JavaScript Application Frameworks

the parts

Garann Means | @garannm
in applications and frameworks, the code is the easy part

the hard part is decision making
and it is hard to build an app

- figuring out what it should do
- translating requirements to workflows
- making it all work together
- (even when you have good tools)
any framework may need customization

- a framework is a basis
- your app may have slightly different needs
- different ways of handling the same concept may exist within the same app
- it may have no opinion on other concepts
compare to jQuery:

- tons of functionality
- some core, some more esoteric
- discovering new features leads to better ways of doing things
- the most discoverable features are those we know should be there
using a tool well means understanding the decisions it makes, and making good decisions about how to use it
what should be there
and what we should expect from it
we expect framework methods to live on a prototype

- unlike a library, whose methods should usually not require an instance
- allows creation of objects set up to work together
- instantiation lets us keep related functionality together
but we also need methods for the app containing the objects

- global initialization
- event broadcasting
- global state changes
- stateless utilities
so it makes sense to start with a global

```javascript
(function(Frmwrk) {

    function _init() {
    
    }

    Frmwrk.test = function() {
        console.log("yup");
    };

    _init();

})(window.Frmwrk = {});

Frmwrk.test(); // yup
```
so that’s nice.
but it should do something
we were going to instantiate some objects

- state
- data
- or both
- we don't want to have to create too many types
one object we know we need

function _init() {
    Frmwrk.app = new Frmwrk.Object();
}
so let's make that the default

Frmwrk.Object = function ( obj ) {
    $.extend( true, this, {
        sel: "body"
    }, obj );
    if ( this.sel ) {
        this.el = $( this.sel );
    }
    return this;
};
hold up..
whose $ is that?
utility libraries

- it's also good to understand libraries
- but this is not a talk on libraries
- make a smart choice providing the most you need but no more
- jQuery, Underscore, Zepto, etc.
- focus on connecting pieces, not writing essential tools
including third-party libraries or tools

- can be appended to your core file
- other users don't need to import the additional script
- difficult to change or upgrade
- may conflict with other versions in use
leaving it up to implementers

- other users need to import the script separately, in order
- framework should work with all supported versions
- may make sense to allow uses of library to be overridden
a note about dependencies
in this example, jQuery is used like so:

```javascript
( function( Frmwrk, $ ) {
    ...

})( window.Frmwrk = {}, window.jQuery );
```
for a more modular approach, you might use require.js

define( [ "jQuery" ], function( $ ) {
    ...
});
let’s get back to our base object

Frmwrk.Object = function (obj) {
    $.extend(true, this, {
        sel: "body"
    }, obj);
    if (this.sel) {
        this.el = $(this.sel);
    }

    return this;
};
what do we expect from a data object?

- a place to store data
- the ability to save and update itself
- notifications if it changes
setting and changing data

- ideally, set on object creation
- but may need to be set on an empty object (like the app)
- should be possible to update one or more properties
the data should be its own object

Frmwrk.Object = function (obj) {
    $.extend(true, this, {
        sel: "body",
        data: {}
    }, obj);
    if (this.sel) {
        this.el = $(this.sel);
    }

    return this;
};
we want control over how that's updated

```
Frmwrk.Object.prototype.set = _set = function( data, val ) {
    if ( typeof data == "string" ) {
        this.data[ data ] = val;
    } else {
        for ( key in data ) {
            this.data[ key ] = data[ key ];
        }
    }
    return this;
};
```
and a shortcut for the app itself

Frmwrk.set = function() {
    _set.apply( Frmwrk.app, arguments );
};
why not just have the implementer modify `[obj].data` directly?

- this is a cheap workaround to observing the objects
- can fire off events, update server
- allows for a predictable pattern
- allows us to do a little magic
the downside is it means we also need a way to get data

```javascript
Frmwrk.Object.prototype.get = _get = function( key ) {
    return this.data[ key ];
};
Frmwrk.get = function() {
    return _get.apply( Frmwrk.app, arguments );
};
```
we could also condense the two, jQuery-style

Frmwrk.Object.prototype.data = _data = function( data, val ) {
    if ( arguments.length == 1 && typeof data == "string" ) {
        return this.data[ key ];
    } else {
        ...
    }
};
once we have data, we have something to store

- data persistence tends to be boring to write
- we need several types of XHRs
- we want saving and updating to be as invisible as possible
should we assume REST?

