Dojo Developer Guide

Dojo Developer Guide place holder page

Part 1: "Introduction"

Dojo provides a lot of power and attempts to make it digestable in layers. For server-side developers, there's "widgets without coding", for HTML+CSS devs Dojo provides wonderful facilities for quickly building template-driven widgets, and for serious JavaScript and DHTML hackers Dojo is the standard library you will wish JavaScript always had.

This book serves as a guide to these layers, introducing concepts as you need them and working downward from high-level usage to getting your hands dirty in building your own widgets, custom namespaces, and unit tests.

In the Introduction, you'll get an overview of how Dojo can help you, what problems it solves, and where in the book you might be able to best find the information you're looking for. Also, remember that because this book is maintained by the community and is online, you can search it (and the rest of the Dojo site) at any time.

Lastly, thanks for checking out Dojo and the Dojo Book. It's your applications that have inspired us to build Dojo and the stories of how people are improving experiences with the toolkit that keep us going.

Dojo Architecture

Dojo is a set of layered libraries. The bottom most layer is the packaging system that enables you to customize the distribution of Dojo for your application. On top of Dojo's package system reside language libraries such as the Dojo event system and the language utilities that greatly improve and simplify the lives of JavaScript developers. Environment-specific libraries are provided in some cases, but in most uses, you'll only need to know that all of Dojo works without incident across every major browser.

The bulk of the Dojo code lives in the application support libraries, which are too numerous to display completely in the diagram. `dojo.gfx` provides native vector graphics support across the major browser implementations of SVG and VML. `dojo.lfx` is a lightweight effects library, while dojo.io is "where the ajax lives."

Most of the "action" in Dojo is in the widget toolkit, that contains a template-driven system for constructing widgets, a declarative parser for putting widgets in pages without writing JavaScript, and a set of base components that implement common-case solutions to web interaction problems that HTML+CSS alone cannot solve.

Dojo: What Is It?

Dojo is an Open Source DHTML toolkit written in JavaScript. It builds on several contributed code bases (nWidgets, f(m) and Burstlib), which is why we refer to it sometimes as a "unified" toolkit. Dojo aims to solve some long-standing historical problems with DHTML which prevented mass adoption of dynamic web application development.

Dojo allows you to easily build dynamic capabilities into web pages and any other environment that supports JavaScript sanely. You can use the components that Dojo provides to make your web sites

more useable, responsive, and functional. With Dojo you can build degradeable user interfaces more easily, prototype interactive widgets quickly, and animate transitions. You can use the lower-level APIs and compatibility layers from Dojo to write portable JavaScript and simplify complex scripts. Dojo's event system, I/O APIs, and generic language enhancement form the basis of a powerful programming environment. You can use the Dojo build tools to write command-line unit-tests for your JavaScript code. The Dojo build process helps you optimize your JavaScript for deployment by grouping sets of files together and reuse those groups through "profiles."

Dojo does all of these things by layering capabilities onto a very small core that provides the package system and little else. When you write scripts with Dojo, you can include as little or as much of the available APIs as you need to suit your needs. Dojo provides:

- **Multiple Points of Entry** - You can start using Dojo at the level you are most comfortable with. For example, expert JavaScript programmers can use the foundation capabilities to be more productive quickly, while Web designers and developers can use the set of easy to use, modify, and extend components that make their applications more responsive without requiring them to learn a large JavaScript API. This fundamental design decision drives the layered implementation of most of the major capabilities of Dojo.
- **Interpreter Independence** - Dojo is squarely a JavaScript toolkit but, within the realm of JavaScript interpreters and environments, not everything was created equally. Dojo supports at least the very core of the system on as many JavaScript enabled platforms as possible. This allows Dojo to serve as a "standard library" for JavaScript programmers as they move between client-side, server-side, and desktop programming environments.
- **Forward Looking APIs** - No one has a crystal ball when it comes to what technologies will be broadly available or used in 5 years, but Dojo attempts to provide APIs that are generic enough to be (directly) useful with todays capabilities while still building in room for future improvement. The dojo.io.bind() interface is a great example of this principle: when first written it wrapped only a single Transport class, but now provides a normalized interface to many ways of receiving and sending data from JavaScript enabled environments.
- ReducingBarriersToAdoption - This core philosophy behind Dojo's design acknowledges the fact that tools which are hard to use just won't get used, no matter how good they are. Dojo should be built in every way (licensing to deployment) to not give users any reason to not trust or use Dojo for the tasks it's good at. Many of the project's overall decisions get made on the basis of this principle.

Dojo is being built around a single markup language that provides application authors a (more) simple way of declaring and using responsive DHTML interface components. Renderings can be made available in several rendering contexts (such as SVG, or perhaps even the desktop or Flash), but the markup language (DojoML) and scripting language (JavaScript) will not change. Better yet, the DojoML parser accepts extended HTML and SVG as valid input, and can be used to easily create Degradeable Responsive Applications.

Dojo's homepage is: http://dojotoolkit.org.

How this Book is Written

Imagine for a second that you've just come from looking through the API of Dojo. You've memorized every function, every parameter, every example. All of the API has been studied and digested, but you're still not sure how to proceed. You know that there are many ways to do event handling, but you don't know the difference between them and you certainly don't know which one is best suited for your application.

I've made this assumption in writing the book; it allows me to focus on what's important about the book, and separate it from the API. With understanding Dojo, the *why* of Dojo, you're now free to learn the *how* by looking at the API. Or, if you already know the API, you're free to learn why it

exists in the first place.

Voice

Me

Though this book is a result of many people's work, I(we)'ll be speaking in the first person throughout this book. It not only allows a more personal interaction between the reader and the author, but quite often, describing an author's action is confusing when expressed in the plural form. "We're now opening our text editor" is an awkward sentence at best.

You

Saying "the reader" every sentence is dumb, and since you already know that you're the one reading it, I'll just refer to you as "you".

We

Decisions on the coding of Dojo aren't made by a single person. Almost every significant change to Dojo is done after discussion of how it is engineered. Because of this, I'll run into situations where I want to discuss how something was decided on. In this case, I'll be using the word "we". As in, "we decided that we wanted Dojo to be awesome."

The Tree Structure

On the way to a final topic, you'll be going through some higher-level topics. In order to understand the lower-level topics, and where they fit in to Dojo, you need to have some explanation of how you got way down there. To do this, each of the points along the tree, all the way down to the edge of the branch will have a few paragraphs of explanation. These are overviews for all topics contained underneath. It's worthwhile, if you have a specific topic that you want to read about, to go down the tree, reading each of the brief descriptions, before you read the article itself.

Articles

Readability

I want this to be more a book than a user manual. Therefore, I've tried to use fun analogies and brief stories. Many topics of this book are very difficult to process, and using existing knowledge to bring you up to speed is a great way to make these easy to digest. As a general guideline, I'll try to start an article with a paragraph, followed by a brief analogy or story, flowing into content. Because I don't want you to feel overwhelmed by code, examples will not be back to back. I think that if I have multiple examples, it's important to explain why I provided each one and the differences between them.

Length

I think that the moral of the story of Goldilocks and the Three Bears is that things should neither be lacking, nor in excess, they should be "just right". That said, there's nothing worse than sitting down to get your learn on and be constantly switching from one topic to another, from one page to another, when you want to stay focused on what originally brought you to the article to begin with. Likewise, you don't want to get halfway through an article and realize that you missed both lunch and dinner. To prevent this, as a general guideline, each article in this book should take you about twenty

minutes to read. That sounds just right to me.

Sections

I discuss in the book how Dojo is a toolkit, a framework, and everything in between. It's overwhelming just to learn what Dojo can do for you, much less learn what you can do to Dojo. Because of this, the book is split into three different parts.

Part 1 is all about using Dojo, the toolkit. That is, looking at a set of functions and objects that are available to you "out of the box" that don't require any tinkering. Ideally, you should be able to call a function, or initialize and object, without providing any custom logic to it.

Part 2 is meant to focus on more than just using things. We want you to know how to build your own widgets, packages, extend CSS and HTML and get into the nitty gritty of things, such as replacing the data provider for a widget.

Part 3 is a "look into the engine" explaining not only how thing work but why the Dojo developers designed them that way. Much of Dojo's code seems to work based on voodoo. For the advanced programmer, knowing this voodoo will allow you to write tighter, better code.

How to Create Pages

On any page where the dojo.book sidebar appears (including this page) click "create new page". The new page will have the currently viewed page selected as its parent by default

What Dojo gives you

Because Dojo is a toolkit, it's potential uses are unlimited. The only real restrictions are that you're using it in an envornment that uses JavaScript. Though many of you will be reading this in the context of a browser, it's important to remember that JavaScript isn't limited to just that specific use.

Code Simplification

With "Ajax" becoming such a popular buzzword, many are looking for an end-all solution to its complexities that not only work well in existing environments, but will allow much more complicated interaction in the future, allowing for very complicated transfer of data between client and server. With the aformentioned widget functionality, and an easy way to implement your own widgets, Dojo works very well alongside HTML.

Dojo also abstracts many of the differences between browsers that have pained developers for years. Things such as differing event objects and HTTP transport systems are a worry that will be forgotten.

Reusable Code

Dojo provides a launching point for developing code. This means that you can quickly throw a project together for a client or boss without having to reinvent the wheel. We've built Dojo to be highly reusable, so code you have created for one project can be quickly moved to another without any refactoring.

Portable Tools

Responsive applications make many calls to the back-end of web applications which have different call and response characteristics. Standardizing on a portable set of tools for this is beneficial to understanding and predicting overall system behavior.

Because Dojo's widget system sits on top of standard HTML, designers will be able to dive right in with a very shallow learning curve. Dojo allows for designers to add degradable functionality to their (already existing) page without having to implement any logic or strange programming language.

Additional Resources

This book describes what Dojo provides and how to use the toolkit. You will find detailed information about the Dojo APIs in the API Reference Doc.

What is a Toolkit?

Many people see the words framework, library, and toolkit as synonymous. This is true in the sense that they are all descendants of the same parent. Understanding the difference allows you to go beyond that widely scoped overview.

Three words are used to describe a stereotypical developer: geek, nerd, and dork. Many people assume that these words all mean the same thing when, in fact, they each have very specific meanings. Once you learn the meanings of each of these words, you'll quickly be able to run down the list of all of your friends listening to techno in dimly lit rooms and sort them into each category. Even better, you now have the power of more accurately describing a person.

> **Geeks, Nerds, and Dorks:** A geek has a very focused knowledge of a subject (that guy that memorized the language of myst), a nerd is a master at many subjects (that girl you go to when you need homework help), and a dork is just plain socially inept (Napoleon Dynamite).

Framework

In software development, a framework is a defined support structure in which other project can be organized and developed. A framework typically consists of several smaller components; support programs, libraries, and a scripting language. There may also be other software involved to aid in development and meshing of the different components of a project. As you can see, dojo could be part of a framework, but it isn't a framework in itself.

Library

A library is defined as a collection of related functions and subroutines used to develop software. They are distinguished from executables in that they are not independant programs; rather, they are "helper" code that provide access to common functions in one easy to manage location. After reading this you are probably saying, "Hey! dojo is a collection of libraries!", and you would be correct; however, dojo is much more than just a collection of libraries.

Toolkit

Now on to toolkits. A toolkit is generally used in reference to graphical user interface (GUI) toolkits. Basically, a library that is mainly focused on creating a GUI. Yes, dojo could also fall under this category, in fact our name implies it. Why do we call dojo a toolkit? Certainly not because it focuses

mainly on GUI development, right? Well quite simply, because dojo is so much more than just a collection of libraries.

None of the Above?

The previous paragraphs have probably left you still wondering what exactly we consider dojo. Obviously it is not a framework, but is it a toolkit or a library? Let's solve this once and for all. Typically, a library is a predetermined file that you include into your application, and that is how you gain access to those functions. However, with dojo, we have wrapped a package system around our libraries. This brings a slight twist to the idea of a library.

With this system we have broken each library up into several pieces. You have the core functions, and then several sub libraries where related, but less often used, functions are stored. This helps keep dojo's footprint based entirely on your needs as a developer. More about that will be covered later in the book but, for now, know that because of this flexibility, dojo is more than just a library, which falls into the realm of a toolkit with a few added functionalities. So as the name implies, dojo is a toolkit... and yet is more.

Part 2: "Out of the Box" Dojo

Want to know what Dojo is and how it can help you with your javascript development? In this section we will walk through the process of putting Dojo to use with its collection of pre-packaged "Out of the Box" widgets that can get you using Dojo right away. We will focus on developing programs using existing dojo components. This section will not get into any advance features, as they are included in future chapters. If you are new to Dojo or even new to JavaScript, then this chapter is for you. If, however, you are a seasoned programmer, you can still find reading this section purposeful as it will provide a helpful overview to the Dojo toolkit.

The sample application is the familiar Hello World app however, it does introduce the main features of Dojo and will build the foundation for developing much more sophisticated web applications.

Thanks to Lance Duivenbode and Seth Fair for help in writing this chapter.

Development and Debugging Tips

This section talks about some of the tools available to help develop JavaScript programs. With the increasing popularity ofJavaScript there surely will be alot of changes in this space in a relatively short time. In this section you will find helpful tools that may or may not be part of Dojo.

Also remember to check the Dojo FAQs especially the Common Pitfalls section.

Debugging JavaScript

There are excellent tools to help write and debug Javascript - it isn't all about liberally using alert() any more!

IDEs

IDEs are available which support a wide range of web development activity, including editing Javascript and HTML files, deploying code to servers, and integration with existing features like source control. In addition, some include runtime tools and browser integration to assist in debugging (for example, myeclipse and ATF)

If you use one of the JavaScriptEditors that checks your syntax as you enter your code, you can be warned immediately of simple errors and save yourself a lot of time. Some Eclipse plugins even specifically support Dojo idioms by giving you tooltips for common Dojo functions and doing more complex analysis for errors (not just normal syntax checking).

Browser specific tools

Mozilla/Firefox

- The Web Developer Toolbar is brilliant.
- The Firebug extension is a must for debugging and inspecting html pages - dojo.debug can also be configured to output directly to Firebug's console. It contains a simple Javascript debugger too. just use dojo.require("dojo.debug.Firebug") in your pages.
- Venkman is the mozilla javascript debugger - ugly and cantankerous, but it can be useful if you need to debug in FF. Note that in general the IE debugger is much better. Use the venkman port for FF1.5.
- Live HTTP Headers is a good extension for debugging HTTP traffic (You may prefer to use the equivalent functionality that exists in the in Firebug extension).
- JavaScript Shell - While it's not a debugger, I've found the JavaScript Shell to be a really great tool for analyzing problems, and even just exploring JavaScript libraries like dojo. Works best as a firefox bookmarklet, in which case you can open it in the context of any given page and invoke javascript functions yourself, evaluate expressions, redefine functions, etc.

Debugging on Firefox may need you to use the djConfig flag debugAtAllCosts (see further below for information). The debugAtAllCosts flag is sometimes necessary to locate exceptions or syntax errors. Even without the debugger, Mozilla and Firefox are unable to locate a line of code loaded by dojo.require() due to a flaw in the way eval() debugging hooks are implemented in Spidermonkey. Currently, the Javascript console will report all source references as the location of the eval() call itself (in the bootstrap code), but with an additional line offset equal to the offset in the corresponding *.js file.

Firefox Safe Mode

If you are having wierd problems with Firefox, it is often worthwhile running Firefox in safe mode.

This is because installed extensions can interfere with the DOM tree, CSS, or even with javascript. In Windows there is a shortcut to start Firefox in safe mode from within the Mozilla Firefox folder, from the Start button.

Internet Explorer

- Microsoft Script Editor Ã¢â‚¬â€œ very stable, part of MS Office web scripting add ons.
- Visual Web Developer Express is free.
- Visual Studio .NET has a built in Javascript debugger if you have Visual Studio.
- Microsoft Script Debugger a piece of crap compared with MSE.
- IE DOM Inspector is awesome (commercial product with 15 day free trial)
- IE Developer Toolbar is super useful (free from Microsoft)
- DOMSpy is ok
- IE DocMon is pretty good

If you are using a debugger with IE, go to Tools | Options | Advanced and make sure that Disable Script Debugging is not ticked. Using debugAtAllCosts can also help significantly - read about it below.

Safari

The debugger is Drosera.

The Safari Developer FAQ has some general information about developing with Safari, as well as instructions on how to turn on a debug menu that allows showing a Javascript console.

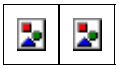
There is a "dom inspector" type tool, but it requires using a nightly build of Safari.

debugAtAllCosts

debugAtAllCosts may have unobvious side-effects - you should only use it if you are actually debugging. If you hit problems with files not loading or __package__.js then check that you are using writeIncludes() correctly, and try removing this flag to ensure that the flag is not the issue.

You generally should not use a packaged build if you want to debug, because in a packaged build the majority of code will end up in your dojo.js file (and usually be obfuscated due to compression).

The easiest way to understand how it helps with debugging is to see it's effect within a debugger. For example within Visual Web Developer Express the image on the left is when using debugAtAllCosts and the RHS image is without debugAtAllCosts being used:



To use it needs one more line in your page. Here's how you might use it:

```
<script>
        djConfig = {
                isDebug: true,
                debugAtAllCosts: true
        };
</script>
```

```
<script src="/path/to/dojo/dojo.js" />
<script>
        // dojo includes here
        dojo.require("dojo.myModule");
        dojo.hostenv.writeIncludes(); // this is a new line
</script>
```

You cannot do this for files that are packaged into your dojo.js (Of course, using packaged files makes

Profiling

Javascript is slow and you want to speed it up?

Use a ProfilingJavascript tool to help you find the functions where all the time is spent, and optimise those routines.

Traffic analysis

A great many problems can be resolved by watching the traffic between the browser and the server. If you are having any problems with Ajax calls such as bind() or with js/html/css/jpg files not loading as you would expect, this is often the best way to diagnose them quickly. Also if you are having problems with required files not loading then this is a good starting point to find out why.

- Fiddler is a fantastic HTTP header and content inspector for Windows. It understands HTTP and presents the information very clearly (once you get used to its slightly quirky interface). Well integrated with IE, but work s with any browser. With FF a useful extension to help use Fiddler is the Switch Proxy extension (scroll down that page to find it).
- Ethereal is great for sniffing all kinds of network traffic.

How do I debug syntax errors? aka How can I find this syntax error in bootstrap?

Sometimes Dojo's error messages about syntax errors are really cryptic due to how dojo loads files via dojo.require(), however you can get a better message by simply directly including the js file with the [script] tag.

The debugAtAllCosts flag can also be used.

There's also a lint program that Brian has recommended.

You can also find the debug error message in the dojo source code and remove the corresponding try/catch statement so that when the error occurs you get a debugger breakpoint instead of a dojo debug message.

Javascript debugger statement

Javascript has a debugger keyword that forces a breakpoint to occur. Just insert debugger; and if you have a debugger for your browser, then it will stop at the debugger keyword.

This is especially useful when using Venkman, because otherwise it can be difficult to get Venkman into debugging mode (e.g. you try to click on a line of source and it puts in a [F] future breakpoint).

It also works with the Firebug extension for Firefox.

Debug Messages

You can debug the old fashioned way (print statements in the code). There are three functions for this:

- dojo.debug - prints a message
- dojo.debugShallow - prints all the members in an object
- dojo.debugDeep - opens new window w/tree showing structure of an object

Debugging output:

- FireBug object - this will print debugging output to the Firebug console
- DebugConsole - will capture all your debug messages in a floating pane.

Alternately, in djConfig, you can specify which element they are appended to by

providing an id of that element: i.e.:

var djConfig = {

isDebug: true,

debugContainerId : "dojoDebug"

};

and have a div

JavaScript Editors

Editors/IDEs

ATF (Eclipse)

Ajax Toolkit Framework is an incubator project within the Eclipse Web Tools Project. It is also a pluggable framework for other AJAX tools. It provides Javascript syntax checking, server deployment, Mozilla embedding, runtime tools such as XHR Monitor, and a debugger based on Mozilla Spidermonkey, JSD, and the Eclipse debug UI. Eclipse integration provides access to existing plugins like ant, subclipse for SVN, server development tools, etc. AJAX "Personalities" provide the potential for tighter integration with Dojo and other toolkits, although so far very little has been done in this area.

There's visual Javascript validation built into ATF. It includes both basic syntax validation and optionally Jslint validation for less obvious potential syntax problems. It works just like MS Word as-you-type spellchecking and Eclipse as-you-type Java validation - as you type if you make a syntax error (or just do something somewhat sloppy) you'll get a red or yellow squiggly under the warning/error and an explanation in the margin. A background task will run validation against all files and place markers in the code such that you can see errors across a project.

You can download and use ATF and its prerequisite Eclipse components (Eclipse SDK, WTP, EMF, GEF, JEM, xulrunner) and choose to use all of ATF or just install the "javascript" feature which implements this validation.

Note - you have to fetch and manually install jslint.js, Dojo and xulrunner due to Eclipse policies - make sure to read the ATF readme to get the full functionality of ATF.

JSEclipse

JSEclipse is a commercial Eclipse plugin which provides a rich Javascript editor with support for syntax validation, code completion and Dojo idioms.

MyEclipseIDE

To be written...

JS-Sourceror (Eclipse)

JS-Sourceror performs syntax checking and variable type and flow analysis on JavaScript files.

JavaScript plugin for jEdit

See http://skrul.com/blog/projects/javascript

It also does syntax checking, as well as scope checking and structure browsing.

Profiling JavaScript

Profiling is the term for looking at where time is spent by any Javascript code. If you have a problem with code taking too long, then it helps to use a profiling tool to diagnose exactly where all the time is being used.

When using some profiling tools you may need to use debugAtAllCosts and not a packaged version of Dojo (see DebuggingJavascript). Using debugAtAllCosts will enable the profiling tool to allocate time spent per function to the correct source file -- otherwise you will end up with the elapsed times being allocated to anonymous functions which will make it difficult for you to understand!

Profiling Tools

dojo.profile

dojo.profile is a package that implements timing primitives for recording how much time is spent in particular functions. To learn how to use it, the best resource is to search the tests directory and look at how it is used by various tests.

