ) {
        amplify.request(
            "Customer-get",
            { customerId: data.customerId },
            function(resp) {
                postal.publish({
                    channel: "data",
                    topic: envelope.replyTo,
                    data: resp.data
                })
            });
    }
});
```

Selective Consumer



Using Channels + Topics

```
// postal.js topic matching supports wildcards
// if you don't have a channel reference already, you can use top-level postal.subscribe
// "#" matches 0-1 'words' (words are period-delimited alpha-numeric sequences in the topic)
// This would match "Customer.firstName.error" & "Address.error"
var validationListener = postal.subscribe( {
    channel  : "validation",
    topic    : "#.error",
    callback : function ( data, envelope ) {
        summary.validationErrors.add( data );
// "*" matches exactly 1 'word'.
// This would match "Customer.ABC123.fetch", as well as "Customer.8675309.fetch"
var readRequestHandler = postal.subscribe( {
    channel  : "data",
    topic    : "Customer.*.fetch",
    callback : function ( data, envelope ) {
        amplify.request(
            "Customer-get",
            { customerId: data.customerId },
            function(resp) {
                postal.publish({
                    channel: "data",
                    topic: envelope.replyTo,
                    data: resp.data
                })
            });
    }
});
```

Selective Consumer

Constraints & more

```
31  // using withThrottle(x) to limit # of times
32  // a subscriber callback is invoked per interval
33  // Subscriber is invoked at most once per 500ms:
34  var locationListener = gpsChannel.subscribe(
35          "location.move",
36          function(data, envelope) {
37              map.plotLocation(data.x, data.y);
38          }
39      ).withThrottle(500);
40
41  // using withConstraint to apply a predicate
42  // each time postal attempts to invoke callback
43  var logger = debugChannel
44                  .subscribe("#", this.socketLogger)
45                  .withContext(app)
46                  .withConstraint(function(data, envelope) {
47                      return app.socketConn.online;
48                  });
49
50  // unsubscribing after {x} number of invocations
51  var doctor_who = tardis.subscribe(
52      "enemy.defies.doctor",
53      function(data, envelope){
54          doctor.giveThemAChance(data.enemy);
55      } ).disposeAfter(1);
```

Selective Consumer

Constraints & more

```
31  // using withThrottle(x) to limit # of times
32  // a subscriber callback is invoked per interval
33  // Subscriber is invoked at most once per 500ms:
34  var locationListener = gpsChannel.subscribe(
35          "location.move",
36          function(data, envelope) {
37              map.plotLocation(data.x, data.y);
38          }
39      ).withThrottle(500);
41  // using withConstraint to apply a predicate
42  // each time postal attempts to invoke callback
43  var logger = debugChannel
44                  .subscribe("#", this.socketLogger)
45                  .withContext(app)
46                  .withConstraint(function(data, envelope) {
47                      return app.socketConn.online;
48                  });
49
50  // unsubscribing after {x} number of invocations
51  var doctor_who = tardis.subscribe(
52      "enemy.defies.doctor",
53      function(data, envelope){
54          doctor.giveThemAChance(data.enemy);
55      } ).disposeAfter(1);
```

Selective Consumer

Constraints & more

