Patterns For Large-Scale JavaScript Application Architecture

Written by: Addy Osmani. Technical Review: Andrée Hansson

Today we're going to discuss an effective set of patterns for large-scale JavaScript application architecture. The material is based on my talk of the same name, last presented at LondonJS and inspired by previous work by Nicholas Zakas.

Who am I and why am I writing about this topic?

I'm currently a JavaScript and UI developer at AOL helping to plan and write the front-end architecture to our next generation of client-facing applications. As these applications are both complex and often require an architecture that is scalable and highly-reusable, it's one of my responsibilities to ensure the patterns used to implement such applications are as sustainable as possible.

I also consider myself something of a design pattern enthusiast (although there are far more knowledgeable experts on this topic than I). I've previously written the creative-commons book 'Essential JavaScript Design Patterns' and am in the middle of writing the more detailed follow up to this book at the moment.

Can you summarize this article in 140 characters?

In the event of you being short for time, here's the tweet-sized summary of this article:

> Decouple app. architecture w/module,facade & mediator patterns. Mods publish msgs, mediator acts as pub/sub mgr & facade handles security

What exactly *is* a 'large' JavaScript application?

Before we begin, let us attempt to define what we mean when we refer to a JavaScript application as being significantly 'large'. This is a question I've found still challenges developers with many years of experience in the field and the answer to this can be quite subjective.

As an experiment, I asked a few intermediate developers to try providing their definition of it informally. One developer suggested 'a JavaScript application with over 100,000 LOC' whilst another suggested 'apps with over 1MB of JavaScript code written in-house'. Whilst valiant (if not scary) suggestions, both of these are **incorrect** as the size of a codebase does not always correlate to application complexity - those 100,000 LOC could easily represent quite trivial code.

My own definition may or may not be universally accepted, but I believe that it's closer to what a large application actually represents.

> In my view, large-scale JavaScript apps are **non-trivial** applications requiring **significant** developer effort to maintain, where most heavy lifting of data manipulation and display falls to the **browser**.

The last part of this definition is possibly the most significant.

Let's review your current architecture.

> If working on a significantly large JavaScript application, remember to dedicate **sufficient time** to planning the underlying architecture that makes the most sense. It's often more complex than you may initially imagine.

I can't stress the importance of this enough - some developers I've seen approach larger applications have stepped back and said 'Okay. Well, there are a set of ideas and patterns that worked well for me on my last medium-scale project. Surely they should mostly apply to something a little larger, right?'. Whilst this may be true to an extent, please don't take it for granted - **larger apps generally have greater concerns that need to be factored in**. I'm going to discuss shortly why spending a little more time planning out the structure to your application is worth it in the long run.

Most JavaScript developers likely use a mixed combination of the following for their current architecture:

- custom widgets
- models
- views
- controllers
- templates
- libraries/toolkits
- an application core.

> **Related Reading**
>
> Rebecca Murphey - Structuring JavaScript Applications
>
> Peter Michaux - MVC Architecture For JavaScript Applications
>
> StackOverflow - A discussion on modern MVC frameworks
>
> Doug Neiner - Stateful Plugins and the Widget Factory

You probably also break down your application's functionality into blocks of modules or apply other patterns for this. This is great, but there are a number of potential problems you can run into if this represents all of your application's structure.

1. How much of this architecture is instantly re-usable?

Can single modules exist on their own independently? Are they self-contained? Right now if I were to look at the codebase for a large application you or your team were working on and selected a random module, would it be possible for me to easily just drop it into a new page and start using it on its own?. You may question the rationale behind wanting to do this, however I encourage you to think about the future. What if your company were to begin building more and more non-trivial applications which shared some cross-over in functionality?. If someone said, 'Our users love using the chat module in our mail client. Let's drop that into our new collaborative editing suite', would this be possible without significantly altering the code?.

2. How much do modules depend on other modules in the system?

Are they tightly coupled? Before I dig into why this is a concern, I should note that I understand it's not always possible to have modules with absolutely no other dependencies in a system. At a granular level you may well have modules that extend the base functionality of others, but this question is more-so related to groups of modules with distinct functionality. It should be possible for all of these distinct sets of modules to work in your application without depending on too many other modules being present or loaded in order to function.

3. If specific parts of your application fail, can it still function?

If you're building a GMail-like application and your webmail module (or modules) fail, this shouldn't block the rest of the UI or prevent users from being able to use other parts of the page such as chat. At the same time, as per before, modules should ideally be able to exist on their own outside of your current application architecture. In my talks I mention dynamic dependency (or module) loading based on expressed user-intent as something related. For example, in GMail's case they might have the chat module collapsed by default without the core module code loaded on page initialization. If a user expressed an intent to use the chat feature, only then would it be dynamically loaded. Ideally, you want this to be possible without it negatively affecting the rest of your application.

4. How easily can you test individual modules?

When working on systems of significant scale where there's a potential for millions of users to use (or mis-use) the different parts it, it's essential that modules which may end up being re-used across a number of different applications be sufficiently tested. Testing needs to be possible for when the module both inside and outside of the architecture for which it was initially built. In my view, this provides the most assurance that it shouldn't break if dropped into another system.

Think Long Term

When devising the architecture for your large application, it's important to think ahead. Not just a month or a year from now, but beyond that. What might change? It's of course impossible to guess exactly how your application may grow, but there's certainly room to consider what is likely. Here, there is at least one specific aspect of your application that comes to mind.

