And now for something totally different …

After reading this document, come back and read this statement:

> *If this doesn't seem really simple to you, then you are not getting it. Perhaps, we need to add/change part of this document. If you do get it, you might wonder why this hasn't been done before since it would have been possible pretty much since the DOM was exposed to JavaScript circa 2000. I do not know why this isn't the standard way to create hand-crafted HTML suitable for applications.*

Browsers really present the DOM (Document Object Model). You can give a browser HTML to parse or you could call DOM methods to create the objects directly. Calling the DOM methods directly is fast but would require a lot of lines of code to do what a few lines of HTML could do.

That being said, the DOM method is programmatic which makes it very suitable for doing things where the DOM objects to be added (or their properties) need to be dynamically determined.

So, lets look at what HTML is .. a tag with attributes. Attributes are sort of like properties, right? So, instead of HTML we could use a JavaScript object.

 or { tag:'div }

But, we can put children inside of the HTML div, so we need a way to do that in JavaScript.

Hello World

becomes

{ tag:'div', children:[{ tag:'span', text:'Hello World' }] }

Of course, this can be constructed programmatically as follows:

```
var div = { tag:'div', children:[] }
var greeting = "Hello World";
var span = { tag:'div', text:greeting };
div.children.push( span );
```

Ok, so that is pretty workable, but we have quite a bit of repetitive "ceremony" that can be dispensed. First, we know that "div" and "span" are common tag properties and the would not be used as a property name. Further, we know that the child node of these tags will either be text node (JavaScript typeof string) or one or more other objects.

So, we could shorten the code above to the following:

var div = { div:[{ span:”Hello World” }] };

Hmmm, that’s pretty concise while still be something that I can differentiate programmatically.

Attributes Supported:

See the HTML specification. Anything you add to the JS object other than the tag name is simply passed through to the browser as an attribute -- except for children, siblings and onrender -- see below. For example, you could do the following:

{ div:’Hi There’, “class”:”someClass”, “data-role”:”someRole” }

Note: class is a keyword in JavaScript so you have to quote it.

What is “data-role”? I don’t know, must be some Angular thing where the Angular code looks for elements having that attribute. Regardless, it will simply get transferred from the JS object to DOM object.

CSS or Styles

The HTML style attribute is a list of properties. So, in JS it is simply a nested object such as:

```
{
	div:”Hi There”,
	style: {
		color:’#ff0000’,
		backgroundColor:’#00ff00’ // could use “background-color” as the property name
	}
}
```

Event Handlers

Ok, now for the a good part .. anonymous functions to be used as event handlers.

```
{
	div:”Click Me!”,
	onclick:function () {
		console.log( “I was clicked” );
	}
}
```

Everything about this object in one place.

Of course, you might was to be able to call the handlers via an automated test in which case you could do something like the following:

```
var handlers = {
	btn:function () {
		console.log( “Button was clicked” );
	}
};

{
	div:”Click Me!”,
	onclick:handler.btn
}

Test code:
	handler.btn();
```

Q. Ok, so what if I commonly use the same type of objects?

A. Write a function that creates such an object and returns it.

```
var makeButton = function (text,onclick) {
	return {
		div:text,
		“class”:”btnClass”,
		onclick:onclick
	}
};

makeButton( “Open”, handler.open )
makeButton( “Close”, handler.close );
```

Ok, so how do you put these into the page?

```
// parentNode .. an HTMLelement such as document.getElementsByTagName(“BODY”)[0];
// object .. one of the JS objects we have been talking about .. including all of its children
jsl.dom.add( parentNode, object );
```

Again, you can create a whole page worth of objects and pass them all into a single call to jsl.dom.add which will then reflect through the whole object.

siblings

Sometimes you want to layout multiple items that are at the same level in the DOM hierarchy. Just as an object can have children, the object can also have *siblings* such as:

```
var div = {
        div:[
                … some child objects
        ],
        siblings:[
                … objects at the same level as the div
        ]
};
```

onrender

Sometimes you want to do something with the HTMLelement after it is rendered such as a jQuery thing. If so, you can add an onrender method to the object. The method is called and the HTMLelement is passed to it. For example:

```
var div = { div:'',
        onrender:function (el) {
                $(el).someJqueryMagicalMethod();
        }
};
```

Again, you can **fully** declare the object in one place.

The el property

After an item is rendered, it has a property named el. You can use this property to update the DOM object such as:

```
var div = { div:howdy };
jsl.dom.add( someParentNode, div );
div.el.style.color = '#ff0000';
```

or, you could add some more stuff inside of it:

```
jsl.dom.add( div.el, ... some object to add inside of the div .. );
```

Test Driven Design (TDD)

There are whole books on TDD, but my opinion is simply this .. I cannot think of a bug that was still there after I tested/fixed the code that had the bug -- key being tested. Bugs are born in

untested code -- either the code itself was never tested or it was never tested in its current "environment". By environment, I means supporting code that the buggy codes relies on or injects into. I would say that 99% of bugs I have created were due to unintended consequences -- I changed some code to effect a new feature and as a result something I didn't mean to change broke. Thus, there is a lot of benefit in just knowing that nothing changed that I did not intend to change. A unit test around the code would provide me with that benefit.

Of course, if I make the change and the test fails because all it is doing is detecting a change that is actually ok, I will have to update the test. For example, if I capture the output of a function and put it in the test, then I change the function the test is going to fail. I will have to update the test to accept the new output.

I would rather do this 100 times than have a customer report a bug (with a vague description as only a customer can create), than spend hours trying to reproduce the issue and track down the bug.

Every Day Usage

The application layout is typically:

1) some elementary navigation elements,
2) some dynamically created content elements,
3) lists,
4) tabs and
5) forms.

The first is simply done via this method or HTML -- it is done once so it doesn't really matter. The second will use this method as the content area is dynamic and maintains state -- i.e. you can navigate to some other area and come back to where you were. Lists are built using a stdList component which in turn is built on this methodology. More to be said about that in another doc Tab panels and individual tabs are also created via this methodology -- again another doc will described the API for such, Forms can also be created using this methodology and support customized onchange events or a generalized handler -- just push the changes to the server for this model.

$().html vs jsl.dom.add

jsl.dom.add is really a lot like the jQuery html() method except you are working with objects versus strings