```
31  // using withThrottle(x) to limit # of times
32   // a subscriber callback is invoked per interval
33  // Subscriber is invoked at most once per 500ms:
34   var locationListener = gpsChannel.subscribe(
35           "location.move",
36           function(data, envelope) {
37               map.plotLocation(data.x, data.y);
38           }
39       ).withThrottle(500);
41  // using withConstraint to apply a predicate
42  // each time postal attempts to invoke callback
43   var logger = debugChannel
44                       .subscribe("#", this.socketLogger)
45                       .withContext(app)
46                       .withConstraint(function(data, envelope) {
47                           return app.socketConn.online;
48                       });
49
50   // unsubscribing after {x} number of invocations
51   var doctor_who = tardis.subscribe(
52       "enemy.defies.doctor",
53       function(data, envelope){
54           doctor.giveThemAChance(data.enemy);
55       } ).disposeAfter(1);
```

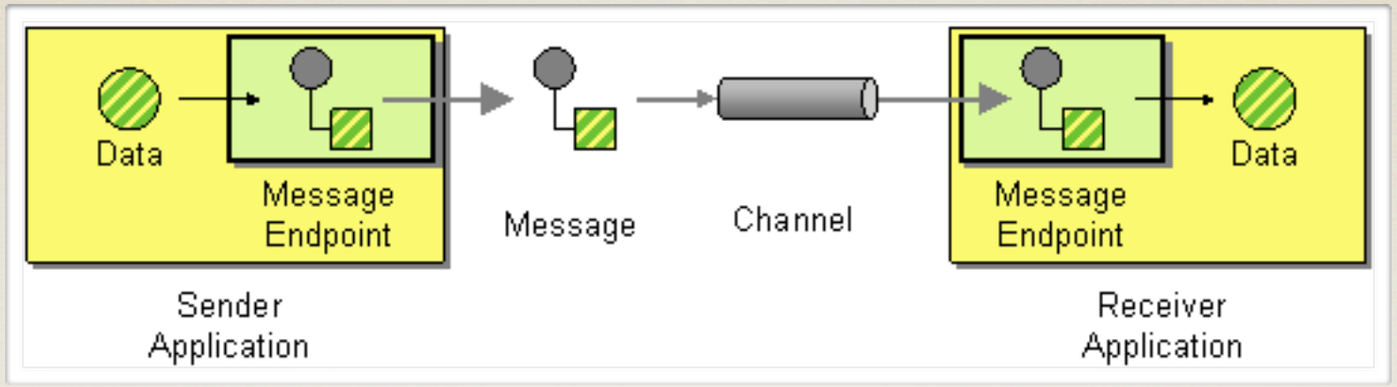
Selective Consumer

Constraints & more

```
31  // using withThrottle(x) to limit # of times
32   // a subscriber callback is invoked per interval
33  // Subscriber is invoked at most once per 500ms:
34   var locationListener = gpsChannel.subscribe(
35           "location.move",
36           function(data, envelope) {
37               map.plotLocation(data.x, data.y);
38           }
39       ).withThrottle(500);
40
41  // using withConstraint to apply a predicate
42  // each time postal attempts to invoke callback
43   var logger = debugChannel
44                       .subscribe("#", this.socketLogger)
45                       .withContext(app)
46                       .withConstraint(function(data, envelope) {
47                           return app.socketConn.online;
48                       });
```

```
50  // unsubscribing after {x} number of invocations
51  var doctor_who = tardis.subscribe(
52      "enemy.defies.doctor",
53      function(data, envelope){
54          doctor.giveThemAChance(data.enemy);
55      } ).disposeAfter(1);
```

Message Endpoint



"Connect an application to a messaging channel using a *Message Endpoint*, a client of the messaging system that the application can then use to send or receive messages."

Message Endpoints Behaving Badly

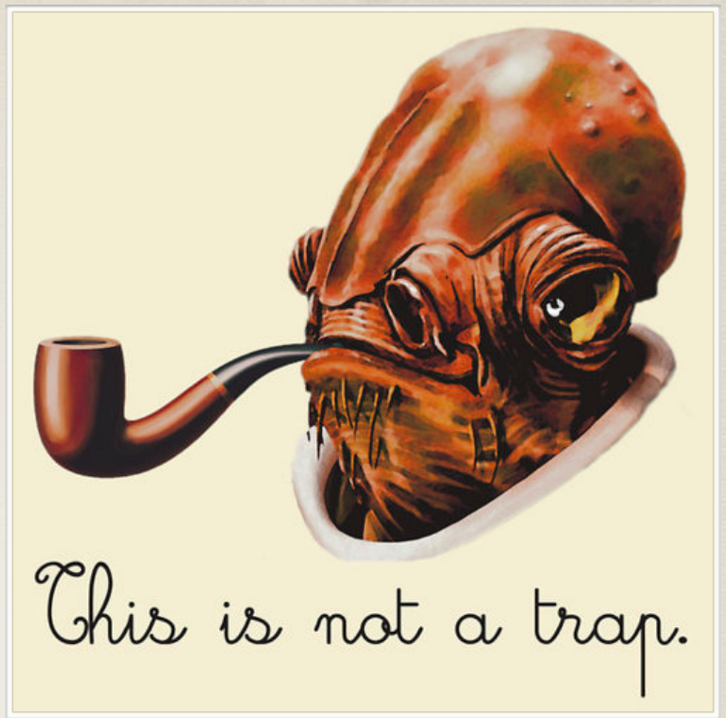
"Making a message bus the *only* means of exposing a public API is a trap"