Developers often couple their DOM manipulation code quite tightly with the rest of their application - even when they've gone to the trouble of separating their core logic down into modules. Think about it..why is this not a good idea if we're thinking long-term?

One member of my audience suggested that it was because a rigid architecture defined in the present may not be suitable for the future. Whilst certainly true, there's another concern that may cost even more if not factored in.

> You may well decide to **switch** from using Dojo, jQuery, Zepto or YUI to something entirely different for reasons of performance, security or design in the future. This can become a problem because libraries are not easily interchangeable and have high switching costs if tightly coupled to your app.

If you're a Dojo developer (like some of the audience at my talk), you may not have something better to switch to in the present, but who is to say that in 2-3 years something better doesn't come out that you'll want to switch to?.

This is a relatively trivial decision in smaller codebases but for larger applications, having an architecture which is flexible enough to support **not** caring about the libraries being used in your modules can be of great benefit, both financially and from a time-saving perspective.

To summarize, if you reviewed your architecture right now, could a decision to switch libraries be made without rewriting your entire application?. If not, consider reading on because I think the architecture being outlined today may be of interest.

There are a number of influential JavaScript developers who have previously outlined some of the concerns I've touched upon so far. Three key quotes I would like to share from them are the following:

> "The secret to building large apps is never build large apps. Break your applications into small pieces. Then, assemble those testable, bite-sized pieces into your big application" -
> **Justin Meyer, author JavaScriptMVC**

> "The key is to acknowledge from the start that you have no idea how this will grow. When you accept that you don't know everything, you begin to design the system defensively. You identify the key areas that may change, which often is very easy when you put a little bit of time into it. For instance, you should expect that any part of the app that communicates with another system will likely change, so you need to abstract that away." -
> **Nicholas Zakas, author 'High-performance JavaScript websites'**

and last but not least:

> "The more tied components are to each other, the less reusable they will be, and the more difficult it becomes to make changes to one without accidentally affecting another" -
> **Rebecca Murphey, author of jQuery Fundamentals.**

These principles are essential to building an architecture that can stand the test of time and should always be kept in mind.

Brainstorming

Let's think about what we're trying to achieve for a moment.

> We want a loosely coupled architecture with functionality broken down into **independent modules** with ideally no inter-module dependencies. Modules **speak** to the rest of the application when something interesting happens and an **intermediate layer** interprets and reacts to these messages.

For example, if we had a JavaScript application responsible for an online bakery, one such 'interesting' message from a module might be 'batch 42 of bread rolls is ready for dispatch'.

We use a different layer to interpret messages from modules so that a) modules don't directly access the core and b) modules don't need to directly call or interact with other modules. This helps prevent applications from falling over due to errors with specific modules and provides us a way to kick-start modules which have fallen over.

Another concern is security. The reality is that most of us don't consider internal application security as that much of a concern. We tell ourselves that as we're structuring the application, we're intelligent enough to figure out what should be publicly or privately accessible.

However, wouldn't it help if you had a way to determine what a module was permitted to do in the system? eg. if I know I've limited the permissions in my system to not allow a public chat widget to interface with an admin module or a module with DB-write permissions, I can limit the chances of someone exploiting vulnerabilities I have yet to find in the widget to pass some XSS in there. Modules shouldn't be able to access everything. They probably can in most current architectures, but do they really need to be able to?

Having an intermediate layer handle permissions for which modules can access which parts of your framework gives you added security. This means a module is only able to do at most what we've permitted it do.

The Proposed Architecture

The solution to the architecture we seek to define is a combination of three well-known design patterns: the **module**, **facade** and **mediator**.

Rather than the traditional model of modules directly communicating with each other, in this decoupled architecture, they'll instead only publish events of interest (ideally, without a knowledge of other modules in the system). The mediator pattern will be used to both subscribe to messages from these modules and handle what the appropriate response to notifications should be. The facade pattern will be used to enforce module permissions.

I will be going into more detail on each of these patterns below:

Module Theory

You probably already use some variation of modules in your existing architecture. If however, you don't, this section will present a short primer on them.

> Modules are an **integral** piece of any robust application's architecture and are typically single-purpose parts of a larger system that are interchangeable.

Depending on how you implement modules, it's possible to define dependencies for them which can be automatically loaded to bring together all of the other parts instantly. This is considered more scalable than having to track the various dependencies for them and manually load modules or inject script tags.

Any significantly non-trivial application should be built from modular components. Going back to GMail, you could consider modules independent units of functionality that can exist on their own, so the chat

feature for example. It's probably backed by a chat module, however, depending on how complex that unit of functionality is, it may well have more granular sub-modules that it depends on. For example, one could have a module simply to handle the use of emoticons which can be shared across both chat and mail composition parts of the system.

> In the architecture being discussed, modules have a **very limited knowledge** of what's going on in the rest of the system. Instead, we delegate this responsibility to a mediator via a facade.

This is by design because if a module only cares about letting the system know when something of interest happens without worrying if other modules are running, a system is capable of supporting adding, removing or replacing modules without the rest of the modules in the system falling over due to tight coupling.

Loose coupling is thus essential to this idea being possible. It facilitates easier maintainability of modules by removing code dependencies where possible. In our case, modules should not rely on other modules in order to function correctly. When loose coupling is implemented effectively, its straight-forward to see how changes to one part of a system may affect another.