- depends very much on your backend
- may be more JavaScripty to do CRUD from all endpoints:
  - data with no ID: create
  - ID only: read
  - data and ID: update
  - ID and string or number: delete
- but only if that fits the larger architecture
let's assume it does

Frmwrk.Object = function ( obj ) {
    $.extend( true, this, {
        sel: "body",
        data: {},
        endpoint: null
    }, obj );
    if ( this.sel ) {
        this.el = $( this.sel );
    }
    return this;
};
we want to sync whenever an update happens

Frmwrk.Object.prototype.set = _set = function( data, val ) {
    if ( typeof data == "string" ) {
        this.data[ data ] = val;
    } else {
        for ( key in data ) {
            this.data[ key ] = data[ key ];
        }
    }
    if ( this.endpoint ) {
        this.sync();
    }
    return this;
};
additionally, we want explicit methods to delete or read an object
please use your imagination for those
(remember that thing about CRUD being boring?)
then, of course, we need a function to do the work

```javascript
Frmwrk.Object.prototype.sync = function( data, cmd ) {
    var id = typeof data === "number" ? data : undefined,
        that = this;
    $.post( this.endpoint, {
        id: data.id || id,
        command: cmd,
        data: id ? undefined : data
    },
    function( res ) {
        if ( typeof res === "number" ) {
            that.data.id = res;
        } else if ( res.id ) {
            for ( key in res ) {
                that.data[ key ] = data[ key ];
            }
        }
    }
}
```
several decisions reflected there

- not RESTful, as discussed
- flexible signature for a single method without explicit flags
- most important: happens in the background
we’ve been talking about objects
but we want those objects to talk to a framework
it would be nice to have notifications

- when object data is set
- when the data is synced with the server
- if an error occurs anywhere along the way
messaging options

- document-level DOM events
- super-basic pub/sub
- promises
- third-party utilities with multiple options
these are all legit.

but we’re going to use the simplest
we want events to be scoped to objects

Frmwrk.Object = function ( obj ) {
    $.extend( true, this, {
        sel: "body",
        data: {},
        endpoint: null,
        events: {}
    }, obj);
    if ( this.sel ) {
        this.el = $( this.sel );
    }

    return this;
};
we’ll manage that cache with three methods

```javascript
Frmwrk.Object.prototype.pub = _pub = function( name, args ) {
    var that = this;
    if ( this.events[name] ) {
        $.each( this.events[name], function() {
            this.apply( that, args || [] );
        });
    }
};
Frmwrk.Object.prototype.sub = _sub = function( name, callback ) {
    if ( !this.events[name] ) {
        this.events[name] = [];
    }
    this.events[name].push( callback );
    return [ name, callback ];
};
Frmwrk.Object.prototype.unsub = _unsub = function( handle ) {
    var evts = this.events[ handle[0] ];
    ```

hat tip to Pete Higgins
now when something happens, we can publish an event

```javascript
Frmwrk.Object.prototype.set = _set = function (data, val) {
    if (typeof data === "String") {
        this.data[data] = val;
    } else {
        for (key in data) {
            this.data[key] = data[key];
        }
    }
    this.pub("updated", this.data);
    if (this.endpoint) {
        this.sync();
    }
    return this;
};
```
or if the wrong thing happens, we can notify observers of the error

```javascript
Frmwrk.Object.prototype.sync = function( data, cmd ) {
  var id = typeof data == "number" ? data : undefined,
      that = this;
  $.post( this.endpoint,
      {
        id: data.id || id || undefined,
        command: cmd || undefined,
        data: id ? undefined : data
      },
      function( res ) {
        if ( typeof res == "number" ) {
          that.data.id = res;
        } else if ( res.id ) {
          for ( key in res ) {
            that.data[ key ] = data[ key ];
          }
        }
      }
    );
};
```
global events can be scoped to the app