```
1   // DON'T DO THIS. SERIOUSLY. MEAN IT.
2   var HateInspiringInvoicer = function() {
3     // other functionality in here somewhere...
4     var processOrder = function ( data ) {
5       // order processing and invoicing logic
6     };
7     // aaaaaand then THIS
8     bus.subscribe("order.process", function(data) {
9       var invoice = processOrder(data);
10      bus.publish({
11        channel: "invoicing",
12        topic: "order.invoiced",
13        data: invoice
14      });
15    } );
16  };
17
18  // OH LOOK! We have a magic Invoicer that has no real API
19  var whyGodWhy = new HateInspiringInvoicer();
```

Message Endpoints - a ‘better’ way™

Advice from the trenches:

* Your application objects should *not* need to know about the message bus
* Give them a public API that clients (w/ a direct reference) could consume.




```
21  // DO THIS. ANGELS WILL APPEAR AND FEED YOU CHEESECAKE.
22   // Well, not really. But angels and developers alike
23  // will salute you over cheesecake. Mean it.
24  var Invoicer = function() {
25     EventEmitter.call(this);
26  };
27   sys.inherits(Invoicer, EventEmitter);
28
29   // we have a public "API" to process an order
30  Invoicer.prototype.processOrder = function(order) {
31     var invoice;
32     // order processing and invoicing logic
33     this.emit("order.invoiced", invoice);
34  };
```

Message Endpoints - a ‘better’ way™

Message Endpoint adapter for the Invoicer

```
1   // Now let's create a message endpoint for the Invoicer
2   // This could (& should) be part of a structured bus adapter for the app
3   var adaptInvoicer = function(invoicer, bus) {
4       var callbacks = [];
5       var procOrderSub = bus.orders.subscribe("order.process", function(data, env){
6           invoicer.processOrder(data);
7       });
8       callbacks.push(procOrderSub.unsubscribe);
9
10      // ...adapt other APIs to the bus via subscriptions here
11
12      // tie local event publishing to the bus
13      var onInvoiced = function(invoice) {
14          bus.invoicing.publish({ topic: "order.invoiced", data: invoice });
15      };
16      invoicer.on("order.invoiced", onInvoiced);
17      callbacks.push(function() { invoicer.off("order.invoiced", onInvoiced) });
18
19      return function() {
20          _.each(callbacks, function(unsubscribe) {
21              unsubscribe();
22          });
23      }
24  };
25
26  var invoicer = new Invoicer(),
27      removeAdapter = adaptInvoicer(invoicer, app.bus);
```

Message Endpoints - a ‘better’ way™

Message Endpoint adapter for the Invoicer

```
1  // Now let's create a message endpoint for the Invoicer
2  // This could (& should) be part of a structured bus adapter for the app
3  var adaptInvoicer = function(invoicer, bus) {
4      var callbacks = [];
5      var procOrderSub = bus.orders.subscribe("order.process", function(data, env){
6          invoicer.processOrder(data);
7      });
8      callbacks.push(procOrderSub.unsubscribe);
9
10     // ...adapt other APIs to the bus via subscriptions here
11
12     // tie local event publishing to the bus
13     var onInvoiced = function(invoice) {
14         bus.invoicing.publish({ topic: "order.invoiced", data: invoice });
15     };
16     invoicer.on("order.invoiced", onInvoiced);
17     callbacks.push(function() { invoicer.off("order.invoiced", onInvoiced) });
18
19     return function() {
20         _.each(callbacks, function(unsubscribe) {
21             unsubscribe();
22         });
23     }
24 };
25
26 var invoicer = new Invoicer(),
27     removeAdapter = adaptInvoicer(invoicer, app.bus);
```

Message Endpoints - a ‘better’ way™

Message Endpoint adapter for the Invoicer

```
1  // Now let's create a message endpoint for the Invoicer
2  // This could (& should) be part of a structured bus adapter for the app
3  var adaptInvoicer = function(invoicer, bus) {
4      var callbacks = [];
5      var procOrderSub = bus.orders.subscribe("order.process", function(data, env){
6          invoicer.processOrder(data);
7      });
8      callbacks.push(procOrderSub.unsubscribe);
9
10     // ...adapt other APIs to the bus via subscriptions here
11
12     // tie local event publishing to the bus
13     var onInvoiced = function(invoice) {
14         bus.invoicing.publish({ topic: "order.invoiced", data: invoice });
15     };
16     invoicer.on("order.invoiced", onInvoiced);
17     callbacks.push(function() { invoicer.off("order.invoiced", onInvoiced) });
18
19     return function() {
20         _.each(callbacks, function(unsubscribe) {
21             unsubscribe();
22         });
23     }
24 };
25
26  var invoicer = new Invoicer(),
27      removeAdapter = adaptInvoicer(invoicer, app.bus);
```

Message Endpoints - a ‘better’ way™

Message Endpoint adapter for the Invoicer