In JavaScript, there are several options for implementing modules including the well-known module pattern and object literals. Experienced developers will already be familiar with these and if so, please skip ahead to the section on CommonJS modules.

The Module Pattern

The module pattern is a popular design that pattern that encapsulates 'privacy', state and organization using closures. It provides a way of wrapping a mix of public and private methods and variables, protecting pieces from leaking into the global scope and accidentally colliding with another developer's interface. With this pattern, only a public API is returned, keeping everything else within the closure private.

This provides a clean solution for shielding logic doing the heavy lifting whilst only exposing an interface you wish other parts of your application to use. The pattern is quite similar to an immediately-invoked functional expression (IIFE) except that an object is returned rather than a function.

It should be noted that there isn't really a true sense of 'privacy' inside JavaScript because unlike some traditional languages, it doesn't have access modifiers. Variables can't technically be declared as being public nor private and so we use function scope to simulate this concept. Within the module pattern, variables or methods declared are only available inside the module itself thanks to closure. Variables or methods defined within the returning object however are available to everyone.

Below you can see an example of a shopping basket implemented using the pattern. The module itself is completely self-contained in a global object called `basketModule`. The `basket` array in the module is kept private and so other parts of your application are unable to directly read it. It only exists with the module's closure and so the only methods able to access it are those with access to its scope (ie. `addItem()`, `getItem()` etc).

```
var basketModule = (function() {
    var basket = []; //private
    return { //exposed to public
        addItem: function(values) {
            basket.push(values);
        },
        getItemCount: function() {
            return basket.length;
```

```
        },
        getTotal: function(){
           var q = this.getItemCount(),p=0;
            while(q--){
                p+= basket[q].price;
            }
            return p;
        }
    }
}());
```

Inside the module, you'll notice we return an `object`. This gets automatically assigned to `basketModule` so that you can interact with it as follows:

```
//basketModule is an object with properties which can also be methods
basketModule.addItem({item:'bread',price:0.5});
basketModule.addItem({item:'butter',price:0.3});

console.log(basketModule.getItemCount());
console.log(basketModule.getTotal());

//however, the following will not work:
console.log(basketModule.basket);// (undefined as not inside the returned object)
console.log(basket); //(only exists within the scope of the closure)
```

The methods above are effectively namespaced inside `basketModule`.

From a historical perspective, the module pattern was originally developed by a number of people including Richard Cornford in 2003. It was later popularized by Douglas Crockford in his lectures and re-introduced by Eric Miraglia on the YUI blog.

How about the module pattern in specific toolkits or frameworks?

Dojo

Dojo attempts to provide 'class'-like functionality through `dojo.declare`, which can be used for amongst other things, creating implementations of the module pattern. For example, if we wanted to declare `basket` as a module of the `store` namespace, this could be achieved as follows:

```
//traditional way
var store = window.store || {};
store.basket = store.basket || {};

//using dojo.setObject
dojo.setObject("store.basket.object", (function() {
    var basket = [];
    function privateMethod() {
        console.log(basket);
    }
    return {
```

```
        publicMethod: function(){
                privateMethod();
        }
    };
}()));
```

which can become quite powerful when used with `dojo.provide` and mixins.

YUI

The following example is heavily based on the original YUI module pattern implementation by Eric Miraglia, but is relatively self-explanatory.

```
YAHOO.store.basket = function () {

    //"private" variables:
    var myPrivateVar = "I can be accessed only within YAHOO.store.basket
.";

    //"private" method:
    var myPrivateMethod = function () {
            YAHOO.log("I can be accessed only from within
YAHOO.store.basket");
        }

    return {
        myPublicProperty: "I'm a public property.",
        myPublicMethod: function () {
            YAHOO.log("I'm a public method.");

            //Within basket, I can access "private" vars and methods:
            YAHOO.log(myPrivateVar);
            YAHOO.log(myPrivateMethod());

            //The native scope of myPublicMethod is store so we can
            //access public members using "this":
            YAHOO.log(this.myPublicProperty);
        }
    };

}();
```

jQuery

There are a number of ways in which jQuery code unspecific to plugins can be wrapped inside the module pattern. Ben Cherry previously suggested an implementation where a function wrapper is used around module definitions in the event of there being a number of commonalities between modules.

In the following example, a `library` function is defined which declares a new library and automatically binds up the `init` function to `document.ready` when new libraries (ie. modules) are created.

```
function library(module) {
  $(function() {
    if (module.init) {
      module.init();
    }
  });
  return module;
}

var myLibrary = library(function() {
   return {
     init: function() {
       /*implementation*/
     }
   };
}());
```

Related Reading

Ben Cherry - The Module Pattern In-Depth
John Hann - The Future Is Modules, Not Frameworks
Nathan Smith - A Module pattern aliased window and document gist
David Litmark - An Introduction To The Revealing Module Pattern

Object Literal Notation

In object literal notation, an object is described as a set of comma-separated name/value pairs enclosured in curly braces (`{}`). Names inside the object may be either strings or identifiers that are followed by a colon. There should be no comma used after the final name/value pair in the object as this may result in errors.