FireBug 1.0

Displays the time it took to load each file.

Venkman (for Firefox)

Free but a bit buggy to use. I have found it easiest to get into debugging mode with Venkman by using a debugger keyword in your sourcecode. Run the Venkman debugger, then run your code and it will stop at the breakpoint.

Venkman contains a profiling tool, although the reports are a bit difficult to use. It does work.

Tito web studio

It seems it does not work with Dojo. It modifies JavaScript, and obviously in some manner incompatible with Dojo.

Manually do it yourself

Get the time at the start and end of a routine, and calculate the difference. Not a great technique, because you have to make educated guesses as to where the inefficient areas are, and it may take quite a lot of work to home in on the problem area.

But it is easy to do if you are measuring a single function:

function() {

var startTime = new Date();

```
// do something here

dojo.debug("Total time: " + (new Date() - startTime));

}
```

Or another example of code to do this is on this page near the top under the heading "Measure your changes".

Getting Started

If you are new to Dojo or want a quick overview of the toolkit then take a look at the HelloWorld Tutorial. This tutorial describes step by step how to build a simple Dojo application. You will learn some basic concepts about widgets, events and how to connect to the server code. Each step builds on the previous until you have a working application. It takes about an hour to go through the tutorial.

Adding Dojo to Your Pages

Dojo offers many editions of its code base. At first, it might seem daunting to try to figure out exactly which one you need. To quickly dispel any worries, let me assure you that every single edition of dojo provides a fully functioning system. Whether you download one of our editions, or the full, uncompressed source code, you'll be able to perform any of the examples discussed in this book.

TODO: fold in information from the README

In order to use dojo in your HTML pages, you need three sections of code, in this order:

1. Flags

```
<script type="text/javascript">
        djConfig = { isDebug: false };
</script>
```

The flags control various options of dojo; often developers will set isDebug to true in order to get debugging output on their page. (There are also some other flags related to debugging; see the Debugging section of the code for details.)

2. Include the dojo bootstrap

```
<script type="text/javascript" src="/path/to/dojo/dojo.js"></script>
```

This includes the bootstrap section of dojo, and if you are using a release build, then dojo.js will also include code for some of the dojo modules.

3. Define what resources you are using

```
<script type="text/javascript">
  dojo.require("dojo.event.*");
  dojo.require("dojo.io.*");
  dojo.require("dojo.widget.*");
</script>
```

This section is much like java's "import" statement. You specify every resources that you are using in your code. However, note that widgets are a special case and don't need to be declared explicitly, assuming that (as is the case with the built-in dojo widgets), a manifest file defines which widget is in which resource file.

Pre-Packaged Builds

Even though Dojo is made up of many different packages, it's frequently used in very specific ways. Because of this, we've created special editions of Dojo aimed toward these users. A visit to dojo's download page will show you which editions are currently available, such as the *Ajax* and *Widget* edition.

An edition is very simple, really. The important file is dojo.js, which is created by merging the most frequently used packages and compressing the resulting code. This means that when you have a script tag that calls dojo.js, you're getting not just the basic Dojo codebase, but the additional functionality that is most pertinent to your specific use.

You might wonder why so much additional code comes in each edition. This is the full code base, and allows you to use functionality that is outside of your specific build. It means that even if you have a very specifically tailored edition of Dojo, you aren't limited to only using that featureset. If your site uses the event and I/O systems heavily, but one of your pages uses a widget, then you don't have to worry that your widget will break. This also means that any of the examples in this book will work no matter what edition you've downloaded.

Introducing Dojo Events

Events in JavaScript or Dojo based applications are essential to making applications work. Connecting an event handler (function) to an element or an object is one of the most common things you will do when developing applications using Dojo. Dojo provides a simple API for connecting events via the dojo.event.connect() function. One important thing to note here is that events can be mapped to any property or object or element. Using this API you can wire your user interfaces together or allow for your objects to communicate. The dojo.event.connnect() API does not require that the objects be Dojo based. In other words, you can use this API with your existing interfaces.

Below is the code in the tutorial handling events. Here we connected the event handler, helloPressed, to the onClick property of the hello button element. When the button is clicked the funtion helloPressed will be called.

```
function helloPressed()
{
  alert('You pressed the button');
}
function init()
{
  var helloButton = dojo.widget.byId('helloButton');
  dojo.event.connect(helloButton, 'onClick', 'helloPressed')
}
```

It is also possible to use the Dojo event model to connect simple objects. To demonstrate, lets define a simple object with a couple of methods:

```
var exampleObj = {
    counter: 0,
    foo: function(){
        alert("foo");
        this.counter++;
    },
    bar: function(){
        alert("bar");
        this.counter++;
    }
};
```

So lets say that I want *exampleObj.bar()* to get called whenever *exampleObj.foo()* is called. We can set this up the same way that we do with DOM events:

```
dojo.event.connect(exampleObj, "foo", exampleObj, "bar");
```

Now calling *foo()* will also call *bar()*, thereby incrementing the counter twice and alerting "foo" and then "bar". Any caller that was counting on getting the return value from *foo()* won't be disappointed. The source method should behave just as it always has. On the other hand, since there's no explicit caller for *bar()*, it's return value will be lost since there's no obvious place to put it.

In either case, each time dojo.event.connect is called with the same arguments it will result in multiple connections. Later we will discuss strategies on how to guard against this.

Notice that dojo.event.connect takes a different number of arguments in the examples above. dojo.event.connect determines the types of positional arguments based on usage.Ã‚Â

The Dojo event system allows you to connect to DOM elements or nodes or plain JavaScript objects. The API is sophisticated enough that it allows you to connect multiple listeners to a single object so you can have multiple actions as a result of a single event such as a mouse click. Of course there is an API to disconnect the listeners too.Ã‚Â The Connecting the Pieces chapter describes the Dojo Event system in more detail.

Introduction to dojo.io.bind

At Dojo, we're committed to making DHTML applications usable, both for authors and for users, and with a lot of help from our friends, particularly Aaron Boodman and Mark Anderson, we have come up with solutions to the usability problems outlined above. We're providing it in a single, easy to use API and a package that requires only two files to function. The `dojo.io` package provides portable code for XMLHTTP and other, more complicated, transport mechanisms. Additionally, the "transports" that plug into it each provide their own logic to make each of them easier to use.

Most of the magic of the `dojo.io` package is exposed through the `bind()` method. `dojo.io.bind()` is a generic asynchronous request API that wraps multiple transport layers (queues of iframes, XMLHTTP, mod_pubsub, LivePage, etc.). Dojo attempts to pick the best available transport for

the request at hand, and in the provided package file, only XMLHTTP will ever be chosen since no other transports are rolled in. The API accepts a single anonymous object with known attributes of that object acting as function arguments. To make a request that returns raw text from a URL, you would call `bind()` like this:

```
dojo.io.bind({
    url: "http://foo.bar.com/sampleData.txt",
    load: function(type, data, evt){ /*do something w/ the data */ },
    mimetype: "text/plain"
});
```

That's all there is to it. You provide the location of the data you want to get and a callback function that you'd like to have called when you actually DO get the data. But what about if something goes wrong with the request? Just register an error handler too:

```
dojo.io.bind({
    url: "http://foo.bar.com/sampleData.txt",
    load: function(type, data, evt){ /*do something w/ the data */ },
    error: function(type, error){ /*do something w/ the error*/ },
    mimetype: "text/plain"
});
```

Errors and Timeouts

Regular web requests and Ajax requests with dojo.io.bind are much alike. Both use URL's and both use the HTTP protocol. But with browser requests, it is always clear to user when something goes wrong. You may get a 404 - Page Not ound, or a Server Unavailable, or at least something that says "Error". Ajax requests happen in the background, so when they error out the user won't know. Even worse, if the response *never* comes the browser may appear to lock up.

That's why it's extremely important to provide an error handler and a timeout handler with dojo.io.bind. *You should consider these as critical as URL or the load function*

At the very least, you should alert the user that something went wrong. Here's an example:

```
var kw = {
        url: "/cgi-bin/timeout.cgi",
        load: function(type, data, evt){
                document.myForm.myBox.value = data;
                dojo.byId("boxLoadTime").innerHTML = new Date();
        },
        error: function(type, data, evt){
                alert("Holy Bomb Box, Batman!  An error occurred: " + data);
        },
        timeoutSeconds: 2,
        timeout: function(type, data, evt){
                alert("I am tired of waiting.");
        }
};
```

The error() function takes the same arguments that load() does. But unlike load(), the only useful parameter is data, which contains the error message.

The timeoutSeconds and timeout parameters should be used together. timeoutSeconds defaults to 0, which means "wait forever". (In other Ajax libraries, this is called a synchronous request). 0 is not desirable. Even if you expect the request will take a long time, you should set a high value here

(e.g. 3600 = 1 hour), not 0. The timeout function takes the same arguments as error() and load(), but they are rarely consulted.

Hello Ajax World

So let's apply this to a trivial example. Suppose you have a text file on the web server with url my_message.txt.

```
Hello Ajax World!
```

You would like to load the file contents in an INPUT box without refreshing the page. Here's how:

```
<html>
<head>
<title>Insert title here</title>
<script type="text/javascript" src="/js/dojo/dojo.js"></script>
<script type="text/javascript">
    function loadRemotely(e) {
        var kw = {
            url:    "my_message.txt",
            load:    function(type, data, evt) {
                        document.myForm.myBox.value = data;
                        dojo.byId("boxLoadTime").innerHTML = new Date();
                    },
            method: "GET"
        };
        dojo.io.bind(kw);
    }
```

```
    function initAjax() {
        dojo.event.connect(dojo.byId("loadIt"), "onclick", "loadRemotely");
    }
    dojo.addOnLoad(initAjax);
</script>
</head>
<body>
    Form loaded at:
    <script type="text/javascript">document.write(new Date());</script>


    <form name="myForm">
    <input type="button" id="loadIt" value="Click here to load value.">


    <input type="text" name="myBox" size="50" />
    Text loaded at: <span id="boxLoadTime">N/A</span>
    </form>
</body>
</html>
```

Click the button and the value in my_message.txt automatically loads into the box. The date and time stamps on the page prove it does not reload when the user clicks the button.

This demonstrates the bare minimum you need for dojo.io to connect with Ajax. At the very least, you need an object of type dojo.io.Request. In our examples, this is the "kw" variable. dojo.io.Request needs the following:

- **url** is the URL containing content - usually generated on the server, and often using a

database
- **load(type, data, evt)** is the function called after the URL has been retrieved. This is an example of a *callback function.* It must have both parameters defined, although you don't need to read both of them. **data** is the most important, and contains the entire content retrieved from the URL. **type** is always "load". **evt** captures data about the load event.
- **method** is either "GET" or "POST". GET is common - we'll see an example of submitting a formful of data later on. That will require a POST.

There are many more optional parameters, and we'll see these in later examples.

Then just call dojo.io.bind with the dojo.io.Request variable. Here, bind means "connect this page with that URL and let things happen."

Why do all that for some static text? The answer is ... you won't. The URL is going to be a server-side program which returns content - mostly XML (the "X" in AJAX), but it could be text, HTML, or even binary data.

Sending Form Data

The Url of dojo.io.Request may contain parameters, like so:

```
url:  'myprogram.php?firstname=Chicken&lastname=Little&key=111111'
```

There are two problems: (1) it's difficult to URL encode everything, (2) it doesn't allow for dynamic parameters. The above works fine for everyone named Chicken Little, but ...

It's easier and more flexible to send an entire form of data. And you can do that with the formNode parameter of dojo.io.Request.

```
<script>
var kw = {
        url: "myprogram.php",
        load: function(type, data, evt){
                document.myForm.myBox.value = data;
        },
        error: function(type, data, evt){
                alert("Holy Bomb Box, Batman!  An error occurred: " + data);
        },
        timeoutSeconds: 2,
        timeout: function(type, data, evt){
                alert("I am tired of waiting.");
        }
        formNode: dojo.byId("myForm");
};
dojo.io.bind(kw);
</script>
```

```
<form id="myForm">
 <input type="hidden" name="key" value="111111" />
 <input type="text"  name="firstname" length="50" />
 <input type="text"  name="lastname" length="50" />

 <input type="text"  name="myBox" length="50" />
</form>
```

Modules, Resources, and Widget Namespaces

Modules

Dojo's code is split into logical units called modules. These are much like packages in Java, except that in Dojo a module can contain both constructors (like classes in Java) and simple functions.

For example, the "dojo.html" module contains a number of functions, such as dojo.html.getContentBox(). The "dojo.dnd" module contains a number of constructors for things like HtmlDragObject etc.

Note the naming convention - functions start with a lowercase letter, and constructors (which are technically functions but act more like classes) start with a capital letter.

Modules could be called "namespaces", except for the fact that "namespaces" has a different (but related) meaning w.r.t. widgets.

Resources

In the simple case, a Dojo module is defined in a single JavaScript file. But sometimes, a single module is split into multiple files.

For example, the dojo.html module, although originally defined in a single file, was getting too big, so we split into multiple files. This is for performance reasons, so that the browser only downloads the code it needs.

Unfortunately this implementation detail is not transparent to the Dojo user. You have to know which file contains the functions you need, and then include that file explicitly.

Each of these files is called a resource.

The line:

```
dojo.require("dojo.html.extras")
```

will include the file src/html/extras.js, which in turn defines some of the functions (but not all the functions) in the dojo.html module.

A single JavaScript file never defines multiple modules, although often a single file will define multiple constructors. In Java this would be equivalent to defining two classes in the same file.

All of this complication is for performance reasons, trying to balance

- not downloading more stuff than you need
- not downloading too many tiny files

Setting up Require Statments

How do you know what resources to dojo.require()?

1. modules

First, find out what modules you are using. In this example we'll assume you are using dojo.lfx.html.

2. resources

By reviewin the API doc you can see that dojo.lfx.html is defined in two files:

- src/lfx/html.js
- src/lfx/extras.js

Depending on what functions you are using, you will either do

```
dojo.require("dojo.lfx.html");
```

or

```
dojo.require("dojo.lfx.html");
dojo.require("dojo.lfx.extras");
```

Wildcards

There is a wild card include, such as dojo.lfx.* . New users might be surprised to learn that this may not necessarily include everything under lfx. Rather, there is an __package__.js file that defines what is included with the wild card, and this may also depend on the environment in which Dojo is loaded. For example, a browser will likely load different modules than Dojo would when loaded with Rhino on the command line.

dojo.provide()

Each file defining a resource should have (exactly) one line at the top defining the name of the resource.

Example:

```
   dojo.provide("dojo.html.extras")
```

For historical reasons, the dojo.provide() call serves two functions:

1. define the name of the resource (and register that the resource is loaded)
2. make sure that the resource's module exists (ex: make sure that dojo.html exists), so that statements like "dojo.html.foo = ..." don't fail with an error about dojo.html not existing.

Widget Namespaces

Widgets are combined into groups called namespaces. All the widgets built into Dojo are in the "dojo" namespace, but someone else could write their own widgets and put them in a different

namespace. For example, you could write your own button and checkbox widgets, and put them into an "acme" namespace. Then "acme:Button" would be your button, and would be unrelated to the button object built into dojo, called "dojo:Button".

Information on writing your own widgets occurs later in this book.

Widgets

When browsing through web sites and online applications, there are hundreds of widgets that come across your screen. Each button in your web browser is a widget. Each text entry box is a widget. We all know what a limited set of widgets that standard HTML provides: an input box, a button, a hyperlink.

Dojo widgets take an item like a text input box and adds functionality of a more user friendly object, like a graphical calendar to choose a date from. And it does this without breaking the original item on which the new functionality is built on.

The Widget Object

The first thing that you'll notice about widgets is that the are somewhat similar to a macro expansion, such as C's #define. Your source HTML is a simple

```
<button dojoType="Button" id="foo"> Click me </button>
```

and yet a pretty blue button shows up, and when you look at the generated DOM, it's a complicated tree of DOM nodes with a lot of absolute positioning and background images.

But, that's not all there is. For each widget, besides the visible manifestation, there's also a pure javascript object that manages that generated DOM tree.

In the above case, the generated javascript object is called, unsuprisingly, "foo". You can get it by doing:

```
var myButton = dojo.widget.byId("foo");
```

Turning Plain HTML Into Widgets

A Dojo widget wraps around your HTML. It looks at how the HTML is organized, what type of tags have been specified, what attributes they have, which tags are children of what other tags. All of these different variables allow a versatility in how skeletons are laid out, in what tags they use, in how the widget chooses to interpret them.

More Than One Way to...

You can think of a widget as the final form that covers a skeleton. The top widget layer reflects the

structure and functionality of the skeleton it sits on rather than covering it up.The simplest example of a skeleton is a single tag. A form input box, for example, is designed to simply accept a value.

```
<inputvalue="default">
```

But what happens when we want to help the user choose from an existing list of items? Enter the ComboBox widget.

```
<input dojoType="ComboBox" value="default">
```

As you can see, this is a functional skeleton. Not only is there an input box if the user does not have JavaScript enabled, but we can use the widget as part of a normal form. After all, we've only added on to it, we haven't changed the original purpose of the input element.

But this widget provides no data for the combo box to use.

```
<input dojoType="ComboBox" value="default" dataUrl="comboBoxData.js">
```

If you want to produce valid W3C HTML, you will have to use an alternative method to building your skeletons.Ã‚Â The dojoType attribute is not recognized by W3C and its validation tool will complain about it being their.Ã‚Â Below are two examples that do not use invalid attributes to build on your skeletons.

```
<input class="dojo-ComboBox" value="default" dataUrl="comboBoxData.js">
```

```
<dojo:ComboBox value="default" dataUrl="comboBoxData.js">
```

When there are certain attributes (dataUrl in this example), this widget will use that information to process the information to be contained in the combo box.

Sometimes you'll have skeletons defined in code and not even know it. In fact, many of the widgets provided by dojo assume that the underlying HTML is the same that would be encountered in every day life. Making this select element into a widget is as simple as adding the dojoType attribute.

```
<select dojoType="combobox">
  <option value="foo">foooption>
  <option value="bar">baroption>
  <option value="baz">bazoption>
  <option value="thud">thudoption>
select>
```

As you can see, you've just gained a whole lot of something for nothing. And, the data that will be used in the combo box is provided in a way that any web designer would understand.

TODO: this section needs to differentiate between ComboBox and Select.

Why Use Widgets?

You may ask yourself, "Why would I use wigets?" I honestly couldn't have answered this a few months ago, before finding Dojo anyway. The answer is really quite simple once you see how widgets improve the functionality and appearance of your web applications, without taking a long time to implement.

Enhanced User Experience

Widgets "enhance the user experience".Ã‚Â In layman's terms, that means that you can design web pages that are easier for people use, more quickly understandable, less error-prone, and flashier than web pages in plain html.

Easier to use - the Select widget for example, will narrow down the list of available choices based on keystrokes the user enters. That makes it faster to use than a normal HTML select box.

More quickly understandable - a web page with tabs will let the user easily navigate between different sections, and helps to make clear all the different sections of code that are on one page.

Less error prone - validation widgets will immediately notify users when they have entered an incorrect value, and/or automatically correct the value.

Flashier - dojo's menu code will fade in / out menus, or use some other effect, rather than a plain appear/disappear that you get with pure CSS menus

Faster Development

Widgets make it easy for web developers to add enhanced functionality. Here's why:

No Javascript Required

Web designers are generally very good with HTML. The really good ones are usually so involved in design that they don't even bother with learning the extra stuff that comes along with its dynamic aspect.

For these kinds of users, specifying widgets via HTML is a great solution. Not only is markup useful for being able to design with a placeholder element laying in wait, but many of the widgets actually analyze what they've laid out and use them as if properties were passed to the JavaScript object.

A great example of this is the tree widget. All that the designer has to do is lay out an HTML list, assign some attributes and they can have things going without having to touch a bit of code.

No worrying required

Widgets solve a bunch of issues (like cross-browser support) behind the scenes, so you don't have to worry it. See the next section for more detail about that.

Part 3: "The Dojo Programming Model"

Using Dojo to add dynamic capbilities to your web applications can be a little daunting at first.

Let's look at the programming model in more detail to better understand how to use Dojo to build some really cool apps. The programming model is object-oriented inspired and includes "classes" with methods and multi-level inheritance coupled with aspect-oriented event model famous in JavaScript. You will find the API doc quite useful to determine the methods and properties that are inherited from the parent "classes".

Thanks to Eugene Lazutkin and Bill Keese for help on this chapter.

Declarative vs Programmatic model

Dojo supports two programming models, declarative and programmatic. Which one you use depends on what you are doing. In most cases, the declarative model may be easier to use as it is markup but there are times when you will use the programmatic model. You can, of course, intermix the models on the same page as needed.

The best way to show the models is through the use of widgets which can be used either declaratively or programmatically. Although both models are available we will mainly use the declarative model through out this book when both are an option.

The tutorial used the declarative model to create a button on the page using the code below.

```
<BUTTON widgetId="helloButton" dojoType="Button">Hello World!</BUTTON>
```

The following declarative formats are equivalent.

```
        <?xml:namespace prefix = dojo /><dojo:widget></dojo:widget>

        <DIV dojoType="widget">

        <DIV class=dojo-widget></DIV></DIV>
```

We will discuss the declarative model in more detail shortly.

You can also declare widgets programmatically using the dojo.widget.create API as follows. When declaring widgets programmatically, the API returns the widget id which you use to call the appropriate methods.

```
var myTabPane= dojo.widget.createWidget("TabPane", {id: "myTabPane"}, srcDiv);
```

Which approach to use is discussed later in the book but for now we want to introduce the idea that you have different options.

Infrastructure

Before we get into the declarative model let's look at the widget infrastructure to better understand what methods are available when using widgets in your pages.

- Lifecycle methods are called by the infrastructure when a widget is created. They are not meant to be called by the user.
- Internal methods are called by the widget class code for supporting functionality and are

"private" and not meant to be called by the application developer. Unfortunately there is no easy way to identify the lifecycle or internal methods. We are working on conventions that will help identify these methods in future releases.

- "on" methods begin with the string "on" and are available to application developers. These are called by the widget infrastructure based on an action or event. Application developers can provide application specific code for these methods and those that application specific code will be called automatically when the related event or action is triggered.