```
1  // Now let's create a message endpoint for the Invoicer
2  // This could (& should) be part of a structured bus adapter for the app
3  var adaptInvoicer = function(invoicer, bus) {
4      var callbacks = [];
5      var procOrderSub = bus.orders.subscribe("order.process", function(data, env){
6          invoicer.processOrder(data);
7      });
8      callbacks.push(procOrderSub.unsubscribe);
9
10     // ...adapt other APIs to the bus via subscriptions here
11
12     // tie local event publishing to the bus
13     var onInvoiced = function(invoice) {
14         bus.invoicing.publish({ topic: "order.invoiced", data: invoice });
15     };
16     invoicer.on("order.invoiced", onInvoiced);
17     callbacks.push(function() { invoicer.off("order.invoiced", onInvoiced) });
18
19     return function() {
20         _.each(callbacks, function(unsubscribe) {
21             unsubscribe();
22         });
23     }
24 };
25
26 var invoicer = new Invoicer(),
27     removeAdapter = adaptInvoicer(invoicer, app.bus);
```

Message Types

```
1   // Document Message - useful to transfer data between modules/apps
2   bus.orders.publish( {
3     id : "Z4356tGFx1",
4     customer : {
5       firstName : "Jim",
6       lastName  : "Cowart",
7       location  : "Chattanooga"
8     },
9     items : [
10      { productId : 1, description : "A Brain", cost : 1000000.23 },
11      { productId : 2, description : "A Life",  cost : 2000000.76 },
12      { productId : 3, description : "A Clue",  cost : NaN }
13    ]
14  } );
15
16  // Command Message - useful to tell remote target to perform an action
17  bus.fbi.publish( {
18    topic : "joint.irs.strike-team",
19    data  : {
20      command   : "raidCompany",
21      companyId : "sommet",
22      replyTo   : "tax.delinquent.operations"
23    }
24  } );
25
26  // Event Message - notify listeners of an event that occurred
27  bus.users.publish( {
28    topic : "user.updated",
29    data  : {
30      timeStamp     : "2010-07-06T09:00:00.000Z",
31      updatedBy     : "bwhitfield",
32      updatedUserId : "jcowart",
33      changed       : {
34        location : { old : "Nashville, TN", new : "Chattanooga, TN" }
35      }
36    }
37  } );
```

Document Message

Command Message

Event Message

Message Types

```
// Document Message - useful to transfer data between modules/apps
bus.orders.publish( {
  id : "Z4356tGFx1",
  customer : {
    firstName : "Jim",
    lastName  : "Cowart",
    location  : "Chattanooga"
  },
  items : [
    { productId : 1, description : "A Brain", cost : 1000000.23 },
    { productId : 2, description : "A Life",  cost : 2000000.76 },
    { productId : 3, description : "A Clue",  cost : NaN }
  ]
} );
```

Document Message

```
  topic : "joint.irs.strike-team",
  data  : {
    command   : "raidCompany",
    companyId : "sommet",
    replyTo   : "tax.delinquent.operations"
  }
} );

// Event Message - notify listeners of an event that occurred
bus.users.publish( {
  topic : "user.updated",
  data  : {
    timeStamp     : "2010-07-06T09:00:00.000Z",
    updatedBy     : "bwhitfield",
    updatedUserId : "jcowart",
    changed       : {
      location : { old : "Nashville, TN", new : "Chattanooga, TN" }
    }
  }
} );
```

Command Message

Event Message

Message Types

```
// Document Message - useful to transfer data between modules/apps
bus.orders.publish( {
  id : "Z4356tGFx1",
  customer : {
    firstName : "Jim",
    lastName  : "Cowart",
    location  : "Chattanooga"
  },
  items : [
    { productId : 1, description : "A Brain", cost : 1000000.23 },
    { productId : 2, description : "A Life",  cost : 2000000.76 },
    { productId : 3, description : "A Clue",  cost : NaN }
  ]
```

Document Message

```
// Command Message - useful to tell remote target to perform an action
bus.fbi.publish( {
  topic : "joint.irs.strike-team",
  data  : {
    command   : "raidCompany",
    companyId : "sommet",
    replyTo   : "tax.delinquent.operations"
  }
} );
```

Command Message