Object literals don't require instantiation using the `new` operator but shouldn't be used at the start of a statement as the opening `{` may be interpreted as the beginning of a block. Below you can see an example of a module defined using object literal syntax. New members may be added to the object using assignment as follows `myModule.property = 'someValue';`

Whilst the module pattern is useful for many things, if you find yourself not requiring specific properties or methods to be private, the object literal is a more than suitable alternative.

```
var myModule = {
    myProperty : 'someValue',
    //object literals can contain properties and methods.
    //here, another object is defined for configuration
    //purposes:
    myConfig:{
        useCaching:true,
        language: 'en'
    },
```

```
    //a very basic method
    myMethod: function(){
        console.log('I can haz functionality?');
    },
    //output a value based on current configuration
    myMethod2: function(){
        console.log('Caching is:' +
(this.myConfig.useCaching)?'enabled':'disabled');
    },
    //override the current configuration
    myMethod3: function(newConfig){
        if(typeof newConfig == 'object'){
           this.myConfig = newConfig;
           console.log(this.myConfig.language);
        }
    }
};

myModule.myMethod(); //I can haz functionality
myModule.myMethod2(); //outputs enabled
myModule.myMethod3({language:'fr',useCaching:false}); //fr
```

Related Reading

Rebecca Murphey - Using Objects To Organize Your Code
Stoyan Stefanov - 3 Ways To Define A JavaScript Class
Ben Alman - Clarifications On Object Literals (There's no such thing as a JSON Object)
John Resig - Simple JavaScript Inheritance

CommonJS Modules

Over the last year or two, you may have heard about CommonJS - a volunteer working group which designs, prototypes and standardizes JavaScript APIs. To date they've ratified standards for modules and packages.The CommonJS AMD proposal specifies a simple API for declaring modules which can be used with both synchronous and asynchronous script tag loading in the browser. Their module pattern is relatively clean and I consider it a reliable stepping stone to the module system proposed for ES Harmony (the next release of the JavaScript language).

From a structure perspective, a CommonJS module is a reusable piece of JavaScript which exports specific objects made available to any dependent code. This module format is becoming quite ubiquitous as a standard module format for JS. There are plenty of great tutorials on implementing CommonJS modules, but at a high-level they basically contain two primary parts: an `exports` object that contains the objects a module wishes to make available to other modules and a `require` function that modules can use to import the exports of other modules.

```
/*
Example of achieving compatibility with AMD and standard CommonJS by
putting boilerplate around the standard CommonJS module format:
*/

```

```
(function(define){
define(function(require,exports){
// module contents
 var dep1 = require("dep1");
 exports.someExportedFunction = function(){...};
 //...
});
})(typeof define=="function"?define:function(factory)
{factory(require,exports)});
```

There are a number of great JavaScript libraries for handling module loading in the **CommonJS** module format, but my personal preference is RequireJS. A complete tutorial on RequireJS is outside the scope of this tutorial, but I can recommend reading James Burke's ScriptJunkie post on it here. I know a number of people that also like Yabble.

Out of the box, RequireJS provides methods for easing how we create static modules with wrappers and it's extremely easy to craft modules with support for asynchronous loading. It can easily load modules and their dependencies this way and execute the body of the module once available.

There are some developers that however claim CommonJS modules aren't suitable enough for the browser. The reason cited is that they can't be loaded via a script tag without some level of server-side assistance. We can imagine having a library for encoding images as ASCII art which might export a `encodeToASCII` function. A module from this could resemble:

```
var encodeToASCII = require("encoder").encodeToASCII;
exports.encodeSomeSource = function(){
    //process then call encodeToASCII
}
```

This type of scenario wouldn't work with a script tag because the scope isn't wrapped, meaning our `encodeToASCII` method would be attached to the `window`, `require` wouldn't be as such defined and exports would need to be created for each module separately. A client-side library together with server-side assistance or a library loading the script with an XHR request using `eval()` could however handle this easily.

Using RequireJS, the module from earlier could be rewritten as follows:

```
define(function(require, exports, module) {
    var encodeToASCII = require("encoder").encodeToASCII;
    exports.encodeSomeSource = function(){
            //process then call encodeToASCII
    }
});
```

For developers who may not rely on just using static JavaScript for their projects, CommonJS modules are an excellent path to go down, but do spend some time reading up on it. I've really only covered the tip of the ice berg but both the CommonJS wiki and Sitepen have a number of resources if you wish to read further.

Related Reading

The CommonJS Module Specifications
Alex Young - Demystifying CommonJS Modules
Notes on CommonJS modules with RequireJS

The Facade Pattern

Next, we're going to look at the facade pattern, a design pattern which plays a critical role in the architecture being defined today.

When you put up a facade, you're usually creating an outward appearance which conceals a different reality. The facade pattern provides a convenient **higher-level interface** to a larger body of code, hiding its true underlying complexity. Think of it as simplifying the API being presented to other developers.

Facades are a **structural pattern** which can often be seen in JavaScript libraries and frameworks where, although an implementation may support methods with a wide range of behaviors, only a 'facade' or limited abstract of these methods is presented to the client for use.

This allows us to interact with the facade rather than the subsystem behind the scenes.

The reason the facade is of interest is because of its ability to hide implementation-specific details about a body of functionality contained in individual modules. The implementation of a module can change without the clients really even knowing about it.

By maintaining a consistent facade (simplified API), the worry about whether a module extensively uses dojo, jQuery, YUI, zepto or something else becomes significantly less important. As long as the interaction layer doesn't change, you retain the ability to switch out libraries (eg. jQuery for Dojo) at a later point without affecting the rest of the system.

