twoBirds v5 brief info:

google polymer working on current devices

Goodies

multiple inheritance in javascript
adds a selector tb() to the programming
has a coding scheme providing a client side repo and an instanciation mechanism
async requirement loading to the extreme
does not interfere with other libs (unless they utilize $(...).data('tbo'))
failsafe, even when encountering logical coding errors

twoBirds quick start:

1. add the twobirds.....js file(s) to the head
2. for every DOM node you want to be a tb object (selector target):
   <myNode data-tb='/path/to/file.js'></myNode>
   the JS file is supposed to fill some content into the path.to namespace (repo)

The DOM node body should look like this:

<body data-tb="/body.js" … ></body>

… and obviously this file should exist ;-)

typical sample <anyfile>.js ...

path.to.<anyfile> = {

...

}

after twoBirds is done with it <anyfile>.js …

… the **DOM node** should contain a **$().data('tbo')** which is the twoBirds object instance.

twoBirds selector:

tb(selector) always returns the tb object selected (not a jquery or DOMnode object), or an array of matching objects:

- if there is no match, its an empty array
- if there is one match, its the object itself
- if there is more than one match, it is an array of twobirds objects

There are 2 types of selectors:

1. A sting is treated as a jquery type selector, $(selector) is executed, and if the result contains a $(selector).data('tbo'), this tbo is added to the tb(selector) result. DOM nodes not containing $.data('tbo') are ignored.
2. A js RegEx parameter is treated as a selector on the tbo.name property and executed on it. So if a tbo objects name or either of its sub objects names match the regex, its added to the result set.

You can refine the result set by the following chained methods:

- .parent(selector)
- .parents(selector)
- .children(selector)
- .descendants(selector)
- .is(selector)
- .instanceOf(repo object)

All communication between twoBirds instances is by triggering asynchronous …

tb(selector).trigger(event[, data]);

… event triggers.

Since twoBirds instances can recursively contain other twoBirds instances, multiple inheritance is possible and easily maintained.

twoBirds instances

Default and dotted properties:

- ***target***
  DOM node that contained the data-tb attribute
- ***status***
  (should always be 'done')
- ***_super***
  if object is within another tbo, this points to the parent
- ***_root()***
  if object is within another tbo, this points to the topmost anchestor
- ***name***
  in the topmost anchestor, its a system generated id, otherwise the name given in the repo object
- ***handlers***
  an array of handlers as created by tb.events
- ***any property with a '.' (dot) in it receives special handling.***

Methods

- **initChildren()**
  ...will walk the inner DOM of this.target looking for data-tb=“path/to/file.js“ nodes and initialize their loading
- **require(<array>, <callback>)**
  ...will async load the required files, and execute the <callback> function when done
- **trigger(<eventname> [, data])**
  ...will trigger <eventname> on the instance and all of its tb sub-instances recursively.
  If a handler exists, it will be executed in the scope of the (sub-)instance.
  Multiple sub-instances can have a matching handler, then all of these are executed in their respective scope.
- **structure()**
  ...will output the complete structure of the tbo in question to the console for debugging.

Dotted Properties:

Best explained in an **example** - the repo object has this property:

```
'tb.events': [
        {       name: /tb.require:done/,
                handler: function(ev){
                        $(this.target).html( tb.loader.get('demoapp/body.html') );

                        this.initChildren();
                }
        }
],
```

If upon instanciation twoBirds encouters this property, it first looks up the namespace indicated by the dotted properties name: so, if window['tb']['events'] exists it continues, if not, tb.events is converted to the file name 'tb/events.js', this file is loaded and when the loading has successfully finished, then the system continues. Of course, the content of this JS file should somehow set the namespace.

Now that the namespace definitely exists, the system looks up its type, which should be one of these:

- plain object
- **function**
- constructor (function having a prototype)

...and this converts to...

- A plain object is simply merged into the instance in question using $.expand()
- **A function is executed in the scope of the instance** , <propName>(<propValue>), assuming it changes the instance somehow.
- A constructor is initialized using 'new <propName>(<propValue>)', the result treated as a plain object

In our example, this.events happens to be a **function**, and the instance will after execution of the function have this property:

```
'handlers': [
```

```
        {       name: /tb.require:done/,
                handler: function(ev){
                        $(this.target).html( tb.loader.get('demoapp/body.html') );

                        this.initChildren();
                }
        }
],
```

THATS IT. HAVE FUN!