```javascript
Frmwrk.pub = function() {
  _pub.apply( Frmwrk.app, arguments );
};
Frmwrk.sub = function() {
  return _sub.apply( Frmwrk.app, arguments );
};
Frmwrk.unsub = function() {
  _unsub.apply( Frmwrk.app, arguments );
};
```
notifications as an implementer

- controllers, presenters, viewmodels
- ideally, publishing events controlled by framework
- implementations merely listen
- but we have events we don't control coming from the DOM
connecting to the view
a JS framework should be decoupled from HTML and CSS

- maintainability
- modularity
- separation of concerns
- not using JS for things that don't require it
note: this is JS frameworks

client-side frameworks may be more tightly coupled, and do more view stuff
elements of the view

- templates
- CSS
- DOM listeners
- potentially a data transformation layer
an object might have more than one view, though

for that we need states
a new object type

Frmwrk.State = function( obj ) {
    $.extend( true, this, {
        parent: Frmwrk.app,
        tmpl: null,
        settings: {},
        callback: null
    }, obj );

    return this;
};
Frmwrk.Object = function ( obj ) {
  $.extend( true, this, {
    sel: "body",
    data: {},
    endpoint: null,
    events: {},
    states: {}
  }, obj);
  if ( this.sel ) {
    this.el = $( this.sel );
  }

  return this;
};
child objects force us to think about initialization

- we want implementers to be able to pass in literals
- but we want instances that can have prototype methods
- our options are:
  1. force states to be added explicitly
  2. filter states supplied on initialization
this is a place where it's nice to do some magic

```javascript
Frmwrk.Object = function ( obj ) {
  $.extend( true, this, {
    sel: "body",
    data: {},
    endpoint: null,
    events: {},
    states: {}
  }, obj );
  if ( this.sel ) {
    this.el = $( this.sel );
  }
  for( var s in this.states ) {
    if ( !this.states[ s ] instanceof Frmwrk.State ) {
      this.states[ s ] = new Frmwrk.State( this.states[ s ] );
    }
  }
};
```
once we have states we can switch between them

Frmwrk.Object.prototype.state = _state = function( name ) {
    if ( !name ) {
        return this._currentState == "_default" ? null : this._currentState;
    }
    if ( this.states[ name ] ) {
        this.states[ name ].render();
        this._currentState = name;
        return this.states[ name ];
    }
};
Frmwrk.state = function() {
    _state.apply( Frmwrk.app, arguments );
};
and, since they’re instances, use their methods

bringing us back to templates
Frmwrk.State.prototype.render = function() {
  var data = $.extend( true, {}, this.parent.data, this.settings );
  this.parent.el.html( thistmpl( data ) );
  if ( this.callback ) {
    this.callback.apply( this, arguments );
  }
};
contains several assumptions:

- `tmpl` has already been compiled to a function
- A state will always replace the object it belongs to
- Any partial templates are already accessible somehow
- The correct CSS is already loaded
that’s a lot

can we make some of those more flexible or automatic?
Frmwrk.State.prototype.render = function() {
    var data = $.extend( true, {}, this.parent.data, this.settings );
    if ( this.css ) {
        $( "head" ).append( '<style class="' + this._name + '">' + this.css + '}'
    }
    this.container ?
        this.container.html( thistmpl( data ) ) : 
        this.parent.el.html( thistmpl( data ) );
    if ( this.callback ) {
        this.callback.apply( this, arguments );
    }
};
loading templates and partials is trickier

- should be able to take a URL or a template element
- should know whether the template's already loaded
- should handle partials differently
- and know which partials it requires
we should handle that in our constructor

Frmwrk.State = function( obj ) {
  $.extend( true, this, {
    parent: Frmwrk.app,
    tmpl: null,
    settings: {},
    callback: null
  }, obj );
  // tmpl needs to be compiled
  if ( this.tmpl && typeof this.tmpl !== "function" ) {
    this.tmpl = _compileTmpl( this.tmpl, thispartials );
  }

  return this;
};
and do the loading work, if necessary, in a static function

```javascript
var _tmplCache = {};
function _compileTmpl( tmplString, partials, name ) {
    var tmplName = name || tmplString.replace( "\"", "\"" ).replace( ".", 
        def = {};
    if ( partials && typeof partials == "object" && partials.length ) {
        partials.forEach( function( p ) {
            def[ p.name ] = _tmplCache[ p.name ];
        });
    }