Below is a very basic example of a facade in action. As you can see, our module contains a number of methods which have been privately defined. A facade is then used to supply a much simpler API to accessing these methods:

```
var module = (function() {
    var _private = {
        i:5,
        get : function() {
            console.log('current value:' + this.i);
        },
        set : function( val ) {
            this.i = val;
        },
        run : function() {
            console.log('running');
        },
        jump: function(){
            console.log('jumping');
        }
    };
    return {
        facade : function( args ) {
            _private.set(args.val);
            _private.get();
            if ( args.run ) {
```

```
                _private.run();
            }
        }
    }
}());


module.facade({run: true, val:10});
//outputs current value: 10, running

```

and that's really it for the facade before we apply it to our architecture. Next, we'll be diving into the exciting mediator pattern. The core difference between the facade pattern and the mediator is that the facade (a structural pattern) only exposes existing functionality whilst the mediator (a behavioral pattern) can add functionality.

Related Reading

Dustin Diaz, Ross Harmes - Pro JavaScript Design Patterns (Chapter 10, available to read on Google Books)

The Mediator Pattern

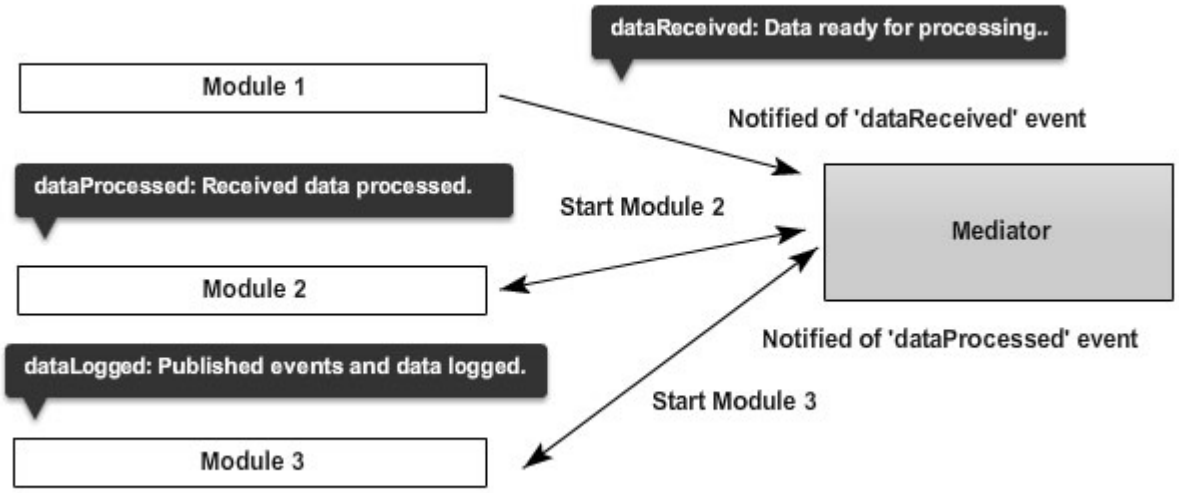
The mediator pattern is best introduced with a simple analogy - think of your typical airport traffic control. The tower handles what planes can take off and land because all communications are done from the planes to the control tower, rather than from plane-to-plane. A centralized controller is key to the success of this system and that's really what a mediator is.

Mediators are used when the communication between modules may be complex, but is still **well defined**. If it appears a system may have too many relationships between modules in your code, it may be time to have a central point of control, which is where the pattern fits in.

In real-world terms, a mediator **encapsulates** how disparate modules **interact** with each other by acting as an intermediary. The pattern also promotes loose coupling by preventing objects from referring to each other explicitly - in our system, this helps to solve our module inter-dependency issues.

What other advantages does it have to offer? Well, mediators allow for actions of each module to vary independently, so it’s extremely flexible. If you've previously used the Observer (Pub/Sub) pattern to implement an event broadcast system between the modules in your system, you'll find mediators relatively easy to understand.

Let's take a look at a high level view of how modules might interact with a mediator:



Consider modules as publishers and the mediator as both a publisher and subscriber. Module 1 broadcasts an event notifying the mediator something needs to done. The mediator captures this message and 'starts' the modules needed to complete this task Module 2 performs the task that Module 1 requires and broadcasts a completion event back to the mediator. In the mean time, Module 3 has also been started by the mediator and is logging results of any notifications passed back from the mediator.

Notice how at no point do any of the modules **directly communicate** with one another. If Module 3 in the chain were to simply fail or stop functioning, the mediator could hypothetically 'pause' the tasks on the other modules, stop and restart Module 3 and then continue working with little to no impact on the system. This level of decoupling is one of the main strengths the pattern has to offer.

To review, the advantages of the mediator are that:

It decouples modules by introducing an intermediary as a central point of control.It allows modules to broadcast or listen for messages without being concerned with the rest of the system. Messages can be handled by any number of modules at once.

It is typically significantly more easy to add or remove features to systems which are loosely coupled like this.

And its disadvantages:

By adding a mediator between modules, they must always communicate indirectly. This can cause a very minor performance drop - because of the nature of loose coupling, its difficult to establish how a system might react by only looking at the broadcasts. At the end of the day, tight coupling causes all kinds of headaches and this is one solution.