- Developer methods are the rest of the methods in the widget. They are provided for use in the programmatic model.

In addtion to methods, each class has parameters that are useful. There are two types of parameters listed here.

- init parms are set at initialization time and then are readonly
- rest of the list should be in the API doc

Declarative model

dojoType instructs Dojo how to process the element when the page is loading. Keep in mind that Dojo manipulates the DOM as it renders the page so you must use the Dojo APIs to access the widget ids. Dojo keeps a reference of all widgets it has created that can be accessed with the `dojo.widget.byId` function - providing you specify either the `widgetId` or `id` attribute in your markup. Also you can use dojoAttachEvent using this method.

Namespace Details

In order to make namespaces practical and easy to use, Dojo has a concept of modules and resources. This concept was introduced in Part 2 Modules, Resources and Widget Namespaces when we first saw the HelloWorld tutorial. Resources are used to define a namespace and can be dynamically loaded on demand.

In order to load a resource you should request it using dojo.require(). It takes a string of text, which denotes a downloadable component. The content of this string is interpreted and a subject of Dojo conventions. First, it is traced to a single JavaScript file on disk (on web server). There are ways to affect this interpretation but out of the box it has a very sane behavior:

dojo.xxx => dojo/src/xxx.js

dojo.xxx.yyy => dojo/src/xxx/yyy.js

dojo.xxx.yyy.zzz => dojo/src/xxx/yyy/zzz.js

and so on. For example if you see a statement like that:

dojo.require("dojo.json");

It will load a file named dojo/src/json.js. If you don't know what it contains, you can go and look it up. To sum it up: a namespace hierarchy essentially reflects a file system hierarchy. By convention, if you see somebody using dojo.foo.bar.baz(), you can find it's definition in dojo/src/foo/bar.js, or, if it is not there, in one of files of dojo/src/foo/ (more on that case later), or in dojo/src/foo.js.

Dojo allows you to define your own resources and modules. For your custom resource you define your own top-level object. I will use "example" in my examples. If the Dojo loader sees that the first component is not "dojo", it applies following rule:

example.xxx => dojo/../example/xxx.js

example.xxx.yyy => dojo/../example/xxx/yyy.js

example.xxx.yyy.zzz => dojo/../example/xxx/yyy/zzz.js

Essentially it means that on your web server next to the "dojo" folder there is an "example" folder, which files are interpreted in the same way.

There is more information about the namespace and how it is used in the Widgets chapter.

Object Oriented concepts and inheritance

JavaScript is at its heart an object oriented language, but it is a prototype based object oriented language which does not have the same structure as class based languages like Java.This concept can be hard for new programmers who are not familiar with its construct. Dojo brings the object orientedness into a more familiar domain by modeling concepts that can be followed from Java and letting the toolkit handle the prototyping, inheritance and odd procedures JavaScript requires to make it work. Because of this, it not only allows people to get programming in object oriented JavaScript quicker, but it makes it faster to program because you can let the toolkit handle all of the odd procedures JavaScript requires to make it work. It all begins with a simple dojo.declare() function.

Simple declaration

Classes in Dojo are declared with a declare statement and assigning it a Class Name.Within the body can be variables, methods and constructors (know in Dojo as an initializer).

```
dojo.declare("ClassName",null, {
```

```
//class body
```

```
});
```

(Note: ClassName is the basic name, but to avoid naming conflicts, use package names like my.class.ClassName. For simplicity sake, we will start out with using just the simple name.)

Let's add some more content to our class by giving it a name and showing what the initilizer can do.Following is a persons class with an initializer and a moveToNewCity() function:

```
dojo.declare("Person", null, {
                //acts like a java constructor
                initializer: function(name, age, currentResidence){
                this.name=name;
                this.age=age;
                this.currentResidence=currentResidence;
        },
        moveToNewCity: function(newState)
        {
                this.currentResidence=newState;
        }

});
```

To create an object of this class you use the new keyword:

```
//create an instance of a new person

var matt= new Person('Matt', 25, 'New Mexico');
```

The initializer function is called once the object is created and the arguments are passed to it initializing the object.Our Matt object who is 25 currently lives in New Mexico, but let's say he moves a little further west to California.We can set his new currentResidence with the Person class method moveToNewCity(): matt.moveToNewCity('California');

Now the current value of matt.currentResidence shows that he now lives in California.

Inheritance

A person can only do so much, so let's create an Employee class that extends the Person class.The second argument in the dojo.declare() function is for extending subclasses.

```
dojo.declare("Employee",Person, {
        //acts like a constructor
        initializer:function(name, age, currentResidence, position)
        {
                Employee.superclass.initializer(name, age, currentResidence);
                this.password="";
                this.position=position;
        },

        login: function()
        {
                if(this.password!="" && this.password!=null){
                alert('you have successfully loged in with the password '+this.pas
        }
        else
        {
                alert('please ask the administrator for your password');
        }
}});
```

The first line in the initializer calls Employee.superclass.initializer, the Person class constructor. Dojo handles all of the requirements for setting up the inheritance chain.Methods or variables can be overridden by setting the name to the same as it is in the parent class. The Employee class can

override the Person class moveToNewCity(), perhaps by letting the company pay for moving expenses.

You initialize the sub class the same as the Person class with the new keyword.

```
var kathryn=new Employee(' Kathryn ', 26, 'Minnesota', 'Designer');
```

The Employee class passes the first three arguments down to the Person class, and sets the position.Kathryn has access to the login() function found in the Employee class, and also the moveToNewCity() function by calling kathryn.moveToNewCity(Ã¢â‚¬Ëœ Texas Ã¢â‚¬â„¢); Matt on the other hand, does not have access to the Employee login() function.

```
matt.login() // ERROR canÃ¢â‚¬â„¢t log in because he is not an Employee
```

Array/Object Declarations

If your class contains arrays or other complex objects, they should be declared in the initializer, due to some subtleties of object inheritance in javascript. Note that simple types (strings, numbers) are fine to declare in the class directly.

```
dojo.declare("my.classes.bar", my.classes.foo, {
        // coupledObjects: [1, 2, 3, 4] - doesn't do what I want;
        //                                ends up being like a static!!
        numItem : 5,        // one per bar
        strItem : "string", // one per bar

         initializer: function() {
                this.coupledObjects = [ ]; // each bar should have it's own array
                this.expensiveResource = new expensiveResource(); // one per bar
        }
});
```

Statics

On the other hand, if you want an object or array to be static (shared between all instances of my.classes.bar), then you should do something like this:

```
dojo.declare("my.classes.bar", my.classes.foo, {
        initializer: function() {
                dojo.debug("this is bar object # " + this.statics.counter++);
        },

        statics: { counter: 0, somethingElse: "hello" }
});
```

Mixins

The example below inherits from my.classes.foo and then mixes in "my.mixin". This is similar to multiple inheritance but there are some subtle differences, namely that this.inherited can only reference my.classes.foo, not my.mixin.

```
dojo.declare("my.classes.bar", [my.classes.foo, my.mixin], {
        initializer: function() {
                my.mixin.call(this /*, args*/); // invoke some mixin constructor
                // (note: my.mixin.prototype is ignored)
```

```
        },

        valueForPrototype: 3,

        methodForPrototype: function() {
        }

});
```

Design Notes

Constructor vs Initializer

In non-trivial cases, constructor definitions contain code. The constructor code performs initialization tasks and defines instance-only properties (properties whose values belong to a particular object, as opposed to prototype-properties which are shared by all objects using the prototype).

However, there is an issue with respect to constructor code in any simple JavaScript inheritence system. The constructor function of an object must be executed to create a prototype object for an inheritor. Therefore the constructor function must serve as both prototype-initializer and instance-initializer. The double duty of inherited constructors can be non-obvious and lead to subtle bugs.

Given the constructor above, when a bar object is created to use as a prototype, the mixin properties and properties created in the constructor become members of the inherited object's prototype. Almost always these properties are not intended to be part of a prototype.

As a practical matter, the extra prototypical properties are usually ignored as matching instance properties are created at object-instantiation time. However, for example, having an extra expensiveResource can be costly. And errors can result if the environment is not ready to create an expensiveResource at inherits-time. Errors caused by these conditions can be hard to track down, especially if the developer is not aware of how constructors are used when inheriting prototypes.

Separating instance-initializer tasks from prototype-initializer tasks eliminates these concerns. Therefore dojo.declare creates a standard, controlled constructor and separates instance-initialization tasks into a separate, optional initializer method.

Note: dojo.declare cannot inherit from an object that has a non-trivial constructor because dojo.declare does not allow constructors to also perform instance initialization.However, you can inherit from a dojo.declare created constructor without restriction

Calling Inherited (Ancestor) Methods

Sometimes one wants to invoke a method on an object from an ancestor prototype. JavaScript allows any function to call any other function in any context via the *call* and *apply* built-ins. So there are techniques like:

```
my.classes.bar.prototype.someMethod == function() {
        // invoke any function in our context
        anyFunction.call(this);
         // invoke inherited version of this method in our context
        my.classes.foo.someMethod.apply(this, arguments);
}
```

(Note: in these examples, the == indicates an assertion that the named property is equivalent to the function shown. Actual assignment is done via *dojo.lang.extend* or *dojo.declare*.)

As a convenience, *dojo.inherits* puts a reference to the ancestor prototype into the descendent constructor, and a reference to the descendent constructor into the descendent prototype. These extra references allow a great deal of extra flexibility in general, and also allow calling ancestor methods without explicitly naming the ancestor:

```
// invoke inherited version of this method in our context
        this.constructor.superclass.someMethod.apply(this, arguments);
```

However, the above technique will cause an infinte loop if someMethod is once removed. E.g., if we have *foo -> bar -> zot*, you can run into an issue like this:

```
 my.classes.foo.prototype.identify == function() {
   return "I'm a foo";
}
my.classes.bar.prototype.identify == function() {
        return "I'm a bar and " +
        this.constructor.superclass.identify.apply(this, arguments);
}
my.classes.zot.prototype.identify == function() {
        return "I'm a zot and " +
        this.constructor.superclass.identify.apply(this, arguments);
}
bar = new my.classes.bar();
alert(bar.identify()); // "I'm a bar and I'm a foo"
zot = new my.classes.zot();
alert(zot.identify()); // stack overflow
```

The error results because *this.constructor.superclass* referenced in bar's *identify* function refers to zot's superclass (causing *bar.identify* to call itself).

To resolve these issues, objects created from *dojo.declare* constructors include a function called *inherited* that safely invokes an ancestor method.

```
inherited: function(methodName /*string*/, arguments /* arrayLike */)
```

inherited correctly handles the problem scenario above:

```
my.classes.foo.prototype.identify == function() {
        return "I'm a foo";
}

my.classes.bar.prototype.identify == function() {
        return "I'm a bar and " + this.inherited('identify', arguments);
}

my.classes.zot.prototype.identify == function() {
        return "I'm a zot and " + this.inherited('identify', arguments);
}

bar = new my.classes.bar();
alert(bar.identify()); // "I'm a bar and I'm a foo"
zot = new my.classes.zot();
alert(zot.identify()); // "I'm a zot and I'm a bar and I'm a foo"
```

Syntax

The syntax is:

```
dojo.declare(className /*string */, superClass /*function*/
        [, initializer /* function*/]);
```

or

```
dojo.declare(className /*string */, [superClass /*function*/, mixin /* function *
        [, initializer /* function*/]);
```

Including the target *className* in the argument list allows the object path to be created automatically (i.e. intermediate namespaces are created as needed). Also, *dojo.declare* stores *className* in an eponymous property in the created object's prototype (e.g. *my.classes.foo.prototype.className == "my.classes.foo"*).

Naming

Technically speaking, JavaScript does not have classes: object construction is based on prototypes. For this reason reference to the term class has been (mostly) avoided above.

It seems that the difference is not of great practical importance. It's true that constructor functions in JavaScript are actual objects, but they operate like classes in the sense that they generally have no other purpose than as a mold for object instantiation. It is noted that classes are typically compile-time (or at least meta-) constructs and JavaScript constructors exist as Objects at runtime and contain actual data.

The name dojo.declare was chosen after much debate. The name is vague but easy to remember, read, and type. The method name inherited is lifted (at least) from ObjectPascal.Ã‚Â

Order matters

In general, a script should do the following in the ... section:

1. (Optional) set the djConfig options
2. Load the Dojo script
3. Call dojo.require(...) for all libraries used in the page
4. (Optional) define initialization functions and call addOnLoad

As in this example:

```
<!-- Step 1 (Optional) Set djConfig -->
<SCRIPT type=text/javascript>
        djConfig = {
          debug: true
        };
</SCRIPT>

<!-- Step 2: Load dojo -->

<SCRIPT src="js/dojo/dojo.js" type=text/javascript></SCRIPT>
<!-- Step 3: call dojo.require -->
<SCRIPT>
   dojo.require("dojo.book.myWidget.*");
   <!-- Step 4 (Optional): define initialization functions -->
   function initMyStuff() {
```

```
      ...
   }
   dojo.addOnLoad("initMyStuff");
</SCRIPT>
```

The order is important! If you do the steps out of order, dojo may not initialize properly, and your page will be a mess.

This script element is responsible for loading the base Dojo script that provides access to all the other Dojo functionality. Following this we add the requires statements which pulls in functionality needed by the application.

Use the dojo.addOnLoad to call functions which use the widget ids because Dojo must completely load the page and finish parsing the HTML before a reference can be made to the id. So, for example, the following will not work:

```
<BUTTON widgetId="helloButton" dojoType="Button">Hello World!</BUTTON>
<SCRIPT>
// ILLEGAL!!  helloButton does not exist yet
dojo.byId("helloButton").width2height = 0.5;
</SCRIPT>
```

Instead, place the script in an initialization function:

```
    <SCRIPT src="js/dojo/dojo.js" type=text/javascript></SCRIPT>
    <SCRIPT>
       function initMyStuff() {
          dojo.byId("helloButton").width2height = 0.5;
       }
       dojo.addOnLoad("initMyStuff");
    </SCRIPT>
<BUTTON widgetId="helloButton" dojoType="Button">Hello World!</BUTTON>
```

The global Dojo Objects

Dojo defines a global object called "dojo" which serves as an umbrella for everything Dojo-related. It simulates a namespace and was created to prevent clashes in the global JavaScript? namespace between the code in Dojo and other toolkits or user supplied code. Unfortunately it cannot be used during a bootstrap process, so special global variables should be used. All of them are prefixed with "dj".

You will need to use exactly two top-level Dojo-defined objects: "dojo", which serves as a namespace, and "djConfig", which is used to supply initialization parameters to Dojo, and should be created before Dojo's bootstrap.

For example, to turn off global widget searching, add these lines just *before* you include dojo.js:

```
<script type="text/javascript">
        djConfig = {
                parseWidgets: false
        };
</script>
```

Part 4: "More on Widgets"

P

Advanced ContentPane Usage

Introduction

A common use case for DHTML/ajax is to fetch a fragment of html using XHR or some other way, and change the innerHTML of a div with that content. Problem with this is that it doesn't instanciate widgets and doesn't fire scripts. ContentPane was created to make widgets and scripts work and reduce the potential for memory leaks. ContentPane is a base widget for many (Html) widgets, it handles remote loading as well as local setting of content and instanciating widgets in that content. Think of it as islands in your page that can easily switch content using setContent() or setUrl().

Many other widgets inherits ContentPane, like Tooltip, Dialog, FloatingPane etc. That means that all the methods and properties of ContentPane also applies to them.

ContentPane is often used as children of Layout widgets like LayoutContainer, TabContainer, AccordionContainer

Dont misstake it for a Iframe though, It should not be used on very large html fragments.

Usage

Simple usage ... Goto nextPage ...

Basic options

- **loadingMessage** Default: "Loading..." Set a custom loading message, see onDownloadStart to avoid showing this message completely
- **adjustPaths** Default: true When content is setUrl'ed from a different folder paths to images, links etc. is adjusted so the point to the correct dir
- **href** Default: "" Use this to grab initial content when contentpane is created.
- **extractContent** Default: true Only insert the html that is inside Script and style tags are not affected by this setting
- **parseContent** Default: true Create widgets inside content
- **cacheContent** Default: true Use dojo.io.bind javascript cache and, if it exists, browsers cache
- **preload** Default: false Lazyload switch, if true it will download content even if domNode is hidden **Note:** To make use of the default lazyload setting you need to hide your domNode Like this:
- **bindArgs** Default: {} Send in a custom setting to the dojo.io.bind call, like: mypane.bindArgs = {sync: true, preventCache: false}; mypane.seetUrl ('nextHtmlFragment.html');
- **refreshOnShow** Default: false Re-download content each time ContentPane is shown again
- **executeScripts** Default: false Fire scripts in content **Note:** see scriptSeparation
- **scriptSeparation** Default: true Run scripts in a separete scope for ContentPane **Note:** set to false if you want similar behaviour as a normal pageload
- **handler** Default: "" Java function name, generate pane content
- **isLoaded** Default: false Tells wheather we are loaded or not, see also: onLoad() and

addOnLoad()

Methods apart form those provided by ContentPane's superclass HtmlWidget

- **setContent(String or DomNode)** Use this instead of innerHTML
- **setUrl(String or dojo.uri.Uri)** Use this to set a new href and download and diplay that href
- **refresh()** Re-download and display href
- **loadContents()** Like refresh but only when isLoaded is false
- **setHandler(Function)** Set a function callback for javacontent generation
- **abort()** Abort a async download
- **addOnLoad(Object, "functionname" or Function)** Push a callback that will be run when content the next onLoad occurs. It's a fire and forget stack, if you want a callback each onLoad, see onLoad() Works for setContent as well
- **addOnUnload(Object, "functionname" or Function)** Same as addOnLoad but for onUnLoad event

Methods (Intended as event hooks using dojo.event.connect)

- **onLoad()** Called when everything rendered initialized and ready
- **onUnload()** Called before content is cleared
- **onDownloadStart(e)** preventDefault'able Called before a download occurs To prevent showing the loading message, do like this:
- **onDownloadEnd(url, string)** Called when download is completed, before it is setContent'ed
- **onDownloadError(e)** preventDefault'able Called when a load error occures, before The load error message is displayed. Prevent it the same way as onDownloadEnd **Tip:** During debug, you can display debug info like responseHeaders, responseText etc.
- **onContentError(e)** preventDefault'able Called when content insertion generates a error, before error mesage is displayed, like DOM faults, dojo.require() *syntax* faults etc.
- **onExecError(e)** Called when there is errors evaling script, doing java setContent and download errors of external scripts

In order to prevent the default messages you can do something like this:

```
<script>
        var myLoadMessage = {
                show: function(event){
                        event.preventDefault();
                        ... custom code here
                },
                hide: function(){...}
        }

        dojo.addOnLoad(function(){
                var pane = dojo.widget.byId('myPaneId');
                        dojo.event.connect(pane, "onDownloadStart", myLoadmes
        });
</script>
<div dojoType="ContentPane" id="myPaneId">...startcontent...</div>

or

<div dojoType="ContentPane"
        onDownloadStart="myLoadMessage.show(arguments[0]);">...startcontent.
```

When used as a child to TabContainer, AccordionContainer or PageContainer TabContainer, AccordionContainer or PageContainer extends Widgets with these extra options

- **label** Tab text
- **selected** Preselect this tab after creation
- **closable** Display close button**Note:**
- Height and width settings is done on the Container, not the ContentPane.
- In order for lazyload to work you have to hide your domNode initialy

When used as a child of LayoutContainer LayoutContainer package extends Widgets with this option

- **layoutAlign** "left", "right", "bottom", "top", or "client" see layout section of the book for more info

FAQ

- **Why doesn't my widgets show up?** Most likely you have used mypane.domNode.innerHTML = htmlstr; Use setContent instead: mypane.setContent (htmlstr);
- **Why doesn't lazy load work?** You have to hide your domNode, style="display:none;", initialy while creating ContentPane
- **ContentPane displays strange looking characters when loaded remotly in some browsers, why?** Like all server communication your browser need to know what charset your html is encoded with. Make sure your server is sending the correct Content-type header. example in php: header("Content-type: text/html; charset=utf-8"); Make sure you type utf-8 and not utf8, else IE will generate a warning and bail out.
- **Why is ContentPane so slow?** You probaly send it a big chunk of html with deeply nested tags.
  - Send it a html fragment, not a complete page with doctype and everything
  - Try to make the HTML simpler and use css for styling
  - Turn off the options you dont need: adjustPaths, extractContent, executeScripts, parseContent
  - Consider redesigning your page with serveral ContentPanes which grabs a smaller portion of your html that way you dont have to scan, render and create as many DomNodes/Widgets on each update.
  - Perhaps dojo.io.updateNode("nodeId", "myUrl") is all you need, and ContentPane is to heavy for your needs
- **dojo.addOnLoad() is called to early, before my Content is loaded** See onLoad event and addOnLoad for ContentPane, it is usualy easier to do this using <script>_container_.addOnLoad(..)</script> in your downloaded page, just be sure to set executeScripts=true
- **My inline scripts doesnt work when loaded in ContentPane, I have turned executeScripts to true? Short answer:** set scriptSeparation=false **Long answer:** ContentPane separates scope of scripts between different ContentPane's see: scriptScope page in dojo book
- **When I press submit in my form inside a ContentPane, the whole page unloads, why?** ContentPane doesn't have a form handling feature, look at dojo.widget.Form or see sample use case below

ContentPane examples

scriptScope

Executing scripts in ContentPane in dojo-0.3.1

This is a explanation about scripthandling in a ContentPane in other words when you set executeScripts to true, it is false by default.

ContentPane has some convenience functions related to scripts that makes life easier:

- .addOnLoad()
- .addOnUnLoad()
- And the replacement for scriptScope to dojo.widget.byId('thisWidgetId').scriptScope in html content attributes.

All scripts within content is evaled in the ContentPane property scriptScope, this means that you can have 2 or more content panes in the same page with the same name without risk of collision. This implementation does have its pros and cons. but correctly handled they will be useful. Also the scripts evals before widgets is parsed and after html is inserted.