```
// Event Message - notify listeners of an event that occurred
bus.users.publish( {
  topic : "user.updated",
  data  : {
    timeStamp     : "2010-07-06T09:00:00.000Z",
    updatedBy     : "bwhitfield",
    updatedUserId : "jcowart",
    changed       : {
      location : { old : "Nashville, TN", new : "Chattanooga, TN" }
    }
  }
} );
```

Event Message

Message Types

```
// Document Message - useful to transfer data between modules/apps
bus.orders.publish( {
  id : "Z4356tGFx1",
  customer : {
    firstName : "Jim",
    lastName  : "Cowart",
    location  : "Chattanooga"
  },
  items : [
    { productId : 1, description : "A Brain", cost : 1000000.23 },
    { productId : 2, description : "A Life",  cost : 2000000.76 },
    { productId : 3, description : "A Clue",  cost : NaN }
  ]
} );
```

Document Message

```
// Command Message - useful to tell remote target to perform an action
bus.fbi.publish( {
  topic : "joint.irs.strike-team",
  data  : {
    command   : "raidCompany",
    companyId : "sommet",
    replyTo   : "tax.delinquent.operations"
```

Command Message

```
// Event Message - notify listeners of an event that occurred
bus.users.publish( {
  topic : "user.updated",
  data  : {
    timeStamp      : "2010-07-06T09:00:00.000Z",
    updatedBy      : "bwhitfield",
    updatedUserId : "jcowart",
    changed        : {
      location : { old : "Nashville, TN", new : "Chattanooga, TN" }
    }
  }
} );
```

Event Message

Format Indicator

```
var usrTranslator = {
    "1" : function(data) {
        return utils.parseAddressFromString(data);
    },
    "2" : function(data) {
        var _data = data;
        _data.firstName = data.name.first;
        _data.lastName = data.name.last;
        delete _data.name;
        return _data;
    }
};

bus.users.subscribe("user.import", function(data, envelope){
    if(usrTranslator.hasOwnProperty(envelope.version)) {
        var _data = usrTranslator[envelope.version](data);
        userStore.add(_data);
    }
});
```

* Specify format/version of message
* Helps abstract change over time
* Enables better interoperability

Idempotent Receiver

```
1   var inventory = [];
2   var lootChannel = postal.channel('loot.acquired');
3   lootChannel.subscribe(function (item) {
4       inventory.push(item);
5       if (inventory.length === 3) {
6           postal.publish({
7               channel : 'achievement.earned',
8               topic   : 'full.armor.set',
9               data    : inventory
10          });
11      }
12  }).distinct();
13
14  lootChannel.publish('sword');
15  lootChannel.publish('shield');
16  lootChannel.publish('sword');  //duplicate ignored
17  lootChannel.publish('shield'); //duplicate ignored
18  lootChannel.publish('boots');
19  //achievement earned!
```

* *Opinion*: this is a consumer's responsibility
* But it's OK to get some help from the framework

Correlation Identifier

“How does a requestor that has received a reply know which request this is the reply for?”

```
var viewModel = new SecretSauceModel.extend({
    initialize: function() {
        app.bus.data.subscribe({
            topic: this.getTopic() + ".readComplete",
            callback: this.parseReadComplete
        } ).withContext(this)
           .withConstraint(function(d, e){
                return e.headers.correlationId === this.currentRequestId;
            }
        );
    },
    fetch: function(options) {
        this.currentRequestId = this.generateRequestId(); // for example: Az367Vxw
        app.bus.data.publish({
            headers: {
                replyTo       : this.getTopic() + ".readComplete",
                correlationId : this.currentRequestId
            },
            topic : this.getTopic() + ".read",
            data  : {
                id      : this.id,
                options : options
            }
        })
    }
});
```

Correlation Identifier

“How does a requestor that has received a reply know which request this is the reply for?”

```
var viewModel = new SecretSauceModel.extend({
    initialize: function() {
        app.bus.data.subscribe({
            topic: this.getTopic() + ".readComplete",
            callback: this.parseReadComplete
        } ).withContext(this)
           .withConstraint(function(d, e){
                return e.headers.correlationId === this.currentRequestId;
            }
        );
    },
    fetch: function(options) {
        this.currentRequestId = this.generateRequestId(); // for example: Az367Vxw
        app.bus.data.publish({
            headers: {
                replyTo       : this.getTopic() + ".readComplete",
                correlationId : this.currentRequestId
            },
            topic : this.getTopic() + ".read",
            data  : {
                id      : this.id,
                options : options
            }
        })
    }
});
```

Correlation Identifier

“How does a requestor that has received a reply know which request this is the reply for?”