Example: This is a possible implementation of the mediator pattern based on previous work by @rpflorence

```
var mediator = (function(){
    var subscribe = function(channel, fn){
        if (!mediator.channels[channel]) mediator.channels[channel] = [];
        mediator.channels[channel].push({ context: this, callback: fn });
        return this;
    },

    publish = function(channel){
        if (!mediator.channels[channel]) return false;
```

```
        var args = Array.prototype.slice.call(arguments, 1);
        for (var i = 0, l = mediator.channels[channel].length; i < l; i++)
{
            var subscription = mediator.channels[channel][i];
            subscription.callback.apply(subscription.context, args);
        }
        return this;
    };

    return {
        channels: {},
        publish: publish,
        subscribe: subscribe,
        installTo: function(obj){
            obj.subscribe = subscribe;
            obj.publish = publish;
        }
    };

}());
```

Example: Here are two sample uses of the implementation from above. It's effectively managed publish/subscribe:

```
//Pub/sub on a centralized mediator

mediator.name = "tim";
mediator.subscribe('nameChange', function(arg){
        console.log(this.name);
        this.name = arg;
        console.log(this.name);
});

mediator.publish('nameChange', 'david'); //tim, david


//Pub/sub via third party mediator

var obj = { name: 'sam' };
mediator.installTo(obj);
obj.subscribe('nameChange', function(arg){
        console.log(this.name);
        this.name = arg;
        console.log(this.name);
});

obj.publish('nameChange', 'john'); //sam, john
```

Related Reading

Stoyan Stefanov - Page 168, JavaScript Patterns
HB Stone - JavaScript Design Patterns: Mediator
Vince Huston - The Mediator Pattern (not specific to JavaScript, but a concise)

Applying The Facade: Abstraction Of The Core

In the architecture suggested:

> A facade serves as an **abstraction** of the application core which sits between the mediator and our modules - it should ideally be the **only** other part of the system modules are aware of.

The responsibilities of the abstraction include ensuring a **consistent interface** to these modules is available at all times. This closely resembles the role of the **sandbox controller** in the excellent architecture first suggested by Nicholas Zakas.

Components are going to communicate with the mediator through the facade so it needs to be **dependable**. When I say 'communicate', I should clarify that as the facade is an abstraction of the mediator which will be listening out for broadcasts from modules that will be relayed back to the mediator.

In addition to providing an interface to modules, the facade also acts as a security guard, determining which parts of the application a module may access. Components only call **their own** methods and shouldn't be able to interface with anything they don't have permission to. For example, a module may broadcast `dataValidationCompletedWriteToDB`. The idea of a security check here is to ensure that the module has permissions to request database-write access. What we ideally want to avoid are issues with modules accidentally trying to do something they shouldn't be.

To review in short, the mediator remains a type of pub/sub manager but is only passed interesting messages once they've cleared permission checks by the facade.

Applying the Mediator: The Application Core

The mediator plays the role of the application core. We've briefly touched on some of its responsibilities but lets clarify what they are in full.

The core's primary job is to manage the module **lifecycle.** When the core detects an **interesting message** it needs to decide how the application should react - this effectively means deciding whether a module or set of modules needs to be **started** or **stopped**.

> Once a module has been started, it should ideally execute **automatically**. It's not the core's task to decide whether this should be when the DOM is ready and there's enough scope in the architecture for modules to make such decisions on their own.

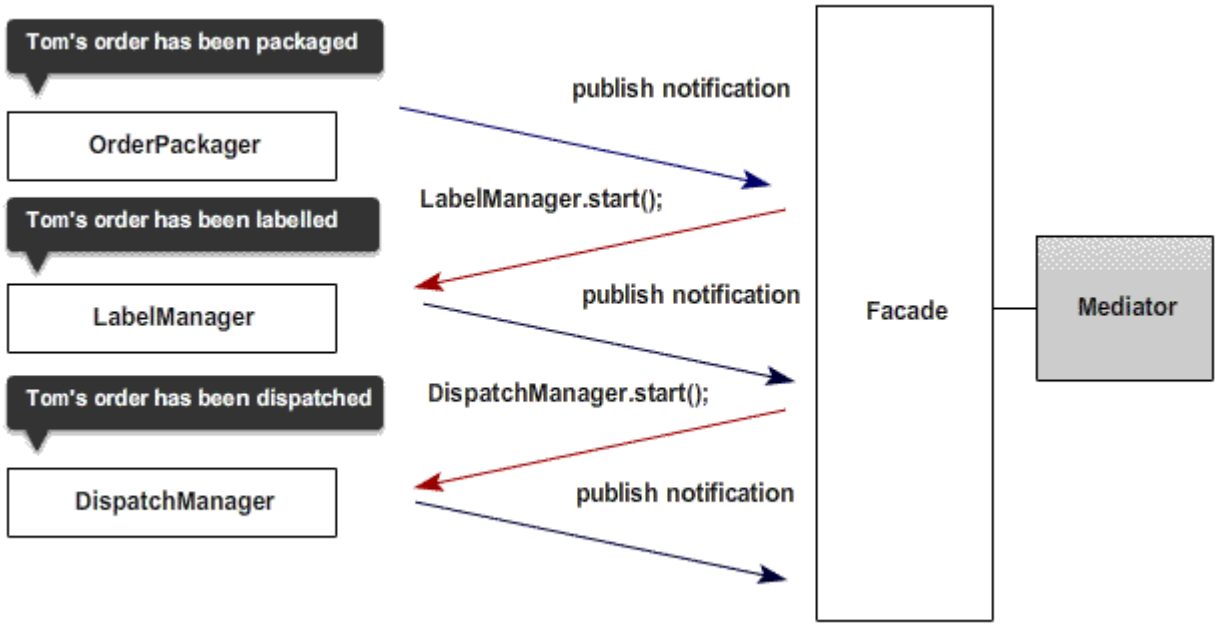
You may be wondering in what circumstance a module might need to be 'stopped' - if the application detects that a particular module has failed or is experiencing significant errors, a decision can be made to prevent methods in that module from executing further so that it may be restarted. The goal here is to assist in reducing disruption to the user experience.