So the content scripts is evaled inside a freestanding scope that inherits window. Take this function declaration.

```
<script>
    var i = 0;
    function addToI( j ){
        i = i + j;
        return i;
    }
</script>

becomes (from window scope):

(function( ){
    var i = 0;
    function addtoI( j ){
        i = i + j;
        return i;
    }
})
```

Due to the way javascript works addToI will be private and you cant reach it. Read Douglas Crockford excellent description of this. http://javascript.crockford.com/private.html

You can fix this in a couple of ways, one is to append it to global

```
<script>
    i = 0; // note lack of var
    addToI = function( j ){ // we could also do window.addToI = function( j ){ ..
        i = i + j;
        return i;
    }
</script>

becomes (from window scope)

i = 0;
function addtoI( j ){
    i = i + j;
    return i;
}
```

This is'nt recommended, if you want global scripts It is better to include them in the main page the old fashion way.

A better way would be to use Privileged functions:

```
<script>
    this.i = 0;
    this.addToI = function( j ){
        this.i = this.i + j;
        return this.i;
    }
</script>

becomes (from window scope)

(function(){
     this.i = 0; // now it is a property of this function and can be reached from
     this.addToI = function( j ){
        this.i = this.i + j;
        return this.i;
     }
})
```

As you might have guessed by now the (function(){...}) is the function scope that is reference held by ContentPane.scriptScope.

So to call the addToI function from the outside we can do:

```
var added = dojo.widget.byId('myPaneId').scriptScope.addToI( 10 );
dojo.debug(added) // prints 10

added = dojo.widget.byId('myPaneId').scriptScope.addToI( 10 );
dojo.debug(added); // prints 20

and so on...
```

Now lets say we have html that would like to alert the value of i (this.i) in plain html that would be:

```
<button onclick="alert(i);">Tell me i !</button>
// As explained above this wont work in ContentPane(unless you set it to global by
```

Now if we now the ID of the contentPane that pulls in this content we could do:

```
<button onclick="alert(dojo.widget.byId('myPaneId').scriptScope.i);">Tell me i !<
// this will work in ContentPane
```

That is'nt very useful as we don't always know the ID of the contentPane that pulls in the html when we write the content.

ContentPane has a convenience replacer function the scans the html and replaces all occurrences of the keyword scriptScope in html attributes. So to achieve the above:

```
<button onclick=" scriptScope.i">Tell me i !</button>
// this will work in ContentPane
// NOTE: Due to a bug in ContentPane 0.3.1 you need to add a extra space before th
// Thank you Sasha Firsov for finding that!
```

The parent scope of scriptScope is window, the reason for that is to avoid messing with widget internals. Just imagine the disaster a redefinition of setUrl function would cause otherwise.

To enable the content scripts to talk to the containing ContentPane there is set a private variable on scriptScope construction.

That is the _container_ variable. Lets say we have a widget in our content that we would like to connect a event callback on:

```
... some content
<div dojoType="DatePicker" id="myPicker"></div>
... rest of content
```

A content script could look like:

```
<script>
    var o = {
        storeDate: function( ){
            var datePick = dojo.widget.byId('myPicker');
            var date = datePick.storedDate;
            // save date somewhere
        }
    };
    _container_.addOnLoad(function(){
        var picker = dojo.widget.byId('myPicker');
        dojo.event.connect(picker, "onSetDate", o, "storeDate");
    });

    // remember to disconnect onUnLoad, very important!!
    _container_.addOnUnLoad(function(){
```

```
        var picker = dojo.widget.byId('myPicker');
        dojo.event.disconnect(picker, "onSetDate", o, "storeDate");
    });
</script>
```

When the content is cleared in ContentPane the scriptScope is unreferenced, this means that if there are no other variables that holds reference to any of the scriptScope objects (like event connects, varible connection etc), the script will be garbage collected by the javascript engine (memory freed).

Different browsers are more or less conservative about GC (garbage collect), IE and Mozilla beeing the more relaxed browsers, khtml and Presto (Opera) engines are more conservative.

Some usecase samples

Lets say you have page that you need to run in as both stand alone and within a ContentPane.

Then you can do something similar to this.

Courtesy of Sasha Firsov for a pointer to this example!

```
<html>
<head>
<script>
    var djConfig = {isDebug: true};
</script>
<script src="dojo/dojo.js"></script>
<script>
    var scriptScope = this;
    if(typeof _container_ == 'undefined'){
        var _container_ = dojo;
    }

    _container_.addOnLoad(function(){
        dojo.debug("Successfully loaded!");
    });

    this.doWhenClicked = function(txt){
        dojo.debug(txt);
    }
</script>
<body>
    <a href="javascript:scriptScope.doWhenClicked('You clicked a link!');">Click h
</body>
</html>
```

Perhaps you want to prevent all <a href='...' link clicks with a ContentPane from clearing your page, and use the href to set your client ContentPane.

```
*********Your mainpage***********
<html>
<head>
<script src="dojo/dojo.js"></script>
<script>
    dojo.require("dojo.widget.ContentPane");
    dojo.require("dojo.widget.LayoutContainer");

    function changeUrlInClient(url){
        var client = dojo.widget.byId("client");
        client.setUrl(url);
    }
</script>
</head>
<body>
    <div dojoType="LayoutContainer" layoutChildPriority='none' style="border: 1px
        <div dojoType="ContentPane" layoutAlign="left" style="width: 200px;" execu
        <div widgetId="client" dojoType="ContentPane" layoutAlign="client" style=
    </div>
</body>
</html>

*******linkpage.html************
<html>
<head>
<script>
    var o = {
        listen: function(evt){
            // if the onclick came from a
</head>
<body>
    <a href="content1.html">content1</a>

    <a href="content2.html">content2</a>

    <a href="content3.html">content3</a>

    <a href="content4.html">content4</a>
</body>
</html>
```

A simple example of using a form in ContentPane. NOTE! dont use this login example in real world applications, password is sent in cleartext

```
**********mainpage************
<html>
<head>
<script src="dojo/dojo.js"></script>
<script>
    dojo.require("dojo.widget.FloatingPane");
    dojo.require("dojo.widget.Button");
</script>
</head>
<body>
    <div dojoType="FloatingPane"
        title="Login example"
        style="width: 300px; height: 300px;"
        executeScripts="true"
        cacheContent="false"
        href="login.php">
```

```
    </div>
</body>
</html>

*********login.php**********
<?php
    session_start();

    // are we trying to login?
    if(isset($_GET["login"])){
        // this could of be a database instead
        $users = array(
                "JohnDoe"=>
                    array("pass"=>"foo", "id"=>1),
                "JaneDoe"=>
                    array("pass"=>"bar", "id"=>2),
                "JuniorDoe"=>
                    array("pass"=>"baz", "id"=>3)
                );

        if(isset($_POST["user"]) && isset($_POST["pass"])){
            $pass = $_POST["pass"];
            $user = $_POST["user"];
            if(isset($users[$user]) && ($users[$user]["pass"] == $pass)){
                $_SESSION["id"] = $users[$user]["id"];
                exit("(true);");
            }
        }

        //if we get here we have failed to login
        exit("(false);");
    }

    // logout?
    if(isset($_GET["logout"])){
        unset($_SESSION["id"]);
    }

    if(isset($_SESSION["id"])){
        // it is safe to show secret content
?>

    <script type="text/javascript">
        this.logout = function(){
            _container_.setUrl("login.php?logout=true");
        }
    </script>
    <h3>You have successfully logged in!</h3>
    showing secret content here

    <a href="#" onclick="scriptScope.logout();">log out</a>

<?php
    }else{
    //no it wasnt safe, show our login script
?>
    <script type="text/javascript">
        this.ok = function(){
            _container_.domNode.style.cursor = "wait";
            dojo.io.bind({
                formNode: dojo.byId("login"),
                mimetype: "text/javascript",
                handler: function(type, data){dojo.debug(data);
                    _container_.domNode.style.cursor = "";
                    if(type=="load"){
                        if(data){
                            _container_.setUrl("login.php");
```

```
                        }else{
                            dojo.byId("message").innerHTML = "Wrong username or pa
                        }
                    }else{
                        dojo.byId("message").innerHTML = "An error occured while
                    }
                }
            });
        }

        this.quit = function(){
            _container_.hide();
        }
    </script>
    <form name="login" id="login" method="post" action="login.php?login=true">
        <div id="message" style="text-align:center; color: red;">You need to logi
        <label for="user">Username:
        <input type="text" name="user"/>


        <label for="pass">Password:</label>
        <input type="password" name="pass"/>

        <button dojoType="Button" onClick="scriptScope.ok();"/>login</button>
        <button dojoType="Button" onClick="scriptScope.quit();">quit</button>
    </form>

<?php
    }
?>
```

In the follwing scenario you wont need executeScripts.

In fact it wont affect the script at all, the script will run just as any other regular script in ordinary page

```
<html>
<head>
    <script src="dojo/dojo.js"></script>
    <script>
        dojo.require("dojo.widget.ContentPane");
    </script>
</head>
<body>
    <div dojoType="ContentPane" >
        <script>
            // this script will fire before dojo makes our parent a ContentPane wi
            // affected by executeScripts at all.
            // i wont have a _container_ variable and scriptScope wont hide any va
            // it will work just as a inlne javascript block always has

            alert("This alert will fire event if you have executeScripts=false");
        </script>
    </div>
</body>
</html>
```

It should be fairly easy to pull in some small customized scripts that is tweaked to the content, like a form validation script or a button callback.

The executeScripts and scriptScope has the potential to be very usefull, I cant think of all the possible implementations but probably you can.

Catches:

- If you event.connect to _container_ be sure to disconnect onUnLoad, else you get all sorts of strange errors
- Be sure to unref. all references into and out of scriptScope before setting new content, else there will be a memleak
- <img src="images/flatScreen.gif" width="32" height="32"> big

Like HTML buttons, the dojo button sizes to fit its content. Usually, you will provide an onclick="..." attribute to specify what happens when the button is pressed.

This needs to be rewritten for 0.9

Using Your Own Backgrounds

By default, dojo uses a blue gradient background. But you can provide your own. You will need to create three .gif images: one for the left, one for the right, and one for the center. The filenames must end with l, r, or c, respectively. You can specify image sets for four different conditions:

- activeImg - the mouse pointer is over the button
- inactiveImg - the mouse pointer is not over the button
- pressedImg - the button is being pressed
- disabledImg - the button cannot be pressed

For example, you can use these files:

- /images/buttons/disabled-l.gif
- /images/buttons/disabled-r.gif
- /images/buttons/disabled-c.gif

as the disabled image of your button like this:

```
<button dojoType="Button" disabledImg="/images/buttons/disabled" >
   Quit
</button>
```

API Reference: dojo.widget.Button

See Also: DropDownButton, comboButton

comboButton

Used in HTML Element:button

A combination Button and DropDownButton. Use this for a button that has a common action (e.g. "Make Regular Dinner") and less common related actions (e.g. "Make Romantic Dinner" and "Make TV Dinner")

Example

```
<button dojoType="comboButton" menuId='saveMenu'>
   <img src="images/editIcon.gif" width="32" height="32">
   Save
</button>

<div dojoType="PopupMenu2" id="editMenu" toggle="wipe">
        <div dojoType="MenuItem2" iconSrc="images/save.gif" caption="Save" accelK
        <div dojoType="MenuItem2" iconSrc="images/saveAs.gif" caption="Save As...
</div>
```

You can also specify your own background images, as in Button.

API Reference: dojo.widget.ComboButton

See Also: Button, DropDownButton, PopupMenu2, MenuItem2

Editor2 (RichText) Widget

Introduction

Editor2 Widget in dojo provides a WYSIWYG editor for HTML content. The core is compact and lightweight, while a plugin framework ensures that any functionality can be achieved by plugins.

Basic html editing capacity is implemented in the core, which is the RichText widget. Currently keyboard shortcuts are also hardcoded in this widget (TODO: generalize this, or use KeyRouter instead?).

Editor2 is a subclass of RichText Widget which adds a toolbar (Editor2Toolbar Widget) to the top of the editing area.

Basic Principles of Editor2

In order to have an extensible structure, the new Editor2 introduced several new concepts.

Command (dojo.widget.Editor2Command)

The first and most fundamental one is call Command, which executes a specific function on the

editing area. It also provides the API to retrieve the current state of the command. The base class for Command is dojo.widget.Editor2Command (defined in Editor2.js).

Each command should have a unique name and each command is a singleton object per page: no matter how many Editor2 instances there are in one page, they share the same command objects.

Toolbar Item (dojo.widget.Editor2ToolbarButton)

The toolbar (defined in Editor2Toolbar.js) for the editor2 contains serveral toolbar items. The basic class for toolbar item is dojo.widget.Editor2ToolbarButton (defined in Editor2Toolbar.js), which essentially is a simplified version of dojo widget.

Toolbar item can be of any type, besides buttons, you can have more complex items, such as a dojo combobox like item with a dropdown (see dojo.widget.Editor2ToolbarFormatBlockSelect in Editor2Toolbar.js).

Available Plugins

All the builtin plugins are located under src/widget/Editor2Plugin directory. Those files ending with Dialog are the actual popups. The table below lists all the other plugins:

Name	**Description**	**Features**		
ContextMenu	**Command**	**ToolbarItem**		
ContextMenu	Context Menu Core, with menu items for builtin commands	Cut/Copy/Paste, Link/Unlink, Image properties	-	-
FindReplace	Implement find and replace functionalities	-	Find/Replace	Find/Replace
TableOperation	Support for table related operation	Insert/Delete Table	Insert/Delete Table	Insert Table
AlwaysShowToolbar	Ensure the toolbar is visible when scrolling the page	-	-	-
ToolbarDndSupport	Toolbar Set/Item drag and drop support	-	-	-
SimpleSignalCommands	Add simple signals to Editor2, such as save() and createLink()	-	-	-

Misc

SetupCopyPasteForFirefox - Copy, Cut and Paste are disabled by default in Mozilla/Firefox, this tip is how to enable it for your trusted websites.

Form Widgets

There are many widgets used for forms:

- Button - just like HTML's button, except with a few advanced features
- Checkbox - like HTML's checkbox but in soria (blue) theme
- ComboBox - like a text input field, but w/suggested values
- DropDownDatePicker - for specifying a date by selecting a cell of a calendar
- DropDownTimePicker - for specifying a time (scheduled for 0.4 release)
- Editor2 (RichText) - like HTML's textarea, but allows editing of rich text

- HslColorPicker - pick a color
- Select - just like HTML [select] element, except w/autocompletion, and loading of possible values from a remote data source
- Slider - graphical slider control used to specify a number within a range
- Spinner - numeric input field that can be adjusted up/down by pressing arrow keys
- dojo.widget.*Validate - a bunch of widgets that check the user's input and correct it or print an error message if it doesn't conform to a certain format

The main principle of these widgets is that:

- each widget corresponds to a native HTML element.
- each widget (w/the exception of Button) represents a single input value
- each widget has a (possibly hidden) _____ element, to which it serializes its input value, so that form submission (either normal submission or via FormBind) works as expected

All these widgets should have these attributes just like native HTML input elements.Ã‚Â You can set them during widget construction, but after that they are read only:

- disabled
- tabIndex
- name
- value

And they also share some common methods:

- disable()/enable()
- onValueChanged() - called with the new value of the widget whenever it's changed
- setValue()Ã‚Â (note: you can get the value by accessing accessing value; exception: Editor)

(note: some widgets don't conform but we plan to convert them soon)

ComboBox Widget

The ComboBox widget is a text input box that supplies a list of possible preexisting values for the user to choose from. It can query the server for an updated list of values as the user types, allowing it to offer a large list of values without requiring the entire list to be downloaded to the browser.

To demonstrate the power of this widget, let's dive right in and make an autocompleting combo box that queries the server for the list of matches for what the user has typed so far. We'll declare the widget using HTML:

```
<select dojoType="ComboBox"
    autoComplete="true"
    dataUrl="/suggest.php?match=%{searchString}"
    maxListLength="15"
    mode="remote"
    name="myComboBox">
```

There are a few attributes of note here.

- **autoComplete** should be set to "true" if you want the widget to fill in the rest of the input box with the contents of the first item in the suggestion list. For example, if the user has entered "Ala" and the first suggestion on the list is "Alabama", the widget will put "bama" after the letters the user has typed (selected, so the user can simply continue typing to replace them with something else.)
- **dataUrl** is the location of a URL which will be queried each time the user types something into the box. It will be discussed in more detail below.
- **maxListLength** is the number of suggestions that will be visible to the user at one time. If the server supplies more suggestions than this, the user will have to scroll the list of suggestions to see them.
- **mode** is one of "local" (the default; the **dataUrl** is fetched once at load time), "remote" (the **dataUrl** is fetched on each keypress, and is expected to return a JSON object; see below) or "html" (the **dataUrl** is fetched on each keypress, and is expected to return HTML.)

When the page containing that widget is loaded, it gets rendered as a text entry box with a little dropdown list button on the right side, not as a normal HTML <select>.

When you enter text into the box, Dojo tries to find matches for the text you've just entered. For example, suppose you type "a". Because the widget's **mode** is set to "remote", it will fetch the **dataUrl** and substitute your input for the magic **%{searchString}** token. (That token is only valid when **mode** is set to "remote" or "html".) In this case, it will fetch `/suggest.php?match=a` from the server. You don't have to use PHP on the server side, of course; it's simply used as an example here. The point is that the widget will replace the magic token in dataUrl with the user's input and fetch the resulting URL from the server.

What should the server return? In the "remote" mode, the widget expects a JSON array of entries, each entry of which contains a displayable option name and a value. For example, the server might return something like

```
[
    [ "Alabama", "AL" ],
    [ "Alaska", "AK" ],
    [ "Arkansas", "AR" ]
]
```

Many server-side programming languages have existing libraries to output native objects in JSON form. In this case, for simplicity's sake, we'll do it by hand. Here's what an extremely simple, inefficient suggest.php might look like.

```
<?php
$states = array( "Alabama" => "AL",
                 "Alaska" => "AK",
                 ...
               );
$userInput = $_GET['match'];
$result = "[";
foreach ($states as $state => $abbreviation) {
```

```
    if (strpos($state, $userInput) === 0) {
        $result = $result . '[ "' . $state . '", "' .
                  $abbreviation . '"],';
    }
}
$result = $result . ']';
print $result;
?>
```

FormBind

`FormBind` allows you to quickly setup your Ã¢â‚¬Å“Web 1.0Ã¢â‚¬Â³ form for asynchronous submission. Basically it sets things up so that whenever the user hits the submit button, rather than submitting the form in the usual way, and refreshing the entire page, the contents are sent over xmlhttp (or any transport), and then the results are passed to the given callback.

How do you do it? Easy:

```
function magicForm() {
  var x = new dojo.io.FormBind({
    // reference your form
    formNode: document.forms[1],

    load: function(load, data, e) {
      // what to do when the form finishes
      // for example, populate a DIV:
      dojo.byId('myDiv').innerHTML = data;
    }
  });
}

dojo.addOnLoad(magicForm);
```

Note the unfortunate naming between dojo.io.bind() and dojo.io.FormBind.

dojo.io.bind() is a function that immediately sends the given info to the specified URL. (It would probably better be called something like dojo.io.send() but it isn't.)

dojo.io.FormBind(), on the other hand, doesn't send anything to the server. It just hooks up events so that *when the user presses the submit button* then the data is sent via dojo.io.bind(). Also note that you call "new" to make it work.

Note also that although dojo.io.bind() also takes a formNode argument, it's tricky to use and you are better off using FormBind. That's because for forms containing the Editor/Editor2 widgets, they need to serialize their data back to the [textarea] before the form is submitted, and that only happens when the form's onSumbit handler is called. Just calling dojo.io.bind() and specifying a formNode won't do that. However, with FormBind (and with an actual [input type="submit] button in in the form), everything works perfectly.

You can play with the demo to see it in action.

Validation

There are three methods which you can use to validate your form data on the client side before it is sent to the server - with each having their own benefits and drawbacks. Often, the most effective validation is performed using a combination of these methods.

*It is important to understand that whilst client side validation is effective, it should not be considered a replacement of server side validation techniques. In order to provide an enjoyable and secure user experience it is **essential** that a combination of both methods is used.*

Manual processing of input fields

The dojo.validate.* module provides a number of functions for validating user input. Currently, these functions are broken into the following groups:

- common
- datetime
- de
- jp
- us
- web

Common dojo.validate functions

dojo.validate.isText

This function takes two arguments - a value, and an optional flags object. Depending on the properties of the flags object (*length*, *minlength*, or *maxlength*), the value can be tested for exact length, a minimum length, or a maximum length.

dojo.validate.isInteger

This function takes two arguments - a value, and an optional flags object. Depending on the properties of the flags object (*signed* or *separator*), the value can be tested for the presence of a sign (+ or -) character at it's beginning, or whether it has separators. Whilst there is no default separator, single characters (such as ',') or arrays listing multiple separators can be specified.

dojo.validate.isRealNumber

dojo.validate.isCurrency

dojo.validate.isInRange

dojo.validate.isNumberFormat

TODO: summary of each function group and functions within each group

Using dojo.validate.check on a form

The second method of validation is known as dojo.validate.check. This function let's you setup a table of rules for checking a form's input elements.

TODO: more info on this

Validation widgets

There are several widgets in the dojo.widget.validate module that will either correct user input (converting lowercase to uppercase, etc.), or print errors when the input doesn't match a certain pattern. A few of the widgets are:

- dojo.widget.validate.IntegerTextbox - allow only integer input
- dojo.widget.validate.UsZipTextbox - entering a US zip code
- dojo.widget.validate.UsPhoneNumberTextbox - entering a US phone number

Unfortunately currently the validation widgets, although they do display an error message alongside illegal values, do not actually prevent the form from submitting. This needs to be addressed at some point. You need to do some javascript coding yourself to make this happen.

Graphical widgets

The alternative to validating user input is to provide such an interface that the user can't enter a bogus value to begin with. For example, the DatePicker widget won't let the user input an invalid date.