```
var viewModel = new SecretSauceModel.extend({
    initialize: function() {
        app.bus.data.subscribe({
            topic: this.getTopic() + ".readComplete",
            callback: this.parseReadComplete
        } ).withContext(this)
           .withConstraint(function(d, e){
                return e.headers.correlationId === this.currentRequestId;
            }
        );
    },
        this.currentRequestId = this.generateRequestId(); // for example: Az367Vxw
        app.bus.data.publish({
            headers: {
                replyTo       : this.getTopic() + ".readComplete",
                correlationId : this.currentRequestId
            },
            topic : this.getTopic() + ".read",
            data  : {
                id      : this.id,
                options : options
            }
        })
    }
});
```

Wire Tap

- "Splices" into the message bus - gets a copy of each message
- Good for logging & diagnostics
- Example:

```
1    // Log every message to the console
2    app.debugWireTap = new DiagnosticsWireTap("allTheThings", function (x) {
3        console.log(x);
4    });
5
6    // log only messages from the data channel
7    app.debugWireTap = new DiagnosticsWireTap("onlyDataStuff", function (x) {
8        console.log(x);
9    }, [ { channel: "data" }]);
```

Wire taps can be easily adapted to push messages into storage, transmit them to remote endpoints for analysis and/or replay, etc.

Aggregator

How do we combine the results of individual, but related messages so that they can be processed as a whole?

```
postal.when( [
        { channel : "customer", topic : "AzXk6t7.readSuccess" },
        { channel : "orders",   topic : "AzXk6t7.readSuccess" },
        { channel : "contacts", topic : "AzXk6t7.readSuccess" },
        { channel : "ar",       topic : "AzXk6t7.readSuccess" }
], function ( a, b, c, d ) {
    var customer = _.extend({
            orders      : b.items,
            contacts    : c.items,
            receivables : d.items
        }, a);
        postal.publish({
            channel: "customer",
            topic: "aggregate.assembled",
            data: customer
        })
} );
```

* deferred-style behavior
* waits for all messages to arrive before invoking callback

Aggregator

How do we combine the results of individual, but related messages so that they can be processed as a whole?

```
postal.when( [
    { channel : "customer", topic : "AzXk6t7.readSuccess" },
    { channel : "orders",   topic : "AzXk6t7.readSuccess" },
    { channel : "contacts", topic : "AzXk6t7.readSuccess" },
    { channel : "ar",       topic : "AzXk6t7.readSuccess" }
], function( a, b, c, d ) {
    var customer = _.extend({
        orders      : b.items,
        contacts    : c.items,
        receivables : d.items
    }, a);
    postal.publish({
        channel: "customer",
        topic: "aggregate.assembled",
        data: customer
    })
} );
```

* deferred-style behavior
* waits for all messages to arrive before invoking callback

Aggregator

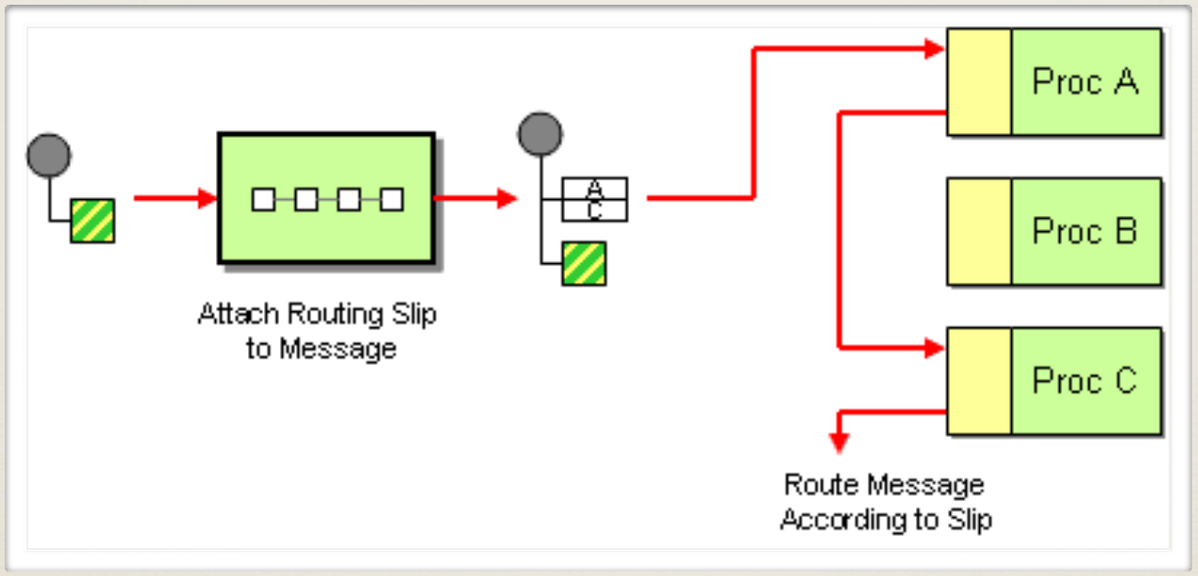
How do we combine the results of individual, but related messages so that they can be processed as a whole?

