JavaScript Libraries - Dojo Toolkit
Best Practices Guide

Version: 2013.11.22

Checklist

The following coding guidance are current for the Dojo 1.9 release. If using older versions of Dojo, some of these newer guidelines may not be relevant, but you should be aware of them in preparation for upgrading to newer Dojo versions.

General Coding Guidelines

- ❑ Always use AMD based modules
- ❑ Always use the AMD-based async loader
- ❑ Define a single class per AMD module
- ❑ Avoid deprecated and experimental Dojo modules
- ❑ Use dojoConfig and not djConfig
- ❑ Do not use DOM Level 0 "onload", use “dojo/domReady!” instead
- ❑ Do not use global variables in application modules
- ❑ Always document public variables and functions
- ❑ Always use enumerated topics for publish/subscribe
- ❑ Use `has()` tests for feature based code branching
- ❑ Extract common functionality into common modules

Dijit

- ❑ Use declarative syntax for static content, programmatic for dynamic content

- ❑ Use HTML5 “data-” prefix rather than “dojo” attributes for declarative widgets
- ❑ Consider using custom prefix convention for “`data-dojo-attach-*`” pointers
- ❑ Avoid "id" attributes in Dijit templates
- ❑ Always use AMD module ID notation in “`data-dojo-type`”s
- ❑ Use `own()` for tracking `on()` and `subscribe()` handles in Dijits

I18N

- ❑ Consider using a single base variable to hold all I18N instance strings
- ❑ Always use localized text strings, using standard Dojo I18N implementation

Other Topics:

- ❑ Debugging Dojo App startup visual issues
- ❑ Always follow the Dojo coding style guide

Discussion

Dojo Coding Guidelines

AMD

All non-legacy development should use AMD style syntax. This provides significant performance and modularity improvements. It is advisable to also convert any legacy code to use AMD as well.

If you are sure there is no legacy (non-AMD) modules in your application, then you must also set "`async: true`" in the dojoConfig variable prior to loading Dojo.

AMD modules should be written so that there is a single class -- either static or instantiatable -- per module or file. While it is possible to define multiple classes within a single module, it goes against the spirit of AMD.

See the ***JavaScript - AMD*** document for more information on AMD loading.

Avoid deprecated and experimental Dojo modules

Do not use legacy or deprecated modules in application modules. Note: it is ok for these to be within Dojo Toolkit's own modules.

Additionally, you should avoid "experimental" modules located in dojox/*. There should be a readme for each subsystem in dojox that provides its current state. Experimental modules are likely to have major changes to their API's between releases, and may get dropped altogether if there is no support or upkeep. If you do decide that an experimental module is useful and you add it to your application, then please reach out to the developer and let them know. Your usage could be the deciding factor on if an experimental module as a future or not.

Action: Scan code for the following deprecated modules:

- `dojo` : Instead of using the dojo module, use dependencies on specific smaller modules.
- `dojo/_base/connect` : Replace with:
  - `dojo/on` : Node event handler
  - `dojo/aspect` : Function before, after, around calls
  - `dojo/topic` : Loosely coupled event pub/sub notifications
- `dojo/_base/Deferred` : Replace by `dojo/Deferred`
  - `dojo/_base/Deferred.when` - Replaced by `dojo/when`

- `dojo/_base/kernel` : Avoid using this module which has large dependency chain. This module is not usually required directly by end developers, unless required for declaring your own custom modules are being deprecated, using `kernel.deprecated();`
- `dojo/_base/window` : Originally written to serve two main purposes:
    - Provide methods/variables to access the current document and the `<body>` element of the current document.
    - Provide functions to switch the *current document*, i.e. the document accessed by the methods/variables mentioned above, and indirectly by DOM methods where the document isn't implied by the arguments, for example `dojo.byId("xyz")`.
    - In modern code, you can usually forgo use of this module, and instead use; `window`, `document`, and `document.body` global variables, or equivalent variables for the frame that you want to operate on.
    - If you need to operate on a different frame/document, all of the modern Dojo DOM related methods either take a document parameter or a DOMNode parameter (which implies a document). For example:

```
require(["dojo/dom", "dojo/dom-geometry"],
function(dom, domGeom){
  var node = dom.byId("address", myDocument);
  domGeom.setMarginBox(node, ...);
});
```