Interacting With Widgets

Calling Methods

Previously I talked about the widget object, that you can get access to like this:

```
var myButton = dojo.widget.byId("foo");
```

If you create a widget programatically you automatically get a pointer to the widget object:

```
var myButton = dojo.widget.CreateWidget("Button", {caption: "click me"});
```

What is myButton useful for? Calling methods on the button. For example:

```
myButton.setCaption("Don't press me!!");
```

Note that doing the following **won't** work, because the myButton object doesn't know that the caption variable has been changed:

```
myButton.caption="this won't do anything";
```

Also note that to disable/enable a widget, call disable()/enable(), rather than setting the disabled attribute directly. People often make that mistake.

Read only Variables

There are some read-only variables, however, that are useful to access. Two of the most important ones are:

- domNode - points to the node that replaced your original markup (the [button] tag in the example above)
- containerNode - points to the node that contains the contents of the original markup ("Click me" in the example above)

That reminds me. In the above example of programmatic creation, you also need a line like this:

```
form1.appendChild(myButton.domNode);
```

Events

Consider the markup below:

```
<button dojoType="Button" onClick="alert('hello world')">
```

It looks familiar, but it's actually quite different than the normal onclick handler on the dom node.

onClick() is a method in the Button widget object. It's got a similar name to DOM node's onclick (but not identical; there's a capitalization difference). However, it's not the same. As another example, consider

```
<input type="Slider"

        onValueChanged="alert('new value is ' + arguments[0]);">...
```

In this case, we are using a function of the widget called onValueChanged(newValue), that has no direct equivalent in the dom world.

Attaching vs. Overriding

in the case above, the specified code will be run in addition to the widget's original

onValueChanged() method. It works the same way as dojo.event.connect(). On the other hand, if you just specify a function name like this:

```
<input type="Slider"

        onValueChanged="doit">...
```

Then you are overriding the widget's onValueChanged() funtion w/your own.

Usually, the widget will provide an empty function stub, so it won't matter if you connect to it or override it.

Using dojo.event.connect directly

You can also do something like this, although it seems more difficult than the method above:

```
dojo.event.connect(myButton, onValueChanged, function(x){

        alert("new val is " + x);

});
```

Show and Hide

Widgets all can be hidden (made invisible) and shown:

- myButton.show() - display
- myButton.hide() - make invisible
- myButton.toggleShowing() - switches between show() and hide()
- isShowing() - is widget currently displayed?

For show and hide, there are 4 transitions available, that you set at widget creation time:

- plain
- fade
- wipe
- explode

They are set like this:

```
<div dojoType="FloatingPane" toggle="fade" toggleDuration="250">
```

The explosion effect (often used for tooltips) also requires a point/square from which the element explodes out of, or implodes back into. This is set automatically when using the Toggler or TaskBar widgets.

Layout

Introduction

There are two philosophies to laying out the screen. One way, the "web-way", says that everything should flow naturally from HTML, meaning basically that a bunch of stuff is in the document and

if your window isn't big enough, then you use the browser's scrollbar. This is the way traditional web pages work, and is the best choice for many applications.

There other philosophy is to take the available size of the viewport (basically, the browser window), and then to partition it into smaller and smaller pieces. If you think about a mail application that splits the screen into top/left/right sections, then you are thinking about this kind of design.

The Layout Widgets

Dojo provides a number of widgets for implementing the second design listed above. They fall into two basic categories.

Widgets that split the screen space between a set of widgets

- LayoutContainer- lets you position the children into top/left/bottom/right positions, with the specified center piece taking all the remaining space
- SplitContainer- shows children either horizontally or vertically aligned, and you can adjust the relative size of each child by moving the divider bars between the widgets

Widgets that hold mulitple children but only display one at a time:

- TabContainer - names of children are printed as tab labels
- AccordionContainer - stacks children vertically, and you can show one at a time
- WizardContainer - go through the children in an ordered fashion like a wizard

In addition, there's one widget that isn't a layout widget per se, but it is often used with the layout widgets:

- ContentPane - like a div, but it's a widget, and it can load its contents from a specified href.

These widgets can be nested to arbitrary levels, so that you could have a LayoutContainer with a top/bottom/client section, where the client section is a SplitContainer, and that SplitContainer could contain a TabContainer, which would itself contain a LayoutContainer, and so on.

The leaf nodes of this hierarchy could be any non-layout node, but often are ContentPane nodes.

Example

Example (currently not displaying correctly. wiki needs upgrade?!):

<DIV> <DIV>hello world </DIV> <DIV> <DIV>left side of split </DIV> <DIV> <DIV>second tab </DIV> <DIV>i'm on the bottom </DIV> </DIV>

```
<DIV dojoType="LayoutContainer" >
        <DIV dojoType="ContentPane"> hello world </DIV>
        <DIV dojoType="SplitContainer">
                <DIV dojoType="ContentPane"> left side of split </DIV>
                <DIV dojoType="TabContainer">
                        <div dojoType="LayoutContainer">
                         ..
                        </DIV>
                        <DIV dojoType="ContentPane"> second tab </DIV>
                 </DIV>
          </DIV>
         <DIV dojoType="ContentPane"> i'm on the bottom </DIV>
```

```
</div>
```

Note that all these objects are called containers because they just contain a set of other objects; they don't contain mixed content (text and nodes) like a normal <DIV>.

Also note that there is no "LayoutContainerChild" or "SplitPaneContainerChild" like node. That's to reduce the amount of markup and code required to setup a deep hierarchy of layout widgets.

Parameters

Note that the example above is missing some important parameters. For one thing, it doesn't specify whether the SplitContainer arranges its children vertically or horizontally. For that we need:

```
    <DIV dojoType="SplitContainer" orientation="horizontal">

We are also missing the labels for each of the tabs in the TabContainer, which we
    <DIV dojoType="TabContainer">
      <DIV dojoType="LayoutContainer" label="Tab 1">
       ..
      </DIV>
      <DIV dojoType="ContentPane" label="Tab 2"> second tab </DIV>
    </DIV></DIV>
```

Note that the labels are specified as parameters to the ContentPane and LayoutContainer, the children of the TabContainer, rather than as arguments to the TabContainer itself. "label" is not technically a property on those two objects, but you can still specify it,and the TabContainer will pick it up.

Similarly, for a LayoutContainer, you need to say where each child should be located:

```
        <DIV dojoType="LayoutContainer">
        <DIV dojoType="ContentPane" layoutAlign="top"> hello world </DIV>
        <DIV dojoType="SplitContainer" layoutAlign="client">...</DIV>
        <DIV dojoType="ContentPane" layoutAlign="bottom"> i'm on the bottom </DIV>
        </DIV>
```

You may freely mix sides (top, bottom, left, right) in a layout container. Sides are used from the outside in. The special side name "client" will fill in any part of the container that is not otherwise occupied. Very often you will use fixed-size side panes and a client pane that grows and shrinks as the user resizes the window, for example:

```
<DIV style="OVERFLOW: hidden; WIDTH: 100%; HEIGHT: 100%" dojoType="LayoutContainer

  <DIV dojoType="ContentPane" layoutAlign="top" height="2em">

    Page header goes here; it stretches across the whole width of the window.

  </DIV>

  <DIV dojoType="ContentPane" layoutAlign="bottom" height="1em">

    And a footer here, also stretching across the whole width.

  </DIV>

  <DIV style="WIDTH: 120px" dojoType="ContentPane" layoutAlign="left">
```

```
    Some left-side navigation HTML, bounded by the header and footer
    since they were already added to the layout.
  </DIV>
  <DIV style="WIDTH: 60px" dojoType="ContentPane" layoutAlign="right">
    Some right-side navigation HTML
  </DIV>
  <DIV dojoType="ContentPane" layoutAlign="client">
    Main page body here, bounded by all the fixed-size elements above.
  </DIV>
</DIV>
```

Sizing

For the top level layout container in a hierarchy, you need to specify a size. If you don't, the contents of the container may be displayed oddly or not at all.

```
<DIV style="WIDTH: 500px; HEIGHT: 500px" dojoType="LayoutContainer"></DIV>
```

Many web applications will want to fill the whole screen with their top level layout container. Think of a case like a mail application. For any size browser window, you want the top part to have some menu choices, and then have the bottom part be split between a tree on the left and message list/message on the right.

In this case, you need CSS like this:

```
html, body, #mainWindow {
 width: 100%;
 height: 100%;
 overflow: hidden;
}
```

And then inside your tag you will have something like:

```
<DIV id=mainWindow dojoType="LayoutContainer"></DIV>
```

Programmatic creation

Creating a hierarchy of layout widgets programatically works the same way as normal programatic creation, except that sizing info needs to be specified in a special way.

```
// make a dummy div just to specify size
```

```
var div = document.createElement("div");
with(div.style){ height="500px"; width="500px"; }

// create the layout container
var lc = dojo.widget.createWidget("LayoutContainer", null, div);

// add some children for top, bottom, and center.  Top and Bottom
// children also need to have a size specified, and possibly a scrollbar
var topDiv = document.createElement("div");
with(topDiv.style){ height="30px"; overflow="auto"; }
lc.addChild( dojo.widget.createWidget("ContentPane", { href: "foo/bar.html", layou
var bottomDiv = document.createElement("div");
with(bottomDiv.style){ height="30px"; overflow="auto"; }
lc.addChild( dojo.widget.createWidget("ContentPane", { href: "foo/bar.html", layou
```

One other thing to note in this example is that each ContentPane has two parameters. The href parameter applies to the ContentPane itself, but the layoutAlign parameter is really something that the LayoutContainer processes.

Doing your own positioning

You also have the option to lay stuff out on the screen like LayoutContainer does, but without using LayoutContainer. There's a function called dojo.html.layout() that will position a bunch of elements just like LayoutContainer does. (Actually LayoutContainer calls this function.)

Multiple Renderers

Different browsers have different capabilities when it comes to displaying (rendering) your widget. Dojo provides mechanisms that automatically detect which of these capabilities the browser offers and extends your widget using the most powerful rendering system the widget has code for.

We'll discuss how these mechanisms work, when you should use them, and how to extend widgets to support multiple renderers.

Defining And Extending Widgets with Multiple Renderers

The section discusses how to code a widget that supports multiple renderers, and how to extend such a widget.

How to support multiple renderers

```
// renderer-agnostic portion
dojo.declare("my.widget.Foo"
{
  initializer: function() {
    // do initialization tasks, make instance properties
  },
  foo: 5,
  doit: function() { ... },
  ...
}
);
// render-specific portion
dojo.widget.defineWidget("my.widget.html.Foo", [ dojo.widget.HtmlWidget, my.
```

```
    initializer: function() {
      // do initialization tasks, make instance properties
    },
     ...prototypical properties (in object notation)...
  }
);
dojo.widget.defineWidget("my.widget.svg.Foo", [ dojo.widget.SvgWidget, my.wi
    initializer: function() {
      // do initialization tasks, make instance properties
    },
     ...prototypical properties (in object notation)...
  }
);
```

Subclassing from multiple renderers

```
// renderer-agnostic portion
// add features to my.widget.Foo, but don't explicitly extend or inherit
// my.widget.Foo properties will come in as part of my.widget.[html|svg].Foo
  // do initialization tasks, make instance properties
dojo.declare("my.widget.FooPlus", my.widget.Foo, { ... });
dojo.widget.defineWidget("my.widget.html.FooPlus", [my.widget.html.Foo, my.widget
   initializer: function() {
     // do initialization tasks, make instance properties
   },
    ...prototypical properties (in object notation)...
  }
);
dojo.widget.defineWidget("my.widget.svg.FooPlus", [my.widget.svg.Foo, my.widget.F
   initializer: function() {
     // do initialization tasks, make instance properties
   },
    ...prototypical properties (in object notation)...
  }
);
```

Understanding The Widget Hierarchy

Before you can write your own widget, you should understand how dojo's widgets are organized, both in terms of where files are and how the class hierarchy works.

Renderer Base Classes

The first thing to notice is the following renderer base classes.

```
Widget
   |-- DomWidget
      |-- HtmlWidget
      |-- SvgWidget
```

Each widget implementation will extend either HtmlWidget?, SvgWidget?, or VmlWidget?, according to what browser it supports.

Class Hierarchy

For widgets w/only a single implementation (usually "html"), the class hierarchy is pretty simple. For example, there is a dojo.widget.html.Button that extends HtmlWidget.

However, widgets w/multiple implementations are more complicated, because there's a base class that defines common functionality and the parameter list for the widget. This effectively leads to multiple-inheritance, since the widget implementation to pull in stuff from both from the renderer base class and the widget base class. For example, dojo.widget.svg.Chart needs to effectively inherit from both SvgWidget and from dojo.widget.Chart.

Technically, multiple implementations are defined using mixins, which are similar (but subtly different) than multiple-inheritance. In the above case, dojo.widget.svg.Chart extends HtmlWidget but mixes in dojo.widget.Chart base class.

Directory Structure

For widgets w/a single implementation, like Button:

- src/widget/Button.js - defines dojo.widget.html.Button

For widgets w/multiple implementations, like Chart (in the future):

- src/widget/Chart.js - defines dojo.widget.Chart base class
- src/widget/svg/Chart.js - dojo.widget.svg.Chart, svg implementation
- src/widget/vml/Chart.js - dojo.widget.vml.Chart, vml implementation

Understanding Widget Renderers

Implementations

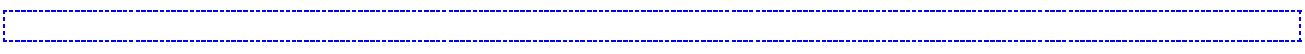
Widgets can have (but are not required to have) multiple implementations, as follows:

- svg - will run on any svg enabled browser (FF, later safari, soon other browsers)
- vml - will run on IE
- html - can run on any browser

Note that the so-called "html" version of the widget might actually run special code for IE, FF, etc., either through calls to utility functions (such as the graphics library) that branch based on browser version, or "if/else" statements, or whatever.

Which widget gets run?

The HTML file just specifies the widget name, without specifying the implementation. For example,



Dojo will pick which version of the widget to run based on the user's browser and what versions of the widget are available. For example, on IE, it will run dojo.widget.vml.Foo if it exists, and otherwise run dojo.widget.html.Foo.

Modules for Implementations

There are three separate modules, corresponding to the implementations above:

- dojo.widget.svg
- dojo.widget.vml
- dojo.widget.html

Examples:

1. The Button widget only has a single "html" implementation. It's defined in dojo.widget.html.Button

2. In the future, the Chart widget will have both "svg" and "vml" implementations, defined in dojo.widget.svg.Chart and dojo.widget.vml.Chart. (The widget doesn't instantiate at all on browsers that don't match either svg or vml)

Why Decouple Code?

Having an object without any code to display it can be a strange idea for many people. "After all," they say, "I'll only be using my widget in an HTML environment." Such a kneejerk reaction is understandable, as many people view the decoupling of code as an effort not worth the time.

What you end up with is a method that does some logic, does some rendering, does some more logic, in a fairly long loop. Splitting these processes up is merely saying, "Let's get all of our business logic out of the way, and then we can draw the results." Even if you won't be splitting your widget into multiple files, as we'll be discussing shortly, this should still be done. The next time something isn't displaying correctly, you won't have to wade through business logic to find the problem. The next time business logic isn't working correctly, you won't have to search through display-specific code. And the most important part, you won't worry about modifying business logic or display-specific code breaking the other piece.

What you should end up with is a plain old JavaScript object that doesn't know about how it will be drawn, and doesn't care. It's the guts of your widget, and can be run in the console, in an SVG environment, in a standard HTML environment, or anywhere that Dojo currently supports or will support in the future.

When a widget is loaded, it has a lifecycle that runs calls several methods. These methods are pretty clearly separated into business logic and display-specific methods. For example, mixInProperties is business logic and setWidth is display-specific. There is no need for these methods to even interact with each other, and splitting these between multiple files make it easier to locate one from the other.

You should also end up with most reusable code. Instead of loading external data in the same method as display-specific code, you can move it to your main widget object and call it from the display-specific code. Then, when another method needs to use the same information, it's ready for you to use.

Navigation

Need help updating this page. Describe the navigation widgets in Dojo including attributes that are common to all these widgets.

Menu2

PopupMenu

ProgressBar

ToolBar

Tree2

FishEye

Tree widget

Introduction

This documentation refers to 3rd major version of the tree widget, sometimes refered to as TreeV3.

Many mentioned classes (e.g TreeLoadingController) have V3 on the end, but that suffix is sometimes omitted, because it will be removed in dojo 0.5.

If there exist 2 same classes, but one with V3 at the end - it's the one you need.

Examples are given in tests (dojo/tests/widget/treeV3), so you might want to check them first and copy-paste exactly the things you need.

Please browse the Book,

then ask questions in dojo-interest list

If you feel the question private

or want to contribute

IRC: Freenode, #dojo by nick [algo]

ICQ: 820317

"Ilia Kantor" ilia @ dojotoolkit.org

Extensions

Extensions are also called plugins, they can be hooked onto widgets in various combinations and provide wanted options.

Currently there is a couple of extensions

TreeDisableWrapExtension

Tree extension, disables wrapping for tree nodes. Also it fixes IE bug when an 'unwrappable' node (e.g single word) will move to next line if no space left.

TreeDocIconExtension

Tree extension, places icon to the left of a node, depening on nodeType property

TreeEmphaseOnSelect

Selector extension, highlights currently selected nodes

TreeDeselectOnDblselect

Selector extension, deselects a selected node when it is clicked. Usually, one should ctrl-click, or click another node.

TreeLinkExtension

Tree extension, turns labels into links, merges object property into tag

Faq

How to make tree unselectable?

To make tree (or its elements) unselectable use dojo.html.disableSelection in nodeCreate and treeCreate hooks. Apply disableSelection to every node you want to make unselectable.

How to bind an object to tree node?

There is an "objectId" property and "object" property ready to be filled in from markup or program-way.

How to walk all node descendants ?

You may use dojo.lang.forEach(nodeOrTree.getDescendants(),function(elem) { ... }) to process all descendants, it will walk children property recursively.

The safer way would be to call TreeCommon.prototype.processDescendants(nodeOrTree, filter, func), it will process all children with func, but will not descend into nodes if filter(node) returns false. E.g see collapseAll controller method uses it to collapse all widgets, but skip non-folders and data objects.

How to evade a situation where all nodes are (re)moved and tree is empty without a way to add new child (no nodes) ?

Make a single root node with actionsDisabled="DETACH;MOVE". User will be unable to remove it, so interface will stay sane.

Also, you may want to set actionsDisabled="ADDCHILD" to tree itself, so now children can be added besides the root.

How to create a custom tree node ?

First, of course, you may explicitly use createSimple for your widget and declare your widgetType in markup.

But sometimes, tree has to create a node from data object or just from "nothing", e.g in case of createAndEdit.

Then it checks for *widgetName* property of data object (can be namespaced), and if no *widgetName*, then *tree.defaultChildWidget* property should contain node class, e.g mycustom.tree.Node.

Usually, when you override a node, all you need is to adjust *defaultChildWidget*,

because *widgetName* uses generic create and hence works slower right now.

How to make pages open when a user clicks on node?

There are 2 ways. The first one is to attach TreeSelector and hook on "select" event. So when a user clicks, event handler will change url to node.object.href. Of course, you should fill hrefs.

A probably more convinient path would be to employ TreeLinkExtension, which will turn your labelNodes into real links, and apply attrbutes from node object to them.

I open very large tree. But navigation away to another page from the tree takes time. What's up?

Dojo performs actions not only when a node is created, but also cleanup when a node is destroyed. Lazy features allow node creation be distributed in time, but when you navigate away from a large tree, large cleanup causes visible delay. I don't know a way to evade that.

How to add icons to nodes ?

TreeDocIconExtension handles that. You should declare nodeType for your nodes, so they'll get nodeIcon[Your type] CSS class. Default type is Document for leaves and Folder for folders.

There is also *setNodeTypeClass* method to update node CSS when its *nodeType* changes e.g programmatically.

Introduction

Introduction

This documentation refers to 3rd major version of the tree widget, sometimes refered to as TreeV3.

Many mentioned classes (e.g TreeLoadingController) have V3 on the end, but that suffix is sometimes omitted, because it will be removed in dojo 0.5.

If there exist 2 same classes, but one with V3 at the end - it's the one you need.

Examples are given in tests (dojo/tests/widget/treeV3), so you might want to check them first and copy-paste exactly the things you need.

Please browse the Book,

then ask questions in dojo-interest list

If you feel the question private

IRC: Freenode, #dojo by nick [algo]

ICQ: 820317

"Ilia Kantor" ilia @ dojotoolkit.org

Features

Features

Flexible styling

- All design in CSS through classes and class combinations
- Different trees be styled with different CSS class families
- Multiline and rich content support

Full set of node operations

- expand/collapse
- create with JS or markup
- destroy/move/clone
- addChild/detach/(de)folderize
- inline editing
- multiple selection and drag'n'drop
- keyboard controls

Performance

- batch operations
- special features
- profiled and optimized

Dynamic node loading & RPC features

- rich API

- callbacks & errbacks
- suited to be in-sync with data
- locking

Event system

- publish

- hook on any tree change

Customization

- change everything through inheritance, events and css
- out-of-the box extensions
- coded with it in mind

Tests and demos

Tree overall structure

Note: most classes here omitt 'V3' suffix

Model + View

The tree itself is a TreeV3 class instance. Hierarchy is maintained in a standard widgety way: through children[] array. Children are usually TreeNodeV3 instances, but you could use your own(overriding?) implementation of course.

TreeV3 instance also represents an 'invisible root' node, so it shares common methods with TreeNodeV3. These methods reside in TreeWithNode mixin.

Model contains data and manipulation methods like "addChild", "detach" .. etc.Ã‚Â It also publishes events when modified.

DOM-structure and view is also merged into model.

Various functionality can be hooked on model's events: controller, menu, drag'n'drop etc.

Model events should help you to integrate tree with application on data-level, so you hook on actual data changes, not the cause (program call, user click etc).

Controller

Main controllers are TreeBasicController -> TreeLoadingController -> TreeRpcController

Basically, they are responsible for operating on model and performing most logic, besides model's action.Ã‚Â It also makes checks / remote calls.

Usually, one should work with controller only and let it process model.

TreeLoadingController and TreeRpcController are known to perform remote calls to server. They use dojo.Deferred and dojo.DeferredList for thatÃ‚Â purpose.