```
postal.when( [
    { channel : "customer", topic : "AzXk6t7.readSuccess" },
    { channel : "orders",   topic : "AzXk6t7.readSuccess" },
    { channel : "contacts", topic : "AzXk6t7.readSuccess" }
], function ( a, b, c, d ) {
    var customer = _.extend({
        orders      : b.items,
        contacts    : c.items,
        receivables : d.items
    }, a);
    postal.publish({
        channel: "customer",
        topic: "aggregate.assembled",
        data: customer
    })
} );
```

* deferred-style behavior
* waits for all messages to arrive before invoking callback

Routing Slip

"How do we route a message consecutively through a series of processing steps when the sequence of steps is not known at design-time and may vary for each message?"



Routing Slip

```
postal.publish({
    channel : "orders",
    topic   : "order.checkout",
    headers : {
        routing : [
            { channel: "orders",    topic: "order.checkout"   },
            { channel: "orders",    topic: "product.shipping" },
            { channel: "invoicing", topic: "invoice.taxes"    },
            { channel: "invoicing", topic: "invoice.charge"   }
        ]
    },
    data: order
});

postal.subscribe({
    channel  : "orders",
    topic    : "order.checkout",
    callback : function(data, envelope) {
        // process order data
        // then forward to next destination
        var next = envelope.headers.routing.splice(0,1);
        postal.publish({
            channel : next.channel,
            topic   : next.topic,
            headers : {
                routing: envelope.headers.routing
            },
            data    : data
        });
    }
});
```

Routing Slip

```
1  postal.publish({
2      channel : "orders",
3      topic   : "order.checkout",
4      headers : {
5          routing : [
6              { channel: "orders",     topic: "order.checkout"    },
7              { channel: "orders",     topic: "product.shipping" },
8              { channel: "invoicing", topic: "invoice.taxes"     },
9              { channel: "invoicing", topic: "invoice.charge"    }
10         ]
12     data: order
13 });
14
15 postal.subscribe({
16     channel  : "orders",
17     topic    : "order.checkout",
18     callback : function(data, envelope) {
19         // process order data
20         // then forward to next destination
21         var next = envelope.headers.routing.splice(0,1);
22         postal.publish({
23             channel : next.channel,
24             topic   : next.topic,
25             headers : {
26                 routing: envelope.headers.routing
27             },
28             data    : data
29         });
30     }
31 });
```

Routing Slip

```
postal.publish({
    channel : "orders",
    topic   : "order.checkout",
    headers : {
        routing : [
            { channel: "orders",    topic: "order.checkout"   },
            { channel: "orders",    topic: "product.shipping" },
            { channel: "invoicing", topic: "invoice.taxes"    },
            { channel: "invoicing", topic: "invoice.charge"   }
        ]
    },
    data: order
});
```