    - The following `dojo/_base/window` functions are DEPRECATED
        - `dojo.doc` : The old dojo.doc variable is accessible through dojo/_base/window::doc, but usually you can (and should) just access the `document` global variable to get a pointer to the document. For functions that operate on a DOMNode, you can get the document via `node.ownerDocument`.
        - win.body()
        - win.global()
        - win.withDoc()
        - win.withGlobal()

- `dojo/_base/xhr` : Replace by `dojo/request`
- `dojo/_base/sniff` : Replace by `dojo/sniff`
- `dojo/DeferredList` : Replace with `dojo/promise/all` and `dojo/promise/first`
- `dijit/hccss` : Replace with `dojo/hccss`
- `dojo/io/iframe` : Replace with `dojo/request/iframe`
- `dojo/io/script` : Replace with `dojo/request/script`

- `dojo/ready` : (1.9) Use `dojo/domReady!` AMD loader plugin instead. The only caveat is that if you are using the parser and have custom javascript code to run, you should run the parser manually rather than setting `parseOnLoad:true`.
- `dojox/mobile/sniff` : (1.9) Replace with `dojo/sniff`
- `has("iphone")` : (1.9) replaced with `has("ios")` to detect iOS device. In future, `has("iphone")` will just be for phone. The dcordova and dworklight libraries can also be used for fine grained feature and platform detection.
- `dijit/_BidiSupport` : (1.9) replaced with dojo config setting like `data-dojo-config="has:{'dojo-bidi': true}"`
- `dojox/charting/BidiSupport` : (1.9) replaced with dojo config setting like `data-dojo-config="has:{'dojo-bidi': true}"`
- `dojo.global` : was originally created to either:
    - Access the global scope in a device independent way. (i.e for code to run on browsers and server side), see dojo/_base/kernel::global. Modern AMD modules probably should not be trying to set or read global variables at all.
    - For browser only code, access the window object such that other code could redirect the window object to point to a different frame. See dojo/_base/window::withGlobal and dojo/_base/window::setContext. For accessing the window object (to control scrolling, setup handlers, etc.), most application code can simply access the `window` global, rather than accessing dojo/_base/window::global.
- `lang.getObject()` : Review to make sure that global objects are not being used. Instead, objects should be passed as arguments or via messages.

Use dojoConfig and not djConfig

Use "`dojoConfig`" rather than "`djConfig`" attribute for establishing the Dojo configuration settings.

If defining the Dojo config inside the same script tag that actually loads the dojo.js file, then use "`data-dojo-config`", and not "`djConfig`"

Do not use DOM Level 0 "onload", use "dojo/domReady!" instead

Use the "dojo/domReady!" AMD macro to delay processing until the page is ready. Using this macro also ensure that Dojo is fully initialized and ready to go as well. Using the DOM onload event can cause problems as only one callback function can be assigned, and is easy to be overwritten.

Example:

```
require([..., "dojo/domReady!"], function(...){
  // We know the DOM and Dojo are now ready for processing
```

```
  ...
});
```

Do not use global variables in application modules

With modern AMD code, hopefully globals are completely unnecessary. If you do need to create/read a global, then the following pattern is preferred:

```
require([...], function(...){
  var global = this;
  ...
  global.myVariable = "hello world";
});
```

For strict modules, there's a slightly more complicated syntax:

```
"use strict";
require([...], function(...){
  var global = function("return this")();
  ...
  global.myVariable = "hello world";
});
```

Scan code for the following global variables. Do not use global reference, instead use AMD define() args:

- `dojo.`
- `dijit.`
- `dojox.`
- Any other potential globals used by your application, such as: `app.`