Most customizations are also about controller.

And, by the way, model has no idea about its controller... It throws events and delivers API to call, that's all.

Extensions

The stuff is loosely coupled, so a bunch of extensions can be hooked on events too

What's new in TreeV3

New HTML/CSS structure

Nested divs

Previous tree used a list of divs, each of them was indented with grid and spacers to right level. The new tree uses natural nested divs structure (children' divs inside parent's div). Grid is contigous and structure is displayed correctly for any node/font size

All design in CSS through classes and class combinations

All image and size information was removed from JS code. There is a bunch of classes applied to nodes, that may denote node folder state, node type, show if there are children, etc. CSS

moves this logical classes into style

Different trees be styled with different CSS class families

Want to put 2 differently styled trees on a page? Give them different classPrefix.

Multiline content support

Rich content support was incomplete, because list-of-divs model could not handle arbitrary-sized nodes. Now you may have
,

and any other width/height

modifiers.

Event system modified

nodeDOMCreated event was removed. That's because listeners are bound to tree and may want to modify the new node, but that's only possible when the node is being bound to the tree, not when it was created and hanging around. afterTreeChange was introduced to help listeners to (un)bind nodes the right moment.

All events were renamed to better reflect the moment of their publishing.

afterExpand, afterCollapse events now fire when the animation (e.g fading in or out) finishes, not when the actual expand/collapse is called.

Lazy widget creation

Before TreeV3, all nodes must be widgets. A node is added - hence graphical widget is created. For performance reasons that behavior was altered. Now when you add a node, you may actually add a "data object", containing node data, e.g {title:"new node"}. You may want to add a large nested branch of such data objects, like {title:"new", children:[...data objects..]}.

Data objects will become real members of children array (you may recursively search them, modify etc), but graphical widgets will be created only when visitor expands them.

The compatibility drawback of such behavior is that old code may erroneously call **widget** methods on **data objects** while recursively traversing a tree, e.g with Widget#getDescendants. You should change such code to use TreeCommon#processDescendants, or handle data objects in special way.

There are no special mechanisms to add laziliy instantiated "data objects". You may manipulate them simply modifying children array, but no events are thrown until a real widget appears on the scene. In most cases that is fine, but you are free to "disable" lazy widget creation - do not modify children directly and enable tree.eagerWidgetInstantiation

Tree extensions

- Many features were moved from core into extensions
- Added TreeDocIconExtension instead of builtin childIcon support
- Selector now only throws events, not doing anything with nodes
- Out-of-the box extensions introduced to be examples and handle well-known

requirements

Implicit helpers removed

The Tree is actually a pack of loosely coupled components, connected through events. To keep things simple and also for compatibility reasons, such components(controller,selector...) were created implicitly, if not declared. But actually this proved to be a source of questions and misunderstandings. So now nothing is created implicitly, read how-to and declare things.

RPC has both sync/async modes

Old callbacks code was removed in favor to dojo.Deferred. Now all operations may be async and run your callbacks at the end.

Drag'n'drop changes

Multiple selection and multiple drag'n'drop (incomplete)

Sounds simple enough.. Select multiple nodes with ctrl and get them with selector.selectedNodes. instead of removed selectorNode call.

Currently, multiple drag'n'drop does not work with multiple selection because of dojo bugs. Hopefully will be fixed.

Drop of any source, not just tree node

If treeNode property is empty, tree will create a new node from the data returned by source.getTreeNode, then source.onDrop will be called to remove old node.

Inline node editing

It became possible to edit nodes inline, using TreeEditor. Base variant uses RichText widget, you can make another wrapper though. Remote calls can be made on save only, or on start/cancel too e.g for locking purposes.

Node creation

There are few code paths that lead to same purpose: to create a tree node. They differ in effeciency and use patterns

Markup creation

You specity a tree and its nodes in HTML, relying upon dojo to parse it and turn into widgets. That is a slowest way, but nice for small trees or if only tree top is specified and the rest is created later.

dojo widget parser walks DOM and creates a special structure. The next pass creates widgets from the structure.

Widget#create

The generic widget creation routine. It basically runs the operations in order:

- Mix in widget properties from parameters/markup
- Register widget in widget.Manager
- Call buildRendering to make fill template and create domNode
- Call initialize
- Call postInitialize. registers widget as a child of its parent and after it creates all subwidgetsCall postCreate

Note that initialize is called in pre-order: parent is initialized before children, postInitialize is called in post-order: a child is postCreated before its parent.

Manual creation

If you create nodes with javascript, then you run create calls manyally. So parents are naturally created (and postCreated) before children.

There seem to be no good way to distinguish betwen markup creation and manual creation. From the one hand its seems good, because allows reuse of generic creation code. From the other hand code paths going through this code are subtly different.

The reliable thing is that initialize will process widget after its domNode is built, BUT it should not assume anything about children.

afterChangeTree event is fired on initialization also. If you want to know anything about children and do something at this point - check addChild, but not node creation.

Input parameters

children array may be

- empty
- contain widgets, e.g if created from markup, or someone created them before parent and pushed in
- contain data objects, that will be turned into widgets when parent expands.
- isFolder comes into play only when there are no children. It allows creation of empty folders, with UNCHECKED state that can be filled later.

Performance

Tree was coded with performance in mind. Although, JavaScript itself is a slow language. Flexible model requires some code that slows it down. It's not DOM manipulations, but actually javascript that I couldn't make lighter. Being a part of dojo/widget structure implies some overhead, but also power.

Almost all operations require small constant time when single node is involved. Depending on your application you may notice slowdown when (most common) creating lots of nodes or performing other batch operations.

Creation from markup or with standard create/addChild routines is 2-3 times slower, because these routines are generic.

Comparison

Fast node creation with dojo tree is 2-3 times slower than xtree 1.7, another tree widget, not so featured, but nicely optimized for performance.

Important

The results described here refer to operations without any lazy features involved. Most of time you will use lazy creation or lazy loading, or both, and operate with thousands of "virtual" nodes with ease.

Performance Tricks

When talking about performance, one should understand, that there are single-node operations that operate on single node... These ones are fast. The examples are: create a node, delete a node, move a node along the tree.

... And there are batch operations that touch a lot of nodes. The examples are: initial tree creation, moving a node from one tree to another which has different listeners, etc.

That performance issues become noticeable at 100-300 tree nodes depending on your trees. All algorithms are linear in worst case, but JS is slow language, DOM is also not that fast.

There is a number of features one could use to get a speedup.

Lazy loading

A node can be created with isFolder=true flag, but without children. Any node has a state, initially UNCHECKED for empty folder, and used by TreeLoadingController.

When a user presses expand, tree controller (supporting lazy loading) will send a request to server asking for nodes, and parse the answer creating children.

The benefit is obvious: you don't have to load/process whole tree at once. You can only load a single node and user will load the rest clicking "expand"

Lazy creation

Node/tree keeps array of its children in children property. Lazy creation is somewhat a half-way approach to lazy loading. It allows you to put data objects into this array and tree will create widgets of them later, when they are expanded.

For instance, one can call node.children = [{title:'node1'},{title:'node2'}]. The objects will be set, but no widgets are created. You can also set children to nested array: node.children = [{title:'node1', children:[{title:'node2'}] }].

You can create tree on server, JSON-serialize it and put to HTML, that is gzip-compressed. Compression will be 6 times or more, so it is not that space hungry.

The benefit comes from postponing almost all real job: widget creation and attaching it to tree will happen in expansion-time.

Comparison between lazy creation and lazy loading

- You need web-service for lazy loading, not for lazy creation

- No network waits for lazy creation
- Lazy creation gives you the tree right here. You can search data objects and modify them without spending time and memory on graphical widgets

Sometimes, lazy creation and loading may work together nicely, providing seamless increase in speed and decrease in memory footprint. For instance, server may pass a whole tree branch in JSON to lazy loading controller. Top nodes will be created right along, because user needs them, but the rest of the branch will be postponed relying on lazy creation feature.

There are operations, like "expandAll" where such lazy tricks don't help, because all graphical widgets must be processed. That is why widget creation process is well-optimized itself. createSimple is a hacky program-only way to create TreeNodes fast. setChildren is a method to assign (and create if needed) all children at once. It helps to evade some extra work happening when children are added one by one.

IE image-reloading fixup (!!!)

IE has a well-known bug. If an image was loaded dynamically - with a new Image(), or img.src= assignment, or even as a background of a new node, it will not be cached. So every time when you create a node, all needed icons get loaded from server (or requested at least). A possible solution is to put a special div into HTML (adjust src to your path):

Ã‚Â Ã‚Â

Ã‚Â Ã‚Â

Ã‚Â Ã‚Â

Ã‚Â Ã‚Â

Ã‚Â Ã‚Â

Ã‚Â Ã‚Â

Ã‚Â Ã‚Â

Ã‚Â Ã‚Â

Ã‚Â Ã‚Â

Ã‚Â Ã‚Â

Server communication

To talk with server, one should use TreeLoadingControllerV3 or TreeRpcControllerV3. They inherit from TreeBasicControllerV3 and override its methods to deliver remote calls possibility.

TreeLoadingControllerV3 contains main methods for server calls, and allows dynamic node loading. TreeRpcControllerV3 adds server calls to **tree manupulations** like "createChild/move/edit...".

Url settings

All requests go through dojo.io.bind, usually via XMLHttpRequest transport.

- RpcUrl

contains basic Url for all requests, e.g "http://site.com/remoteTreeService.do". You can have query string in it also.

- RpcActionParam

every call adds special *action* parameter to query string to distinguish between call types. Actions are *move*, *createChild*..

For children loading, the action is *getChildren*.

An example url for such action would be "http://site.com/remoteTreeService.do?action=getChildren".

Most actions imply additional *data* parameter with information about node/tree and other action details server may want to know.

This way of composing an url is described in *getRpcUrl*, feel free to override if need.

Request format

data parameter is JSON-serialized. It usually sends some information about involved nodes and position. If you want to extend it somehow,

1. override method of controller that corresponds your action, your changes will affect this action only,
2. override *getInfo* method of node/tree to affect parameters globally
3. override getInfo method of controller if that's the right place =)

Response format

All data is JSON-serialized. There are libraries for JSON in most of programming languages.

Server response is **evaluated as javascript**. That means you can embed any javascript, that will be evaluated on client-side. But it should **return object**. Use object *error* property to signalize about server-side error.

Good answer:

dojo.debug('I can also put javascript in server answer');

([{title:"test",isFolder:true,objectId:"myobj"},{title:"test2",children:[{title:"test2.1"}Ã‚Â]}])

Good answer:

({})

Good answer, will return dojo.RpcError

({error: "Permission denied"})

Bad answer format (string), will return dojo.FormatError

Exception: blabla at line 50

Transport error (e.g 404) will also return dojo.CommunicationError

If you don't know what to return, return *({})*. That means just "ok". Note outer brackets, they are needed to make sure it evaluates to javascript Object.

Callbacks and Error handling

Any request may be performed in synchroneous and asynchronous manner.

Both of them return dojo.Deferred object, but for synchroneous call, it will be called until next script line.

You can call deferred.addCallback / deferred.addErrback to add your actions.

An example of usage would be

var deferred = loadingController.expandAll(tree);

```
// add action when operation finishes successfully

deferred.addCallback(function() { alert('expanded all nodes!'); });


// process error

deferred.addErrback(function(err) { dojo.debugShallow(err); });
```

More information about Deferred class and asynchronous programming can be found at http://mochikit.com/doc/html/MochiKit/Async.html (dojo implementation is Mochikit port),Ã‚Â http://twistedmatrix.com/projects/core/documentation/howto/async.html (python implementation and a nice state-of-art intro).

Tree Events

There are many classes of events, published with dojo.event.publish mechanism. Every event has a name and message object, containing more precise information about what happened. You may use events to update your data while tree changes, and to perform additional processing of involved objects.

There is a default naming scheme for an event class. E.g for a tree with widgetId='mytree', event of class afterTreeCreate will be named "mytree/afterTreeCreate". You may provide other names in eventNames property of the tree.

afterTreeCreate

Event occurs after tree creation is complete. There is an alternative to hook on this action by putting your objects in "listeners" property of the tree. The difference is that listeners are guaranteed to hook before nodes get added, and afterTreeCreate is published after Tree widget is created.

source

references to tree

beforeTreeDestroy

Published right before actual Tree#destroy method is called. Useful for cleanups

source

references to tree

beforeNodeDestroy

Right before TreeNode#destroy is called. Node is detached after this event fired.

source

references to node

afterChangeTree

This event is tightly created with node creation process. It is fired when

- a node is created
    - no parent at this stage
    - fires in initialize(), so children may be not added yet
- a node was moved to another tree widget

oldTree

references previous tree, null if node has been just created

newTree

new(current) tree

node

target node

afterSetFolder

Fires when a node obtains "folder" state. That may happen when a first child is added to a leaf, or if a node was initially created with isFolder=true

source

references to node

afterUnsetFolder

Fires when a node obtains looses "folder" state. That may happen when a last child leaves the node, and Tree.unsetFolderOnEmptyis set, or when unsetFolder is called explicitly.

source

references to node

(before|after)Move(From|To)

These events share same arguments and fire when a node is moved. Move process is considered something special. When you move a node, no detach/addChild events get thrown. That allows to tell situations when a node leaves a tree for some time (detached then attached) from situations when a node is simply moved to another location

oldParent

previous parent

oldTree

previous tree

oldIndex

previous index among siblings

newParent

new parent

newTree

new tree

newIndex

new index among siblings

child

target node

afterAddChild

Published when a node is attached to parent. This may occur at the end of creation process, or when a node is lazily instantiated from data object.

Also it occurs when a detached node gets attached.

child

references to node

index

index among siblings

parent

current parent who adopted a child

childWidgetCreated

flag is set if child was laziliy instantiated. That is: it resided as data object in children array, but user expanded its parent, so node widget came to life.

afterDetach

Occurs when a node is detached. This may happen in the process of node destruction. Keep in mind, that detaching a node sets its parent to null, but

tree remains same.

child

references to node

parent

references to old parent

index

references to index among children of old parent

after(Expand|Collapse)

Fire when a node is expanded/collapsed. Some togglers do nice animation hiding/showing node. This event fires when animation finishes.

source

target node

afterSetTitle

When a node is edited, or explicit setTitle method is called, this event helps to inform interested parts about changes.

source

target node

oldTitle

replaced node title

Tree HTML/CSS model

There are few major approaches to building dynamic trees.

1. list of idented divs

Each tree node is a div with indentation. Indentation is e.g 20px * node depth, so everything looks fine. Usually indentation is made of many quadrantic images, each of them represents empty space or grid lines, which visibly link nodes together.nested divs.

Of course, 'div' can be changed to any tag, e.g 'li'.

2. nested divs

Divs are nested same way tree nodes are nested. Can use ul/li instead of divs, there's only symantic difference, of course, if styles are same.

Each div can be idented relatively to its parent with padding/margin property, or with images.

If we use images here, then there will be lots of extra tags, so padding/margin seems better.

Dojo tree adapts the 2nd approach, of course, with padding/margin identation.

Let's consider a simple tree

* Node1

* Node 1.1

* Node 1.2

* Node 2

(Page is unfinished, and content will probably be merged into the Trees intro page -- CAR)

Trees

The trees we see in User Interfaces help sort out long, heirarchical lists. A file system is the classic example, with Windows using it in Explorer and Macintoshes with Finder (is it still called that???).

Nodes are the basis of a dojo tree. A node can include other nodes, and is then called a branch, container or folder. A node containing no other nodes is a *leaf*. Dojo does not force you to distinguish branches from leaves. It deduces the tree structure from your own code.

A dojo tree contains at least two dojo widgets:

- A surrounding *Tree* widget
- Embedded *TreeNode* widgets

But there are many dojo widgets to help you sculpt, mold, and connect behavior to your tree.

1 Hello Tree World

Here's a simple example.

```
<div dojoType="Tree" >
  <div dojoType="TreeNode" title="Item 1">
    <div dojoType="TreeNode" title="Item 1.1" />
      <div dojoType="TreeNode" title="Item 1.2" >
        <div dojoType="TreeNode" title="Item 1.2.1" >
        <div dojoType="TreeNode" title="Item 1.2.1.1" />
      </div>
      <div dojoType="TreeNode" title="Item 1.2.2" />
    </div>
    <div dojoType="TreeNode" title="Item 1.3" />
  </div>
  <div dojoType="TreeNode" title="Item 2" />
</div>
```

Which produces the following lovely tree:

SCREENSHOT

You can do open a node and show its contents by clicking the + icon, or hide them with the - icon, just like you're used to. Nice!

2 Connecting an Action to a Node

The problem is our tree does nothing but stand around looking beautiful. Nothing wrong with that. Normally, though, you'd want some kind of action to occur when the node is clicked. To do this, you can use the TreeSelector Widget.

TreeSelector is a widget without a UI. You use it as a placeholder for connecting the tree to various Javascript actions. This makes it easy to construct many trees, and connect them to the same actions.

```
<script>
        dojo.addOnLoad(function() {
                dojo.event.topic.subscribe("nodeSelected",
                        function(message) { alert(message.node.title+" selected'
                );
        });
</script>

<div dojoType="TreeSelector" widgetId="tSelector" eventNames="select:nodeSelecte

<div dojoType="Tree"  selector="tSelector" >
  <div dojoType="TreeNode" title="Item 1">
    <div dojoType="TreeNode" title="Item 1.1" ></div>
  <div dojoType="TreeNode" title="Item 2">
</div>
```

(Is there an easier way to do this??? -- CAR)

When you click on a node, an alert box will pop up with the name you selected.

3 Submitting a Selected Node

You can make the selection event arbitrarily complex. But many times, you just want to pass the selected node along with a form. Simple!

```
<script>
dojo.addOnLoad(function() {
   dojo.event.topic.subscribe("nodeSelected",
       function(message) { document.menuForm.eatMe.value = message.node.title; }
   );
});
</script>

What would you like to eat first?

<form name="myForm">
  <input type="hidden" name="eatMe" value="" />
  <div dojoType="TreeSelector" widgetId="tSelector" eventNames="select:nodeSelect

    <div dojoType="Tree" selector="tSelector" >
      <div dojoType="TreeNode" title="Dessert (Recommended)">
        <div dojoType="TreeNode" title="Ice Cream" value="ICE76645" />
        <div dojoType="TreeNode" title="Cake" value="CAK85467" />
     </div>
     <div dojoType="TreeNode" title="Entree">
        <div dojoType="TreeNode" title="Meat Loaf" value="MTL18908" />
     </div>
  </div>
</form>
```

Clicking a node fills the value ICE76645, CAK85467, or MTL18908 into the hidden field "eatMe".

In this example, a tree is a standin for a select/options tag. For long lists, a select/option list gets too long to navigate. Humans like their information grouped and organized into smaller chunks. But databases thrive on flat namespaces, like the UPC system or Social Security Numbers. Trees give you the best of both worlds.

4 Gridlines

By default, tree nodes always have a gridline on their left. These gridlines helps the user quickly see which nodes are siblings, and which node is the parent.

You can turn off gridlines at the root level and/or for the entire tree. By default, each 1st level TreeNode connects to a "phantom" root node, as in:

SCREENSHOT

You can remove the phantom Root node so the first level nodes appear with no gridlines to their left, as in:

```
<div dojoType="Tree" showRootGrid="false">
```

Or you can turn all the gridlines off, as in:

```
<div dojoType="Tree" showGrid="false" showRootGrid="false">
```

5 Pre-Expanding Content

Let's say you'd like to highlight a particular Tree node, for example a default value. You can do this easily enough with tags around the node title:

```
<div dojoType="TreeNode"...>
  <div dojoType="TreeNode"...>
    <div dojoType="TreeNode" title="<span style='background-color:yellow' >The mo
    ...
  </div>
  <div dojoType="TreeNode"...>
    <div dojoType="TreeNode" title="Another Choice" />
  </div>
</div>
```

But if this TreeNode is 3 levels down, the user will have to expand both levels above it. A better way is to pre-expand content levels. This requires the attribute "expandLevel", which means "expand all nodes that are n levels below" If n is more than 1, all levels between 1 and n are expanded, since seeing an expanded node requires seeing an expanded node above. For example, if you added expandLevel="2" to the top TreeNode:

```
<div dojoType="TreeNode" expandLevel="2" ...>
  <div dojoType="TreeNode"...>
  ...
```

then both The Most Popular Choice and Another Choice will appear. But:

```
<div dojoType="TreeNode" expandLevel="1" ...>
   <div dojoType="TreeNode" expandLevel="1" ...>
   ...
  </div>
  <div dojoType="TreeNode"...>
</div>
```

will only expand Most Popular Choice.

Widget Namespaces

Overview

Widgets are combined into groups called namespaces. All the widgets built into Dojo are in the "dojo" namespace, but someone else could write their own widgets and put them in a different namespace. For example, you could write your own button and checkbox widgets, and put them into an "acme" namespace. Then "acme:Button" would be your button, and would be unrelated to the button object built into dojo, called "dojo:Button".

Usage

Defaults have been chosen to reduce boilerplate. A namespace maps to a top-level module by

default. A top-level module path defaults to *dojo/../*, and widgets are expected to be in *.widget*.

Given

```
<img dojoType="acme:Image" />
```

acme widgets are expected to be in *acme* folder next to *dojo* folder.

acme.widget module is expected to contain the *Image* widget.

```
/dojo
/acme/
/acme/widget/Image.js <- defines acme.widget.Image
```

Loading *acme.widget.Image* module is the only requirement for using *acme:Image* in this configuration. You can load that module as part of a build, by calling *dojo.require*, or automatically.

To use a folder location other than *../acme* call *dojo.registerModulePath*.

To select a widget module other than *acme.widget*, call *dojo.registerNamespace*.

Automatic Loading

To allow automatic loading of widgets in a namespace, include a manifest file. For the example above, the default resource for the manifest would be:

```
<root>/acme/manifest.js
```

To customize the folder location of module *acme* call *dojo.registerModulePath*.

For most users employing the auto-require system, the manifest file contains a call to *dojo.registerNamespaceResolver*.