    if ( _tmplCache[ tmplName ] ) {
        return _tmplCache[ tmplName ];
    } else if ( $( "body" ).find( tmplString ).length ) {
        return _tmplCache[ tmplName ] = doT.template( $( tmplString ).html)
    } else {
        $.get( tmplString, function( tmpl ) {
            tmplCache[ tmplName ] = doT.template( tmpl, null, def );
        });
    }
}
```
we also need a way to add partials

- we can't expect partials to be tied to an object or state
- we need them to be available when the templates using them are compiled
thus, the first method belonging exclusively to the application

Frmwrk.app.registerTmpl = _registerTmpl = function( name, tmplString ) {
    _compileTmpl( tmplString, null, name );
};
why not just use `_compileTpl`?

we could, but it’s nice to have a separate public interface
it also helps reflect a different use

Frmwrk.Object = function ( obj ) {
    $.extend( true, this, {
        sel: "body",
        data: {},
        endpoint: null,
        events: {},
        states: {},
        _currentState: "_default",
        tmps: {}
    }, obj);
    if ( this.sel ) {
        this.el = $( this.sel );
    }
    if ( this.tmpls ) {
        for( var t in this.tmpls ) {
            _registerTml( t, this.tmpls[ t ] );
        }
    }
}
real talk: that could be much better

- everything's easy until it's async
- external resources are likely to require async code
- this is an ideal place for promises
- ..which we'd get from yet another external resource
we want a framework to also help manage dependencies

- what we have could work if dependencies are finite
- if they're generic, we'd be better off with something like Require.js
- something like that means we have to think of the framework differently
taking Require as an example..

- our object types would be modules
- the app would be a specific dependency
- we could remove any code to load templates or CSS
- all this code would be required by implementations
one more piece
wiring up the DOM
we've left a place for some implementation code

```javascript
Frmwrk.State = function( obj ) {
    $.extend( true, this, {
        parent: Frmwrk.app,
        tmpl: null,
        settings: {},
        callback: null
    }, obj );
    // tmpl needs to be compiled
    if ( this.tmpl && typeof this.tmpl !== "function" ) {
        this.tmpl = _compileTmpl( this.tmpl, this.partials );
    }

    return this;
};
```
some frameworks attempt to contain DOM interaction

- lists of selectors and events
- callbacks belong to the application or a controller
- this keeps everything nice and clean
- but doesn't add a whole lot of functionality
What DOM functionality can a framework add?

- re-rendering
- attaching DOM interactions to state changes
- forwarding DOM events to framework events
the rest is nobody's business but the DOM's
what we need to trigger a state change

- an element to delegate to
- the selector and event
- the name of the state to switch to
we want this wired up when our state is created

Frmwrk.State = function( obj ) {
$.extend( true, this, {
    parent: Frmwrk.app,
    tmpl: null,
    settings: {},
    callback: null,
    trigger: null
}, obj );

// tmpl needs to be compiled
if ( thistmpl && typeof thistmpl != "function" ) {
    thistmpl = _compileTmpl( thistmpl, this.partials );
}

if ( this.trigger && this.trigger.length ) {
    this.trigger.forEach( function( t ) {
        $( t.container || this.parent.el ).on( t.event, t.selector, this
    });
}
**how about more generic events?**

- we may still want to fire events scoped to an object
- our publish method is public, so implementers can do that manually
- for something very small and unopinionated, that's probably enough
but we should talk about opinions
this example is more useful as a complement to something else

- lots of control in HTML or CSS
- a DOM library
- a datavis library
- a widget or component framework
- a client/server framework
there are a lot of tools
but only a handful of strategies for augmenting them
with HTML/CSS

- framework provides initial data and rendering
- states can be exclusively data states, not visual
- state callbacks set up explicit data updates
- framework exists to store and sync data
CSS specifically cause that's kind of hard

- CSS interactions are an alternative to JS
- we can observe animations (e.g. animationStart, animationEnd)
- but we can't directly observe something activating :target, for example
- and we can't forward events from CSS to JS
- the most foolproof way to integrate with CSS is to listen for the same events
with DOM-heavy libraries or plugins

- object DOM element is just the container
- stores data for child elements that aren't objects
- child elements created by object's template
- object's default state callback listens for library events and updates its data accordingly
- again, storing and syncing
with components or widgets

- framework objects could be made less generic to be aware of library functions
- or the two can communicate from their controller code
- components should manage themselves, so framework only needs to worry about public notifications
- still storing and syncing, but at a higher level of remove
with frameworks that handle the backend

- again, we can be less generic and rely on server-aware objects
- those may still need rendering and client-only messaging
- this time, we're observing third-party events coming not from the user but the server or the data
- handle the manipulation of data, leave the rest to the underlying framework
in the real world, we usually only think about pieces of frameworks in conjunction with existing frameworks
what to do with these ideas

- don't go write a new generic do-everything framework
- think about what pieces your app needs from a framework
- choose frameworks that are opionated about those things
- if you can't find a perfect fit, write a small wrapper that adds just what you need
thanks!
any questions?

garann.com | @garannm