Note: In Dojo 1.8+, you may see Dojo's "declare" function used with three or four arguments, with the first argument being a string which is the global path to the name of the class. The string ClassName is for legacy compatibility with pre-Dojo1.7 code and in application code based on AMD can be removed. Example:

```
var myClass = declare("a.b.c.ClassName",
  [BaseClass, Mixin1, Mixin2, ...], {
  //properties and methods map
});
```

instead use:

```
var myClass = declare(BaseClass,[Mixin1, Mixin2],{
```

```
  //properties and methods map
});
```

Always document public methods and variables

It's a good idea to properly document all functions and variables, but make it mandatory for public entities.

Dojo uses a custom format for API documentation, but some may prefer JSDoc style syntax. Pick a format and standardize on it in your style guide.

Always use enumerated topics for publish/subscribe

Publish / Subscribe provides a loosely coupled way to communicate between modules. Using strings as topics is fragile, as changing the string (explicitly or by accident) can break the coupling and introduce hard to find bugs. Using a common enumeration of topics can help avoid these issues, as well as provide a consistent way to document available topics.

All topic names strings used in "`topic.publish()`" and "`topic.subscribe()`" should be defined in either a:

- Common "app.js" within a "topics" map, that is used by all other modules. Each module accesses "app.js" via AMD dependency list.
- A dedicated "topics" module, shared by all consuming modules.

Action: Scan code for "`publish(`" and "`subscribe(`" and ensure that the topic name is defined using "`app.topics.MY_TOPIC`" topic map names rather than hard-coded strings.

Use has() tests for feature-based code branching

The dojo/has module is used for feature detection in application code. You can determine the device or environment you are running under and conditionally provide/ disable features in your app. An example would be checking for a camera, and if detected, then add a capture image button.

This benefit can be extended during build time to create platform specific builds. This can have a dramatic impact on deployment file sizes. The dcordova and dworklight projects provide many such has tests that you can use in your apps. Other projects may also provide has tests, or you can always create your own!

Extract common functionality into common modules

Copy and paste programming must die. There is absolutely no reason to duplicate code blocks, functions, or entire module definitions in a project. This is a common

practice, and it really needs to be avoided. It's lazy, introduces bugs, causes long term maintenance issues.

If you have code that is reuseable, then it's a simple matter to extract that logic into a standalone module. Then other modules can add it as a dependency or class mixin, and we get clean code reuse, not duplication abuse.

We're not advocating many modules consisting of single functions. Try to group like functionalities into common modules (i.e. module provides a library of like functions). To extend this some more, group like modules into usage based subdirectories. Before long, you will have your own library of reusable module assets to be shared with the rest of your organization, or even the world.

Dijit

Use declarative syntax for static content, programmatic for dynamic content

A common question is "should I use declarative Dojo, or programmatic Dojo?". It's not an either-or scenario. You will almost always want to use both, but for different reasons. Basically, any dijits or other content that are static in nature should be defined through markup (declaratively), and let the parser instantiate them. Any content that is dynamic in nature, either through a data service call, or dependent on application state, should be created programmatically.

Declarative content is easier for a CSS designer to understand than JavaScript so having clean HTML fragments separate from the JavaScript code simplifies the process of styling the views of an app. It's easier to style if you can see the HTML structure directly, rather than having to read the program and try to figure out what the structure is based on programmatic api calls or having to use a debugger to derive the runtime HTML structure.

Use HTML5 "`data-`" prefix rather than "dojo" attributes for declarative widgets

Legacy Dojo used a non-conformant "dojo" prefix for elements tags that were used to define declarative widgets. It is now recommended that you use the proper "data-" attribute prefixes. This makes HTML template proper for validation. The follow list shows the new (and old) syntax for various Dojo declarative attributes.