A namespace resolver tells Dojo what module to load for a named widget.

```
dojo.provide("acme.manifest");
dojo.require("dojo.string.extras");
dojo.registerNamespaceResolver("acme",
  function(name){
    return "acme.widget."+dojo.string.capitalize(name);
  }
);
```

The input string *name* will always be lower-case. So this resolver triggers loading of module *acme.widget.Calendar* for widget *acme:calendar*.

The load-time module is not necessarily the same as the widget class module. For example, the *acme.widget.Calendar* class might be loaded via *acme.widget.allWidgets*.

The resolver tells dojo the module to require to load a widget.

To select a widget class module other than *acme.widget*, call *dojo.registerNamespace*.

API

dojo.registerModulePath(module, path): maps a module name to a path (formerly *setModulePrefix*).

An unregistered module is given the default path of *../*, relative to Dojo root. For example, module *acme* is mapped to *../acme*. If you want to use a different module name, use *registerModulePath*.

dojo.registerNamespace(namespace, widget_module [, resolver]): maps a module name to a namespace for widgets, and optionally maps widget names to modules for auto-loading.

An unregistered namespace is mapped to an eponymous module. For example, namespace *acme* is mapped to module *acme*, and widgets are assumed to belong to *acme.widget*. If you want to use a different widget module, use*registerNamespace.*

dojo.registerNamespaceResolver(namespace, resolver): a resolver function maps widget names to modules, so the widget manager can auto-load needed widget implementations.

The resolver provides information to allow Dojo to load widget modules on demand.When a widget is created, a namespace resolver can tell Dojo what module to require to ensure that the widget implementation code is loaded.

The input string in the *name* argument will always be lower-case.

```
dojo.registerNamespaceResolver("acme",
  function(name){
    return "acme.widget."+dojo.string.capitalize(name);
  }
);
```

Examples

Let's say we have a Dojo install at root:

```
/dojo/dojo.js
/dojo/[whatever else is in the particular dojo install]
```

We want to create custom modules, and decide to put them in:

```
/acme
```

Note that the path to *acme* from *dojo* is:

```
../acme
```

For the widget examples, let's say we made some custom widgets, including one called *acme.widgets.Calendar*, and put them in:

```
/acme/widgets/variousWidgets.js
```

Automatic Loading

Main document

```
<script src="/dojo/dojo.js"></script>
<script>
  dojo.require("dojo.widget.*");
</script>
```

Include a manifest file*: /acme/manifest.js*

```
dojo.provide("acme.manifest");
dojo.registerNamespaceResolver(function(name) {
  return "acme.widgets.variousWidgets";
});
```

To support markup like so:

```
```

The *acme* namespace triggers require of *acme.mainfest.* The resolver is used to match *calendar* to a required module (i.e. *acme.widgets.variousWidgets*). Then *acme.widgets* module is searched for *calendar* implementation matching the current rendering environment.

Explicit Loading

Main document

```
<script src="/dojo/dojo.js"></script>
<script>
  dojo.require("acme.widgets.variousWidgets");
</script>
```

Supports markup like so:

```

```

acme.widgets module is searched for *calendar* implementation matching the current rendering environment.

Non-Widget Resources

Main document

```
<script src="/dojo/dojo.js"></script>
<script>
 dojo.require("acme.lib");
</script>
```

acme/lib.js file:

```
// ... additional code ...
```

Builds

With a build you can use any of these formats, but a manifest is not required.

Main document

```
<!-- dojo.js is a build -->
<script src="/dojo/dojo.js"></script>
<!-- dojo.require(s) can be here, although they are ignored -->
```

Supports markup like so:

```
<div dojoType="acme:calendar"></div>
```

Writing Your Own Widget

This section discusses the internals of widgets, and how to write your own.

Compound Widgets

TODO: moved this from "The Memo" page, where it definitely didn't belong, but it could use

some expansion

This is a crucial next step for widget authors - creating widgets which themselves contain inner widgets, resulting in what we could call '*compound widgets*'.

The procedure is simple, just add to your widget .js file, within the widget atributes object, the line:

```
        widgetsInTemplate:true,
```

then, your subwidgets will nest perfectly within the main outer widget. You should also be able to nest to any arbitrary depth. Just remember though to abstain from setting the `id` or `dojoId` attributes in your html, rather set `dojoAttachPoint` instead to insert into your main outer widget a named attribute which references your subwidget. This way, you won't pollute the global element namespace. Otherwise, you'll hit problems if creating multiple instances of your compound widgets.

The Monolithic App Widget

One approach to dojo application design is to build the app as one huge compound uber-widget, containing all the needed sub-widgets.

This approach will likely have a natural feel and appeal to those experienced in desktop GUI programming.

If you want to go in this direction, then your app can get loaded in the client by a very minimal HTML file which just pulls in a minimal stylesheet, **include**s dojo.js, **dojo.require()**s your main app widget, then invokes that widget in a single
tag within your document . There are those of us who feel that the less javascript code you have within html files, the better!

Custom Namespace

TODO:

Ã‚Â Ã‚Â Ã‚Â - this is old info?Ã‚Â current description is at
http://dojo.jot.com/WikiHome/Modules%20%26%20Namespaces

Ã‚Â Ã‚Â Ã‚Â Ã‚Â Ã‚Â Ã‚Â - As indicated below, it is correct for the current stable version (0.3.1). It will need to be updated for the next release.

If you're planning on creating your own widgets then it's probably a good idea to keep your own code completely separate from the Dojo codebase. This will make life easier if/when you come to install a new version of Dojo, and also prevent any name clashes with native Dojo widgets.

First of all, you'll want to create a directory structure outside the Dojo source directory where your code will be stored. For example, let's call this new directory 'user', so that your directory structure looks something like this:

```
/dojo
```

```
/user
```

```
index.html
```

Next you need to tell Dojo that this new namespace exists and where it lives. You can do this with *dojo.setModulePrefix(namespace, path)*, like this:

```
dojo.setModulePrefix("user", "../user");
```

Note that the path (the second parameter) is relative to the root of the dojo source directory.

[Please note: the use of *dojo.setModulePrefix()* is deprecated (by Dojo version 0.5), and will be replaced with *dojo.registerModulePath()*, which takes the same initial parameters.]

Now since Dojo will look for widgets in a subdirectory (under '/user') called 'widget', we need to create that too:

```
/dojo
```

```
/user
```

```
        /widget
```

```
index.html
```

Now you can create our own widgets in the user/widget directory and include them using *dojo.require()* as usual:

```
Ã‚Â Ã‚Â Ã‚Â  dojo.require("user.widget.MyWidget");
```

Unfortunately, in version 0.3.1 you can't use the namespace when calling your widget (this has been

```
Ã‚Â Ã‚Â  Ã‚Â
```

...or programatically...

```
Ã‚Â Ã‚Â  Ã‚Â var new_widget = dojo.widget.createWidget
```

```
Ã‚Â Ã‚Â  Ã‚Â (
```

```
Ã‚Â Ã‚Â  Ã‚Â Ã‚Â Ã‚Â  Ã‚Â 'MyWidget',
```

```
Ã‚Â Ã‚Â  Ã‚Â Ã‚Â Ã‚Â Ã‚Â  Ã‚Â {
```

```
Ã‚Â Ã‚Â  Ã‚Â Ã‚Â Ã‚Â Ã‚Â  Ã‚Â Ã‚Â Ã‚Â  Ã‚Â some_property:'Some Value'
```

```
Ã‚Â Ã‚Â  Ã‚Â Ã‚Â Ã‚Â Ã‚Â  Ã‚Â }
```

```
Ã‚Â Ã‚Â  Ã‚Â );
```

In future versions (and current nightly/SVN builds), you would prepend the name of the widget with the namespace and a colon, for example:

TODO: correct, expand...

More on Templates

This chapter discusses how widget templates work.

The Template

If you remember, in a previous chapter we looked at the template for the floating pane:

Basically, the idea is the the source HTML is replaced by this template. But there's a lot more stuf

Attaching DOM Nodes

Inside of FloatingPane.js you will notice various variables that correspond to (point to) dom nodes within the instantiated template. It's easier to explain by example.

Here are some lines from the template above (note the highlighted section):

```
Ã‚Â Ã‚Â Ã‚Â
```

```
Ã‚Â Ã‚Â  Ã‚Â  Ã‚Â  Ã‚Â Ã‚Â Ã‚Â  Ã‚Â
```

And here's the corresponding code from FloatingPane.js:

```
titleBar: null,
titleBarIcon: null,
...
```

Merely by having that code, the titleBar variable points to the dom node generated by the template. So you can do something like:

```
this.titleBar.style.color="red";
```

The Container Node

There's a special attach point called the "container node". Consider this source HTML:

```
Hello world!
```

press me

This is a floating pane that contains some content, including a widget. What happens to the conte

```
dojoAttachPoint="containerNode"
```

class="dojoFloatingPaneClient">

In addition, since the floating pane contains contents, we have this line in FloatingPane.js:

```
isContainer: true,
```

Attaching Events

Another very useful feature is declarative event handling. Notice this line from the template above:

```
class="dojoFloatingPaneMaximizeIcon">
```

Just by adding that line, whenever the maximize action div is clicked, the widget's maximizeWindow() function will be called.

If you don't specify a function name, it defaults to the event name. For example,

due to the above highlighted code, whenever you mouse down, onMouseDown() is called.

Variable Replacement

```
${this.caption}
```

When dojo creates the widget from the template, it substitutes the value of this.caption from the

More on The Javascript Object

Now we'll see how you write the javascript portion of a widget.

Previously we looked at the CSS and the HTML used to define a widget. The third and final component to widgets is a javascript class to handle widget rendering details and events on the widget.

defineWidget

The first step to writing the javascript for a widget is to call defineWidget. Below I'm defining a widget called my.widget.html.Foo that extends dojo.widget.HtmlWidget,the base class for most widgets. (We'll talk about different base classes and what "html" means in a later document.)

```
dojo.widget.defineWidget("my.widget.Foo", dojo.widget.HtmlWidget, {
    function() {
      // do initialization tasks, make instance properties
    },
    {
       ...prototypical properties (in object notation)...
    }
);
```

Using *dojo.widget.defineWidget*, the tasks below are performed automatically:

- register widget package (tell dojo the namespace contains widgets)
- add the parse tree handler (identify a markup tag with the widget)
- set the widgetType
- invoke ancestor constructor
- inherit from ancestor prototype

Alternately, I might want to extend an existing widget. Here I'm making an enhanced version of Foo called FooPlus:

```
dojo.widget.defineWidget("my.widget.FooPlus", my.widget.Foo, {
```

```
        function() {
          // do initialization tasks, make instance properties
        },
        {
           ...prototypical properties (in object notation)...
        }
    );
```

Parameters

OK, that's the skeleton for the widget, but what do we put inside? The first thing to think about are the parameters that are used when you construct the widget. Every widget can take parameters. For example:

```
```

Where are the parameters defined, and how do you set their types? Actually, they are just properties in the javascript class. In this case:

```
    toggle: "",                 // string
    toggleDuration: 0,          // integer
    onClick: function(){}       // function
```

Javascript doesn't have types, so how do we specify the types of the parameters? By specifying an example. In the above case, 0 means *integer* and "" means *string*.

Note: don't set them to null or it won't work!

Default values:

You can also specify default values in the javascript file. If the user doesn't specify a value for a parameter the default is used. For example:

```
    toggle: "fade"
```

Important Properties To Set

Next, you need to set certain properties that define how the widget operates.

isContainer

True/False. Must be set to true if the widget has child HTML or child widgets

snarfChildDomOutput

True/False. Set this to true if you are making something like a container node, where the input is just a list of widgets. It resolves issues where a child's generated DOM tree cannot be put back into the same place the source dom tree was. (Because [td] cannot be a child of [div], etc.)

templatePath

The path to the template HTML file, if one exists for the widget. This needs to be a

dojo URI object, and is normally one of the two options shown below:

```
dojo.uri.dojoUri("src/widget/templates/HtmlFloatingPane.html"),
 or
dojo.uri.moduleUri("mywidgetset","widgets/html/MyWidget.html"),
```

templateCssPath

The path to the template CSS file, if one exists for the widget. This needs to be a dojo URI object, and is normally one of the two options shown below:

```
dojo.uri.dojoUri("src/widget/templates/HtmlFloatingPane.css"),
 or
dojo.uri.moduleUri("mywidgetset","widgets/css/MyWidget.css"),
```

templateString / templateCssString

If the CSS or HTML for a widget is very simple, you can specify it in the javascript rather than using templatePath/templateCssPath to refer to other files. This is what the dojo build process does automatically to embed templates/CSS in your widget code when you specify the *intern-strings* option.

```
templateString: "
Simple Template
",
templateCssString: ".simple { color:blue; font-size:12pt; }",
```

Initialization Methods

Inside a widget file you will notice a number of functions for initialization. The most important ones are described below in the order they are called during the widget creation process.

postMIxInProperties()

this is called after the properties (see previous section) are initialized to the user specified values, but before the HTML template is instantiated.

Typical actions to perform here are validating and adjusting parameters provided to the widget.

fillInTemplate()

This is called after the template has been instantiated, so this.domNode points to the generated DOM tree. However, the children DOM nodes (for containerNode) and widgets haven't yet been copied over, and the widget's DOM node has not yet been placed in the actual HTML document.

Typical actions to perform here include:

- Enabling or disabling parts of the widget
- Applying styles/classes/etc
- Creating widgets that attach to nodes in the template
- Setting initial state

postCreate()

This is called after the children dom nodes and widgets have been instantiated. *However, for programatically created widgets, none of the children exist yet, because they are added after createWidget() finishes, via the addChild() call.*

Typical actions to perform here include:

- Connecting event handlers
- Manipulating parent or child nodes (with the above caveat)

Arrays, Objects, and Statics

Widget attributes that are Arrays or Objects need to be declared in the initializer() function, rather than like other variables (numbers or strings), so that they are not inadvertently shared between other instances of the same widget.

On the other hand - to make a static variable (i.e. a variable that is shared across all instances of the widget), just take advantage of the above issue:

```
// static1 and static2 are shared across every Foo widget
statics: { static1: 0, static2: "" }
```

```
this.statics.static1++;    // increments the single copy
```

Important Variables

this.domNode - points to root of generated tree

this.containerNode - place where child HTML was attached

this.children - array of child widgets

The Memo

Overview

This page serves as an introduction to the art and science of creating one's own dojo widgets, and invoking them in html code just like mainstream Dojo widgets.

The following walk through will take you through all the steps needed to create a simple widget, ie a widget that contains only dom elements, not any nested dojo widgets.

Following this walk through are some instructions for creating compound widgets (ie, widgets that include other dojo widgets).

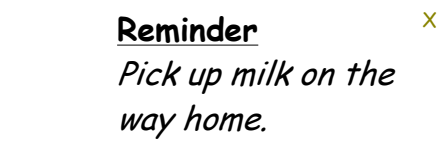
Getting Started

Let's say that you want to make a memo widget. Just a yellow sticky note to remind

yourself of your dentist appointment, or whatever. Something that you can put on the screen and then erase later. Something so that a call like this:

```
<div dojoType="Memo" title="Reminder">
        Pick up milk on the way home
</div>
```

will produce something that looks like this:



The Template

The first step is to write HTML and CSS that prototypes how the widget will look. You can do this in any editor of your choice. I made the prototype above using this HTML:

```
<div class="memo">
  <div class="title">Reminder</div>
  <div class="close">X</div>
  <div class="contents">Pick up milk on the way home.</div>
</div>
```

And this CSS:

```
.memo {
        background: yellow;
        font-family: cursive;
        width: 10em;
}

.title {
        font-weight: bold;
        text-decoration: underline;
        float: left;
}

.close {
        float: right;
        background: black;
        color: yellow;
        font-size: x-small;
        cursor: pointer;
}

.contents {
        clear: both;
        font-style: italic;
}
```

Note how I put as much of the formatting code into the CSS. This isn't necessary, but it does make it easier for other people to customize the widget, merely by altering the CSS.

Turning it into a widget

To make this memo into a widget, you need to declare a javascript object that connects with the HTML and CSS template above. So, put the HTML in a file called Memo.html, the CSS in a file called Memo.css, and make the Memo.js file below:

```
dojo.widget.defineWidget(
        // widget name and class
        "acme.Memo",

        // superclass
        dojo.widget.HtmlWidget,

        // properties and methods
        {
                templatePath: dojo.uri.dojoUri("src/widget/Memo.html"),
                templateCssPath: dojo.uri.dojoUri("src/widget/Memo.css")
        }
);
```

Contents and Parameters

The obvious problem with this widget is that no matter what is inside the source div, it always says "*Pick up milk on the way home*". What you need is for the contents of the source div to be inserted into the generated output. This is what the "containerNode" is for, and you use it like this...

First, in the template, get rid of the static content, and instead mark that the div should hold the content from the source.

```
<div class="memo">
  <div class="title">Reminder</div>
  <div class="close">X</div>
  <div class="contents" dojoAttachPoint="containerNode"></div>
</div>
```

Then, in the javascript object, denote that this widget is a container:

```
dojo.widget.defineWidget(
        // widget name and class
        "acme.Memo",

        // superclass
        dojo.widget.HtmlWidget,

        // properties and methods
        {
                isContainer: true,
                templatePath: dojo.uri.dojoUri("src/widget/Memo.html"),
                templateCssPath: dojo.uri.dojoUri("src/widget/Memo.css")
        }
);
```

OK, what about the title? The title is specified as an attribute:

```
<div dojoType="Memo" title="Reminder">
```

```
        Pick up milk on the way home
</div>
```

That means that it's a parameter to the widget. Parameters are specified as normal widget properties. In this case, the widget properties would look like this:

```
        // properties and methods
        {

                // parameters
                title: "Note",

                // settings
                isContainer: true,
                templatePath: dojo.uri.dojoUri("src/widget/Memo.html"),
                templateCssPath: dojo.uri.dojoUri("src/widget/Memo.css")
        }
```

This widget now has a "title" parameter, with a default value of "Note"

How do you stick this parameter's value into the widget? Luckily widget templates have variable substitution, so no coding is necessary. Just modify the template to use this parameter:

```
<div class="memo">
  <div class="title">${this.title}</div>
  <div class="close">X</div>
  <div class="contents" dojoAttachPoint="containerNode"></div>
</div>
```

Events

OK, the content of the widget is showing up correctly, but how to you make clicking the X cause the widget to disappear? It's pretty simple, and hardly requires any javascript. The first step is to modify the template to handle click events on the X:

```
<div class="memo">
  <div class="title">\${this.title}</div>
  <div class="close" dojoAttachEvent="onClick">X</div>
  <div class="contents" dojoAttachPoint="containerNode"></div>
</div>
```

Then you simply add a method to the widget javascript object to handle the click:

```
                onClick: function(evt){
                        this.destroy();
                }
```

That's it! Your first functioning widget!

Final code

Memo.html

```
<div class="memo">
  <div class="title">\${this.title}</div>
  <div class="close" dojoAttachEvent="onClick">X</div>
  <div class="contents" dojoAttachPoint="containerNode"></div>
</div>
```

Memo.css

```
.memo {
        background: yellow;
        font-family: cursive;
        width: 10em;
}

.title {
        font-weight: bold;
        text-decoration: underline;
        float: left;
}

.close {
        float: right;
        background: black;
        color: yellow;
        font-size: x-small;
        cursor: pointer;
}

.contents {
        clear: both;
        font-style: italic;
}
```

Memo.js

```
dojo.widget.defineWidget(
        // widget name and class
        "acme.Memo",

        // superclass
        dojo.widget.HtmlWidget,

        // properties and methods
        {
                // parameters
                title: "Note",

                // settings
                isContainer: true,
                templatePath: dojo.uri.dojoUri("src/widget/Memo.html"),
                templateCssPath: dojo.uri.dojoUri("src/widget/Memo.css"),

                // callbacks
                onClick: function(evt){
                        this.destroy();
                }
        }
);
```

Containers

LinkPane

ContentPane

Part 5: "Connecting the pieces"

Event System

Unlike the DOM events that web programmers normally associate with the word "event", Dojo takes a broad view of events. The tools in *dojo.event.** allow developers to treat any function call (DOM event or otherwise) as an event that can be listened to. Using Dojo, code can register to "hear" about any action through a uniform API.

Events are essential for Dojo based applications as they drive the user interface, result in AJAX requests, and allow widgets to interact with each other. In a sense events are the glue that ties an application together. Cross browser event handling code can difficult to write from scratch as there are many ways in JavaScript of handling events and each browser has its own quirks and issues.

Dojo abstracts the JavaScript event system in the dojo.event module and provides a few options for handling events which include simple event handlers, event listeners using before, after, and around advice, and topics. The Dojo event APIs are not mutually exclusive; in many cases you will use a combination of the APIs depending on your use cases.

In this chapter we'll show you:

- how to use these tools
- what makes them completely different from other JavaScript event systems you may have used
- why you'll never start writing JavaScript without *dojo.event.connect()* again.

Before, After, and Around Advice

In addition to being able to call any function or method after any other function or method call, *connect()* can be used to call listeners *before* the source function is called. In Aspect Oriented Programming terminology, this is called "before advice" while the previous examples have all be "after advice". The terminology is confusing, but for a lack of anything less mind-bending or better accepted, we adopt it for the advanced cases that *connect()* supports.

Here's how we'd ensure that "bar" gets alerted *before* "foo" when *exampleObj.foo()* is called:

```
dojo.event.connect("before", exampleObj, "foo", exampleObj, "bar");
```

As you can see, we just perpended our previous call to *connect()* with the word "before". In the other cases, the word "after" was the implied first argument, which we could have added if we wanted, but typing more isn't something any of us want, and most of the time "after" is what you want anyway.

The same connection using *kwConnect()* looks like:

```
dojo.event.kwConnect({
    type:       "before",
    srcObj:     exampleObj,
    srcFunc:    "foo",
    targetObj:  exampleObj,
    targetFunc: "bar"
});
```

Before and after advice give us tools to handle a huge range of problems, but what about when the listener and the source functions don't have the same call signatures? Or what about when you want to change the behavior of a function from someone else's code but don't want to change their code? If we take the view that any function call in our environment is an event, then shouldn't we also have an "event object" for each of them? When using *dojo.event.connect()*, this is exactly what happens under the covers, and we can get access to it via "around advice". Long story short, around advice allows you to wrap any function and manipulate both it's inputs and outputs. This'll let us change both the calling signatures of functions and change arguments for listeners (among other things).