In addition, the core should enable **adding or removing** modules without breaking anything. A typical example of where this may be the case is functionality which may not be available on initial page load, but is dynamically loaded based on expressed user-intent eg. going back to our GMail example, Google could keep the chat widget collapsed by default and only dynamically load in the chat module(s) when a user expresses an interest in using that part of the application. From a performance optimization perspective, this may make sense.

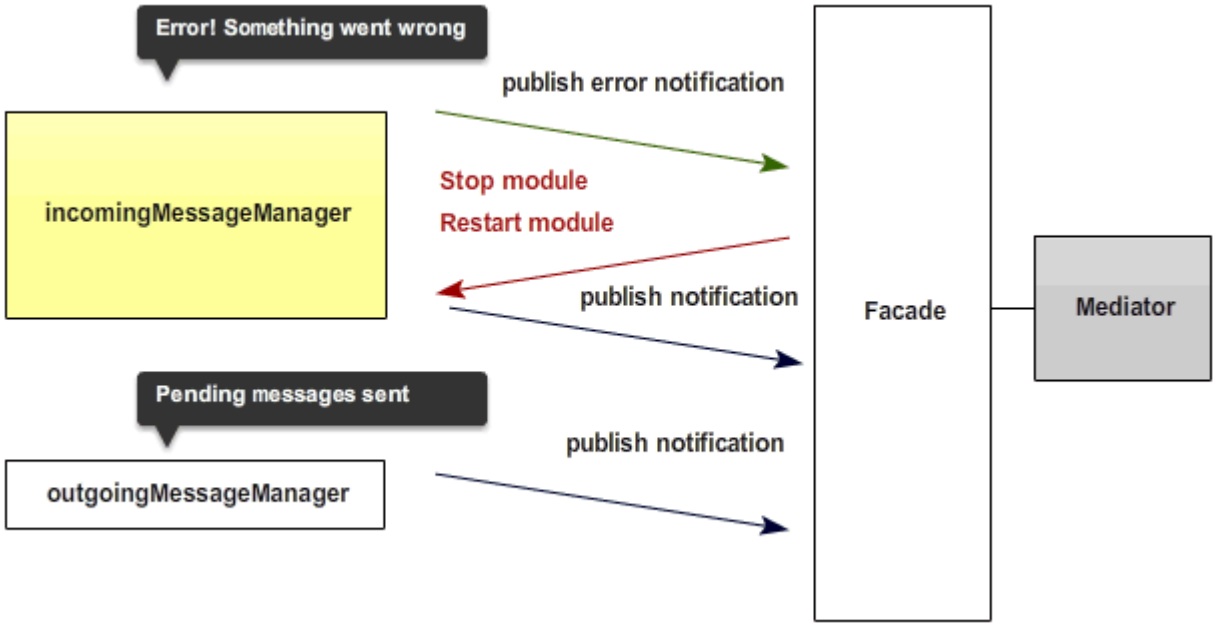
Error management will also be handled by the application core. In addition to modules broadcasting messages of interest they will also broadcast any errors experienced which the core can then react to accordingly (eg. stopping modules, restarting them etc).It's important that as part of a decoupled architecture there to be enough scope for the introduction of new or better ways of handling or displaying errors to the end user without manually having to change each module. Using publish/subscribe through a mediator allows us to achieve this.

Tying It All Together

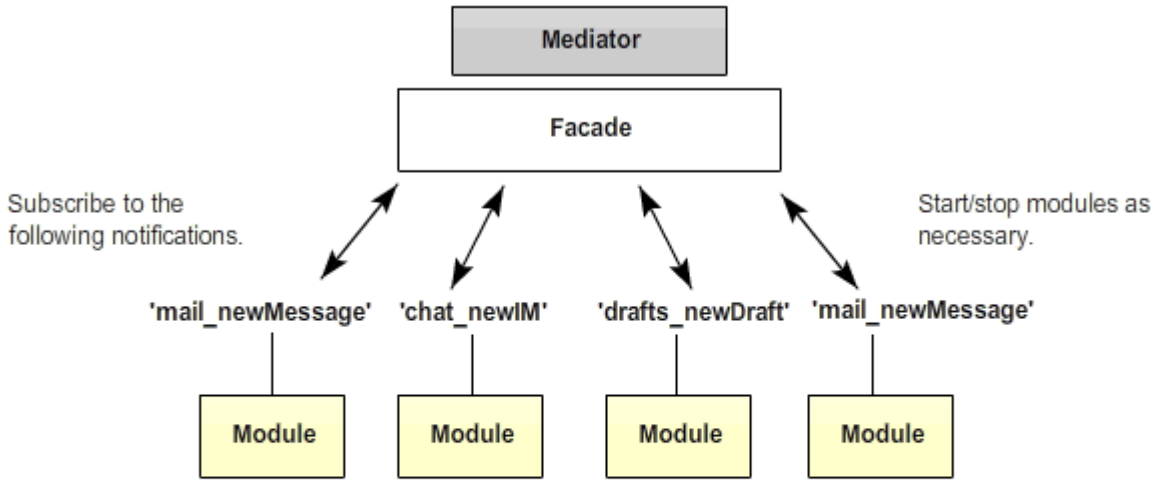
- **Modules** contain specific pieces of functionality for your application. They publish notifications informing the application whenever something interesting happens - this is their primary concern. As I'll cover in the FAQs, modules can depend on DOM utility methods, but ideally shouldn't depend on any other modules in the system. They should not be concerned with:
  - what objects or modules are subscribing to the messages they publish
  - where these objects are based (whether this is on the client or server)
  - how many objects subscribe to notifications