- Use `data-dojo-type` rather than `dojoType`
- Use `data-dojo-attach-point` rather than `dojoAttachPoint`
- Use `data-dojo-attach-event` rather than `dojoAttachEvent`

Consider using custom prefix convention for "data-dojo-attach-*" pointers

Using the "`data-dojo-attach-*`" prefixes allows for a loose coupling of pointers between the HTML template and the controller. Within the controller, if you use non-obvious names, it can get confusing where attach-point variables are defined, and also when attach-event target functions are called from. By standardizing on defined prefixes for these pointers, you can avoid the confusion.

- Use "**dapXxx**" prefix for attach points. In the controller, seeing `var name = this.dapFirstName;` is instantly clear where the source variable came from. Using an ambiguous attach point name like "firstName" may cause confusion.

- Use "**daeXxx**" prefix for attach events. Likewise, seeing a function called `daeFirstNameChange` is obviously from an attach event. Whereas a function named `firstNameChange` is much less instructive.

Avoid "id" attributes in Dijit templates

Using fixed "id" attributes in dijit templates is strongly discouraged as it breaks the ability to instantiate multiple instances of that widget in an app. You have two alternatives to using fixed "id"s.

1. Use "`data-dojo-attach-point`" for local node references. This acts as a local instance ID for any given node in a template. In your dijit's controller, you can use that value as a local variable for direct reference to either the DOM node or the child dijit instance. For example:

   Template HTML:

   ```
   <input data-dojo-attach-point="dapFirstName"
       data-dojo-Type="dijit/form/TextBox" />
   ```

   Controller JS:

   ```
   this.dapFirstName.set("value","Richard");
   ```

2. Use the dijit's instance id as part of the id value. While ID's should generally be avoided, there are times when you need to use them. The <label for="someId"> tag is a good example. Under these circumstance, you should the dijit's "id" variable to ensure unique Ids. For Example:

   ```
   <label for="${id}_firstName">First name:</label>
   <input id="${id}_firstName" type="text" />
   ```

Always use AMD module ID notation in "data-dojo-type"s

Since Dojo v1.8 is is recommended that you use proper AMd module ID notation in "`data-dojo-type`"s. The older "dot" style package name syntax has global annotations, and will likely be removed in Dojo v2. While you should never rely on it, the diji template parser will also attempt to automatically load missing modules defined using AMD notation.

Example: `data-dojo-type="acme/foo/Bar"`

Use `own()` for tracking `on()` and `subscribe()` handles

One of the most common sources of memory leaks in Dijit is failing to track and close connect and subscribe handles. In older version of Dijit, it was proper practice to use local wrappers to perform connects and subscriptions, using this.connect() and this.subscribe() respectively. Both of these have been depreceted and will liekly be removed in Dojo v2. Also the Connect module itself has been deprecated in favor of the on() event tracker.

The own() function call accepts the results of on() and subscribe() calls as a series of arguments. Then when the dijit is eventually destroyed, these handles are properly closed. The syntax for own() is a little unique, but effective, as you can provide as many on() / subscribe() calls as you wish. You will typically define these handles in the postCreate() function. In the following example, we define an event handler and well as subscribe to a pub/sub topic:

```
this.own(
    on( this.dapMsgBtn.domNode, "click",
        lang.hitch(this, "processMessageClick") ),
    topic.subscribe( app.topics.ACCOUNT_CHANGE,
        lang.hitch(this, "processAccountChange") )
);
```

I18N

Consider using a single base variable to hold all I18N instance strings

You should always assume your app will require internationalization support. As such, never hard code any visible strings ever. The easiest way to manage loose strings is to create a class level variable to hold your strings. Then, when and if the app gets translated, you not only have all your strings centralized in one place, you can simply replace the variable's value with the internationalized string set (described in the next topic). In the following sample, we define a class variable called "text" that holds a map of all the strings used in the dijit.

```
text : {
    "title"     : "Hello there",
    "firstName" : "First name",
    . . .
},
. . .
```

Within the template itself, just use a template variable substitution, such as:

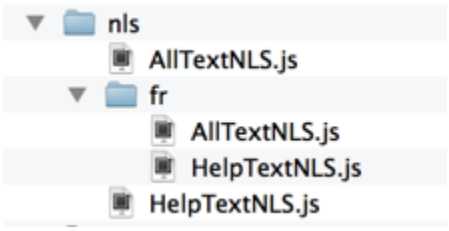
```
<h1>${text.title}</h1>
```

Within the class controller, you use direct local access to the variable:

```
this.text.firstName;
```

Always use localized text strings, using standard Dojo I18N implementation

OK, word came down from the business stakeholders that your new app is quite popular in France. They need an estimate for restructuring the app to support internationalization. You respond with 1 day, because you followed the tip above! Now its a matter of extracting the various text objects from your modules and sending them off to the translation team. Once the translated files come back, you place them into an nls directory with the proper directory and file names. In the image below, we have a module called AllTextNLS that has a default (English) base text, and the French (and possibly others) translations.



It should be noted that there are two common scenarios for app translation. One is having a unique translation per dijit. This is great for modularity, but can lead to overall bloat if there are a lot of redundant key/values spread across many modules. The other option is to have a single (or few) common translation modules. This makes life much easier on the translator, but blurs modularity and encapsulation. Bothn approaches are valid, and its your choice as to which is best for your situation. In the sample here, we are using a single common module.

Within each NLS file is a specific Dojo I18N structure shown below. This is from the base file, which also informs what other translation are available.

```
define({
    root: ({
        "title"     : "Hello there",
        "firstName" : "First name",
        "cancel"    : "Cancel",
        "create"    : "Create",
        "discard"   : "Discard",
        . . .
    }),
    "fr" : true
});
```

In the original Dijit module, inform the module that translation files are available, by adding in a special AMD dependency macro like shown below. By convention you should place it near the top of the list as there is a tight binding between the module and NLS files. This makes spotting errors easier when the module name gets changed.

```
define([
    "dojo/text!./MyWidget.html",    // Dijit Template
    "dojo/i18n!../nls/AllTextNLS",  // Translation
    . . .
], function( template, i18n, . . . ) {

    templateString : template,
    text : i18n,
    . . .
});
```

Now, all we had to do was replace the “text” variable’s value with the proper I18N translation. That’s it.

Other Topics:

Debugging Dojo App startup visual issues

When a Dojo app starts up, there can be noticeable visual issues where the user sees normal HTML elements prior to them being converted into nicely styled Dojo Widgets. Generally, the issue is due to the "parseOnLoad: true" flag in dojoConfig, parsing the page, while also having a lot of startup activity. While this situation is not unique to hybrid apps, lower-end devices can make the issue more obvious.

The basic startup flow of a Dojo-based app is:

1. Main HTML page loads and renders
2. DomReady fires which causes more activity in Worklight, app, etc
3. ParseOnLoad now scans the entire <body> looking for "data-dojo-type" (dojoType) attributes, and instantiates those widgets
4. All Views are forced to "display:none", except either the first view, or the first view that is not already set to display:none.
5. The "visibility:none" flag is removed from the <body> style.
6. Any other app bootstrapping happens (can be a race here based on flow of events)
7. App goes idle waiting for user intput

The typical solution to this issue is to add the following to the page's <body> tag, as Dojo will make the body visible during its parsing.

```
<body style="display: none; visibility: hidden;">
```

The other possible case is that there is too much activity happening during app startup, and the parsing of the page -- either automatically or programmatically -- is getting delayed. If this is the problem, then you will either need to defer some of the extra app processing logic, or delay parsing/showing the page by manually calling the parser, like so:

```
parser.parse();
```

Always follow the Dojo coding style guide

Dojo has very specific standards when it comes to code structure and inline documentation. To ensure overall cohesiveness between the standard Dojo libraries, and your custom application code, it is recommended that you follow the Dojo style.

Reference:

- Dojo style guide
- ***Project and Coding Guidance*** document