```
    // process order data
    // then forward to next destination
    var next = envelope.headers.routing.splice(0,1);
    postal.publish({
        channel : next.channel,
        topic   : next.topic,
        headers : {
            routing: envelope.headers.routing
        },
        data    : data
    });
```

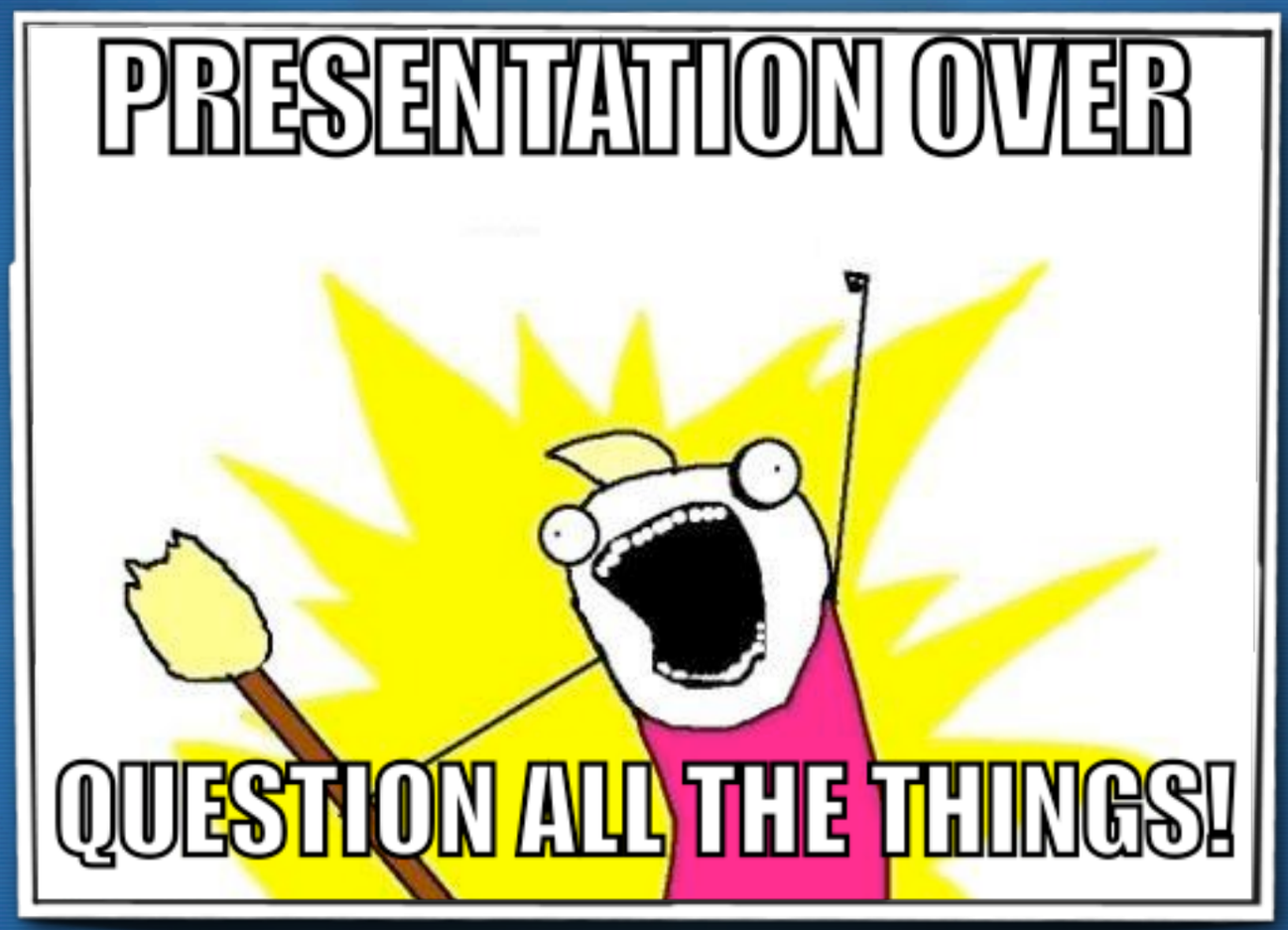
Summary

* Event delegation works well within (and at) boundaries
* Messaging works well at the seams/between modules
* Both promote modular and testable design
* The principles and patterns we've discussed:
  * help lead to expressive code
  * can protect against leaky abstractions & tight coupling
  * provide extensibility as your application changes

Other Resources

* http://www.eaipatterns.com
* Great presentation by Kevin Hakanson:
  https://github.com/hakanson/tccc11
* postal.js wiki: https://github.com/ifandelse/postal.js/wiki
* Fascinating different approach w/ js-signals:
  http://millermedeiros.github.com/js-signals/
* Addy Osmani on understanding pub/sub:
  http://msdn.microsoft.com/en-us/magazine//hh201955.aspx

Code/Slides for this presentation -
http://bit.ly/javascript-messaging-patterns



Q & A