- **The Facade** abstracts the core to avoid modules touching it directly. It subscribes to interesting events (from modules) and says 'Great! What happened? Give me the details!'. It also handles module security by checking to ensure the module broadcasting an event has the necessary permissions to pass such events that can be accepted.



- **The Mediator (Application Core)** acts as a 'Pub/Sub' manager using the mediator pattern. It's responsible for module management and starts/stops modules as needed. This is of particular use for dynamic dependency loading and ensuring modules which fail can be centrally restarted as needed.



As long as modules publish a **consistent** set of notifications, the underlying libraries used within these modules become less important. A module using Dojo that publishes notifications will be treated the same within the system as one which uses jQuery or YUI. This allows a switch later-on with less impact to the rest of the application.

The result of this architecture is that modules (in most cases) are theoretically no longer dependent on other modules. They can be easily tested and maintained on their own and because of the level of decoupling applied, modules can be picked up and dropped into a new page for use in another project without significant additional effort. They can also be dynamically added or removed without the application falling over.

Beyond Pub/Sub: Automatic Event Registration

As previously mentioned by Michael Mahemoff, when thinking about large-scale JavaScript, it can be of benefit to exploit some of the more dynamic features of the language. You can read more about some of the concerns highlighted on Michael's G+ page, but I would like to focus on one specifically - automatic event registration (AER).

AER solves the problem of wiring up subscribers to publishers by introducing a pattern which auto-wires based on naming conventions. For example, if a module publishes an event called `messageUpdate`, anything with a `messageUpdate` method would be automatically called.

The setup for this pattern involves registering all components which might subscribe to events, registering all events that may be subscribed to and finally for each subscription method in your component-set, binding the event to it. It's a very interesting approach which is related to the architecture presented in this post, but does come with some interesting challenges.

For example, when working dynamically, objects may be required to register themselves upon creation. Please feel free to check out Michael's post on AER as he discusses how to handle such issues in more depth.

Frequently Asked Questions

Q: Is it possible to avoid implementing a sandbox or facade altogether?

A: Although the architecture outlined uses a facade to implement security features, it's entirely possible to get by using a mediator and pub/sub to communicate events of interest throughout the application without it. This lighter version would offer a similar level of decoupling, but ensure you're comfortable with modules directly touching the application core (mediator) if opting for this variation.

Q: You've mentioned modules not having any dependencies. Does this include dependencies such as third party libraries (eg. jQuery?)

A: I'm specifically referring to dependencies on other modules here. What some developers opting for an architecture such as this opt for is actually abstracting utilities common to DOM libraries -eg. one could have a DOM utility class for query selectors which when used returns the result of querying the DOM using jQuery (or, if you switched it out at a later point, Dojo). This way, although modules still query the DOM, they aren't directly using hardcoded functions from any particular library or toolkit. There's quite a lot of variation in how this might be achieved, but the takeaway is that ideally core modules shouldn't depend on other modules if opting for this architecture.

You'll find that when this is the case it can sometimes be more easy to get a complete module from one project working in another with little extra effort. I should make it clear that I fully agree that it can sometimes be significantly more sensible for modules to extend or use other modules for part of their functionality, however bear in mind that this can in some cases increase the effort required to make such modules 'liftable' for other projects.

Q: I'd like to start using this architecture today. Is there any boilerplate code around I can work from?

A: I plan on releasing a free boilerplate pack for this post when time permits, but at the moment, your best bet is probably the 'Writing Modular JavaScript' premium tutorial by Andrew Burgees (for complete disclosure, this is a referral link as any credits received are re-invested into reviewing material before I recommend it to others). Andrew's pack includes a screencast and code and covers most of the main concepts outlined in this post but opts for calling the facade a 'sandbox', as per Zakas. There's some discussion regarding just how DOM library abstraction should be ideally implemented in such an architecture - similar to my answer for the second question, Andrew opts for some interesting patterns

on generalizing query selectors so that at most, switching libraries is a change that can be made in a few short lines. I'm not saying this is the right or best way to go about this, but it's an approach I personally also use.

Q: If the modules need to directly communicate with the core, is this possible?

A: As Zakas has previously hinted, there's technically no reason why modules shouldn't be able to access the core but this is more of a best practice than anything. If you want to strictly stick to this architecture you'll need to follow the rules defined or opt for a looser architecture as per the answer to the first question.

Credits

Thanks to Nicholas Zakas for his original work in bringing together many of the concepts presented today; Andrée Hansson for his kind offer to do a technical review of the post (as well as his feedback that helped improve it); Rebecca Murphey, Justin Meyer, John Hann, Peter Michaux, Paul Irish and Alex Sexton, all of whom have written material related to the topics discussed in the past and are a constant source of inspiration for both myself and others.