Unlike the other advice types, around advice requires a little bit more cooperation from the author of the around advice function, but since you'll probably only be using it in situations where you know that you want to explicitly change a behavior, this is isn't really a problem. This example take a function *foo()* which takes 2 arguments and provides a default value for the second argument if one isn't passed:

```
function foo(arg1, arg2){
    // ...
}
function aroundFoo(invocation){
    if(invocation.args.length < 2){
        // note that it's a real array, not a pseudo-arr
        invocation.args.push("default for arg2");
    }
    var result = invocation.proceed();
    // we could change the result here
    return result;
}
dojo.event.connect("around", "foo", "aroundFoo");
```

The *aroundFoo()* function must take only a single argument. This argument is the method-invocation object. This object has some useful properties (like *args*) and one method, *proceed()*. *proceed()* calls the wrapped function with the arguments packed in the *args* array and returns the result. At this point, you can further manipulate the result before returning it. If you don't return the result of *proceed()*, it will appear to the caller as though the wrapped function didn't return a value. At any point you could call another function to do things like log timing information.

Once this connection is made, every time *foo()* is called *aroundFoo()* will check it's argument and insert a default value for *arg2*. Around advice is kind of like goto in C and C++: if you don't know better you can make huge messes, but when you really need it, you *really* need it.

Despite the power of around advice, it's not very often that globally changing a function signature or return value is the best plan. More often, you'll just want to smooth over the differences in calling signatures between two functions that are

being connected. As you might have come to expect by now, Dojo provides a solution for this type of impedance matching problem too.

The solution is *before-around* and *after-around* advice. These advice types apply a supplied around advice function to the listener in a connection. They only apply the around advice when the listener function is being called from the connected-to source. Put another way, it's connection-specific argument and return value manipulation.

To access before-around and after-around advice, just pass in another object/name pair to a normal "before" or "after" connection, like this:

```
var obj1 = {
    twoArgFunc: function(arg1, arg2){
        // function expects two arguments
    }
};
var obj2 = {
    oneArgFunc: function(arg1){
        // this function expects a two-element array
        // as its only parameter
    }
};
// we'd probably connect the functions somewhere else. Perhaps in a
// different file entirely.
function aroundFunc(invocation){
    var tmpArgs = [
                    invocation.args[0],
                    invocation.args[1]
                  ];
    invocation.args = tmpArgs;
    return invocation.proceed();
}
// after-around advice
dojo.event.connect( obj1, "twoArgFunc",
                    obj2, "oneArgFunc",
                    "aroundFunc");
```

Each function now gets what it expects, and the code calling *obj1.twoArgFunc()* never need be the wiser that any of this is happening.

Connecting Multiple Events

Multiple Listeners

Connect also transparently handles multiple listeners. They are called in the order they are registered. This would kick off two separate actions from a single onclick event:

```
var handlerNode = document.getElementById("handler");
dojo.event.connect(handlerNode, "onclick", object, "handler");
dojo.event.connect(handlerNode, "onclick", object, "handler2");
```

We didn't have to change the API we were using, rewire anything for multiple events, etc. It all just works. Now every time you click the node, and *object.handler()* gets called and then *object.handler2()* gets called.

Finally, note that connect can take an array of objects as input:

```
dojo.event.connect(
        [Ã¢â,¬?id1, Ã¢â,¬Å“id2Ã¢â,¬?, Ã¢â,¬Å“id3Ã¢â,¬?], Ã¢â,¬Å“onclic
        listenerObj, Ã¢â,¬Å“handleOnClickÃ¢â,¬?);
```

Disconnection and Multi-Connection

Connecting is one thing, but what about when you want to stop listening? *dojo.event.disconnect()* will stop the listening arrangement between functions, but must be pass *exactly the same* arguments as were passed to connect in order to ensure successful disconnection.

If there's anything that can trip up new users of *dojo.event.connect()*, it's inadvertently connecting multiple times. Very often, a piece of code will get called multiple times, and it will contain a *dojo.event.connect()* call. The developer is then surprised when their listener function is called multiple times for every time the source function fires. What to do?

Connecting Once And Using Keywords

One option is to move your *connect()* call to a location that will get invoked only once, but sometimes that's just not feasible. An optional argument to *connect()* ensures that the same arguments to connect passed multiple times will result in only one connection between functions. Unfortunately, it's the 8th parameter. Ugh. The last thing we want to do is remember 8 different parameters. The best answer in this scenario is to use the keyword-argument version of connect, aptly named *kwConnect()*. To use it, we have to give the parameters we've been using so far names. Here's our object connection example using *kwConnect()* and the *once*

parameter:

```
dojo.event.kwConnect({
    srcObj:     exampleObj,
    srcFunc:    "foo",
    targetObj:  exampleObj,
    targetFunc: "bar",
    once:       true
});
```

As I'm sure you've already guessed, there's an analogous *kwDisconnect* method. Just pass it what you pass *kwConnect*, naturally.

Event Object

Using dojo.event also masks browser differences by normalizing the event object (for DOM node events) so you can use common event code in any browser.

Fixed event objects have these modifications:

For key events, a set of event key code aliases are installed, so you can express `(e.keyCode == e.KEY_ESC)`. Also, a reverse key code lookup is installed, so you can express `(e.revKeys[e.keyCode] == 'KEY_ESC')`.

These properties are made available in all browsers:

- target
- currentTarget
- pageX/pageY - position of cursor relative to viewport
- layerX/layerY
- fromElement
- toElement
- charCode

The following methods are also made available:

- stopPropagation() - stops other event handlers (on parent domnodes) from firing
- preventDefault() - stops things like following the href on a hyperlink.
- callListener() - ???

Additionally, `event` (W3) vs. `window.event` (IE) is taken care of: all connected event handlers get passed a fixed event object (even in IE).

As an example, the code below will work in any browser:

```
dojo.event.connect(dojo.byId("foo"), "onmousemove"),
        function(evt){
                alert("mouse at pos" + evt.pageX + "," + evt.pageY);
        });
```

Events And Widgets

A brief note about events and widgets.

dojo.event.connect() can be used with widgets just like any other objects. However, there is a shortcut for defining "after" advice on a widget.

Click Me!

In the above example, the alert is called after the widget's own onClick() function finishes executing.

On the other hand, in the case below:

Click Me!

The widget's onClick function is *replaced* by function foo.

This is a somewhat confusing discrepancy (the latter behavior is more consistent with widget parameter setting in general), but it's left in place for backwards compatibility.

Page Load / Unload

Often you will want to schedule some code to run on page load. Traditionally, this is done like

```
window.onLoad = ...;
```

or perhaps

```

```

However, that won't work for Dojo, because Dojo needs to override window load and unload. So, you should do this:

```
function init(){
        ...
}
dojo.addOnLoad(init);
function cleanup(){
        ...
}
dojo.addOnUnload(cleanup);
```

Just like the normal dojo.event.connect() call, addOnLoad() and addOnUnload() can be called multiple times without overwriting the previous values, so you don't have to worry about one piece of Javascript code affecting another.

The line `dojo.addOnLoad(init);` tells Dojo to call the `init` function when it has finished loading correctly. This is **very** important! If the `init` function was called before Dojo has finished parsing the HTML then widget objects would not have been instantiated and so would not exist at that point in time - causing a nasty error.

Publish and Subscribe Events

Use publish and subscribe to communicate events anonymously between widgets or any JavaScript functions of your choosing. You may also consider customizing the widget to allow the topic name to be passed in as an initialization parameter to make the widget more flexible.

The following example shows how two objects may use publish and subscribe to communicate with each other.

```
var foo = new function() {
    this.init = function() {
        dojo.event.topic.subscribe("/mytopic", this, processMessages);
    }

    function processMessages(message) {
```

```
            alert("Message: " + message.content);
        }
    }

    var bar = new function() {
        this.showMessage = function(message) {
            dojo.event.topic.publish("/mytopic", {content: message});
        }
    }

    foo.init();
    bar.showMessage("Hello Dojo Master");
```

In the exampe above the object foo registers with a topic called '/mytopic' when the init function is called. Bar publishes a message to the topic '/mytopic' which results in the function showMessages being called. You can create any number of topics to publish and subscribe to.

Using publish and subscibe is very easy and it makes wiring things together easy. Widget communication by default is within the same JavaScipt execution context. Not all event handling need be exposed using publish and subscribe however using these types of events allows your code to be flexible and permit future integration with other widgets.

Topics

Dojo provides a means of anynonymous event communication which can be very useful to connect together widgets in a page that may have no previous knowledge of each other. This maybe done using publish/subscribe style events. Publish subscribe style events require that the components that wish to communicate information simply share the name of a topic or queue to which the events are published/subscribed to. Objects may be passed as an argument of the events which provides a powerful means of inter-object/widget communication.

The API for publishing to a topic is as follows:

```
    dojo.event.topic.publish("/topicName", args);
```

That is pretty much it to publish an event. The arguments are passed as an object literal and will be seen by all clients subscirbed to the corresponding topic "/topicName".

The API for subscribing to a topic is as follows:

```
    dojo.event.topic.subscribe("/scroller", targetObj, targetFunc);
```

A more detailed example follows:

```
    var ac;
    var is;
    function init() {
```

```
        ac = new AccordionMenu();
        ac.load();
        is = new ImageScroller();
        is.load();
    }
    function Scroller() {
        this.setProducts = function(pid) {
            // show the products for pid
        }
        this.handleEvent = function(args) {
            if (args.event == 'showProducts') {
                this.setProducts(args.value);
            }
        }

        this.load = function () {
            dojo.event.topic.subscribe("/scroller", this, handleEvent);
        }
```

```
    function Accordion() {
        function expandRow(target) {
           ...
           var link = document.createElement("a");
           dojo.event.connect(link, "onclick", function(evt){
               this.target = target;
               dojo.event.topic.publish("/scroller", {event: "showProducts", va
           });
        }
    }
```

An "onclick" event on the element link will cause an event to be published to the topic name "/scroller" which is shared by both the Accordion and Scroller objects. In the case of this example the "handleEvent" function of the Scroller object will be callsed with the object literal {event: "showProducts", value : target}.

As can be seen topics can be very useful. When designing widgets or objects that need to interact with widgets or objects consider using publish and subscribe style events.

Working with Simple Events

Events in JavaScript or Dojo based applications are essential to making applications work. Connecting an event handler (function) to an element or an object is one of the most common things you will do when developing applications using Dojo. Dojo provides a simple API for connecting events via the dojo.event.connect() function. One important thing to note here is that events can be mapped to any property or object or element. Using this API you can wire your user interfaces together or allow for your objects to communicate. The dojo.event.connnect() API does not require that the objects be Dojo based. In other words, you can use this API with your existing interfaces.

DOM Events

dojo.event.connect has multiple function signatures, but one of the simplest is:

```
dojo.event.connect(srcObj, "srcFunc", targetFunc);
```

The arguments are the source object, the source function (in quotes) and the target function reference or anonymous function.

Here we have a DOM node called mylink, and whenever that DOM node is clicked myHandler will be called:

```
var link = dojo.byId("mylink");
dojo.event.connect(link, "onclick", myHandler);
```

```
function myHandler(evt) {
   alert("dojo.connect handler");
}
```

Above the "onclick" property of link element is connnected to the function myHandler.

But what if we don't want to set up a named function for the event handler? No problem:

```
var link = dojo.byId("mylink");
// connect link element 'onclick' property to an anonymous function
dojo.event.connect(link, "onclick", function(evt) {
      ...
  });
```

The example above shows how an anonymous function can be mapped to the "onclick" property of a link element with an existing in-lined DOM 1 style handler connected to using the "onclick" attribute of the element.

So far, though, we're not doing anything that can't be done by setting the *onclick* property of the DOM Node. But what about attaching a method of an object to a DOM Node's event handler? Normally, you'd have to do something like:

```
var handlerNode = document.getElementById("handler");
handlerNode.onclick = function(evt){
    object.handler(evt);
};
```

Dojo simplifies it to:

```
var handlerNode = document.getElementById("handler");
dojo.event.connect(handlerNode, "onclick", object, "handler");
```

This *connect()* call ensures that when *handlerNode.onclick()* is called, *object.handler()* will be called with the same arguments. Language limitations of JavaScript make it impossible to pass in the object and function name together, however separating them into an object reference and function name isn't difficult.

Other Events

So we've seen that *connect()* can handle DOM events, but what about that more expansive view of events that was mentioned earlier? To demonstrate, lets define a simple object with a couple of methods:

```
var exampleObj = {
    counter: 0,
    foo: function(){
        alert("foo");
        this.counter++;
    },
    bar: function(){
        alert("bar");
        this.counter++;
    }
};
```

So lets say that I want *exampleObj.bar()* to get called whenever *exampleObj.foo()* is called. We can set this up the same way that we do with DOM events:

```
dojo.event.connect(exampleObj, "foo", exampleObj, "bar");
```

Now calling *foo()* will also call *bar()*, thereby incrementing the counter twice and alerting "foo" and then "bar". Any caller that was counting on getting the return value from *foo()* won't be disappointed. The source method should behave just as it always has. On the other hand, since there's no explicit caller for *bar()*, it's return value will be lost since there's no obvious place to put it.

The Dojo event model

We've also inadvertently demonstrated that *connect()* takes variable forms of arguments. So far, it's correctly handled:

- object, name, name
- object, name, function pointer
- object, name, object, name

This is par for the course when using *connect()*. Since it is used in so many places, for so many things, and in so many ways, *connect()* does a lot of checking and normalization of it's arguments. The connect method tries to disambiguate the types of the positional parameters based on usage. Some common usages are:

- dojo.event.connect(scope1, "functionName1", "globalFunctionName2");
- dojo.event.connect("globalFunctionName1", scope2, "functionName2");
- dojo.event.connect(scope1, "functionName1", scope2, "functionName2");
- dojo.event.connect("after", scope1, "functionName1", scope2, "functionName2");
- dojo.event.connect("before", scope1, "functionName1", scope2, "functionName2");

The first paramether is adviceType ("after" and "before") and is optional. If it is not supplied then it defaults to "before". In the above example, adviceType was not provided and so the default, in this case "before" is used.

srcObj - the scope (scope1) in which to locate/execute the named srcFunc. This is also optional and if it is not supplied then Dojo assumes the global object.

srcFunc - the name of the function to connect to. In the above examples it is "globalFunctionName2" or "functionName2". This is in conjunction with the srcObj parameter. Dojo will look for a function, srcFunc, in srcObj.

adviceObj - scope (scope 2) in which to locate/execute the named adviceFunc. Again this parameter is optional and if not supplied Dojo will assume the global object.

adviceFunc - name of the function ("globalFunctionName1" or "functionName1") being conected to srcObj.srcFunc

Delaying Execution

There's one more modifier up the sleeve of *connect()*/*kwConnect()*; delayed calling. The *delay* property in *kwConnect* (the 9th positional parameter for *connect*) is a delay in milliseconds for those platforms that support it (all browsers do).

The last problem worth mentioning is circular connections. Circular connections can occur when (perhaps even indirectly) a listener also calls the function it's listening to. The good news is that in a JavaScript interpreter, this will pretty quickly yield an exception of some sort. "Too much recursion" is a tip off that you've hit this problem. Debugging circular connections can be opaque, but tools like Venkman help.

I/O

The Dojo project is working to build a modern, capable, "webish", and easy to use DHTML toolkit. Part of that effort includes smoothing out many of the sharp edges of the DHTML programming and user experience. On the back of such high-profile success stories such as Oddpost, Google Maps, and Google Suggest, the XMLHTTP object has been getting a lot of attention of late. Sadly, in spite of all the coverage, developers have been on their own when it comes down to solving the usability problems that come along for the ride.

Cross Domain XMLHttpRequest using an IFrame Proxy

***Note**: The code for this feature is available in Dojo 0.4 and later. IE 7 Support in Dojo 0.4.1 and later.*

Background

The browser security model does not allow using XMLHttpRequest (XHR) from one web page domain to contact an URL on another domain. However, there are cases when it would be nice to do cross domain XHR requests. There is a proposal in the W3C's Web API group to address this need (see this Mozilla tracking bug, and the bug comments for a link to the proposal).

As with most standards, it will take a while for this proper solution to saturate the marketplace. In the meantime, to get something like cross domain XHR requests

today, there are the following options:

- Set up a proxy server on the web page domain and have it forward the requests to the real XHR endpoint (requires server infrastructure).
- Use Flash (user has to have Flash installed).
- Use script tags (can do cross domain requests but return type must be JavaScript/JSON, and a callback mechanism needs to be established).

Another way to allow cross domain requests is to use the technique that is now available via dojo.io.XhrIframeProxy: use iframes that communicate with each other by changing URL fragment identifiers. This has the benefit of being just plain HTML and JavaScript (no additional server infrastructure or Flash), and it should be able to accommodate any asynchronous XHR request. It has been tested and works in IE 6.0, Firefox 1.5, Safari 2.0.3, and Opera 9.

It also contains a security mechanism that API providers can use to restrict the allowed cross-domain requests.

IFrames, Fragment Identifiers and XHR Proxying

Fragment Identifiers are the part of an URL that comes after the # sign:

http://www.a.com/path/to/file.html#fragmentIdentifier

A document in an IFrame can change the fragment identifier on its parent document (the document containing the IFrame). Changing the fragment identifier does not cause the page to reload. Similarly, the parent document can change an IFrame's fragment identifier without causing page reloads. Since the pages don't reload, state can be maintained inside the page.

To communicate between two cross domain documents :

- A document (the Client document) defines an IFrame that loads the other document (the Server document).
- Define a protocol to pass information through fragment identifiers.
- Tell each document about the URL for the other document (so they can set the fragment identifiers correctly -- the browser needs a complete URL when setting a cross domain location).
- Use a JavaScript timer to check for changes in the fragment identifiers.

To send an XHR request to another domain:

- Define a JavaScript object that implements the XHR interface (a Facade).
- Use that object instead of an actual XHR object.
- For the Facade's send() method, serialize the request headers, method, URL and data.
- The browser places a limit on the size of a document's URL, so the Client document breaks this serialized data into a set of fragment identifiers that

will fit under the URL limit.
- The Client document sends each fragment identifier to the Server document. The Server document sends an acknowledgement back to the Client, and the Client sends the next fragment identifier, until all are sent.
- The Server document assembles the fragment identifier parts into the original serialized data, unpacks it into an object, then uses a real XHR object (now on the Server's domain) to do the final API service call.
- The Server document then serializes the XHR response, and sends it back to the Client using fragment identifier segments.
- The Client unpacks the serialized response, and sets the appropriate values on the XHR Facade.

Trade-Offs

Pros

- 100% pure browser. No Flash or additional server infrastructure.
- It can be dropped in fairly transparently to code that is already using XHR.

Cons

- The technique uses IFrames and loads documents into the IFrames, so it takes more browser memory than native XHR. It would be interesting to compare the resource requirements with the amount needed to run Flash.
- More network traffic to download xip_client.html and xip_server.html (the contents of the IFrames). However, you can configure your web server to tell the browser to cache these files for a very long time.
- Timers are involved, with message serialization and deserialization.
- Setting all of those URLs in the IFrames causes MSIE to make lots of those "clicking" sounds (the sound normally to indicate to the user they clicked on a link).

Security Considerations

This approach does not allow cross domain access to any XHR-enabled API service. For it to work, the API service must place the Server document (web page) on its server. That web page is given the Client URL and the XHR request in serialized form, so it can restrict who can contact the service and what types of requests are allowed. Note that all request validation happens inside the Server document's JavaScript.

You should not experiment with this technique unless you are very restrictive on the clients and API URLs that are allowed. Placing the Server document on your web server means opening up the allowed URLs to the world.

Dojo Implementation/Examples

As of 7/31/2006, the Dojo tree has support for XHR IFrame Proxying. The relevant files are:

- src/io/XhrIframeProxy.js: the Dojo package, dojo.io.XhrIframeProxy, that provides the XHR Facade and manages the use of xip_client.html.
- src/io/xip_client.html: the Client document. Used internally by dojo.io.XhrIframeProxy.

- src/io/xip_server.html: the Server document. Used by API service providers to enable cross domain XHR requests.
- tests/io/iframeproxy: test files.

The test files are running here if you want to try it out (note that the API server for these tests is not a powerful box, so it may seem slower than usual to get the responses).

For web page developers

In addition to doing the normal things for dojo.io.bind(), do the following:

- dojo.require("dojo.io.XhrIframeProxy");
- Define an iframeProxyUrl parameter to dojo.io.bind(). This will be an URL to the xip_server.html file on the API service server.
- Only asynchronous XHR requests are supported.

Example code snippet:

```
dojo.require("dojo.io.*");

dojo.require("dojo.io.XhrIframeProxy");


dojo.io.bind({

iframeProxyUrl: "http://some.domain.com/path/to/xip_server.html",

url: "http:/some.domain.com/path/to/api",

load: function(type, data, evt, kwArgs){

/* do stuff with the result here */

}

});
```

For API service providers

API service providers will not care about src/io/XhrIframeProxy.js or xip_client.hml. They will be most interested in xip_server.html. For security reasons, xip_server.html will not run "out of the box". The following function needs to be defined:

```
function isAllowedRequest(request){

/* Decide if you want to allow the request. Return true or false */

}
```

By default, it is expecting this to be declared in an isAllowed.js file in the same directory as xip_server.html. See the comments in xip_server.html for more information.

Reusable Parts for Non-Dojo Implementations

- src/io/XhrIframeProxy.js: Provides the XHR Facade and manages the use of xip_client.html. It does not have all XHR methods defined, only the ones needed by Dojo's usage of XHR. You can look at the package code to see how it manages the Facade objects and the interaction with xip_client.html.
- src/io/xip_client.html: Does not depend on any Dojo files, but it makes a call to a Dojo function when it receives a response from the Server document. Just replace the function call to your own function. Used internally by XhrIframeProxy.js.
- src/io/xip_server.html: Does not depend on any Dojo files. Used for the final XHR request to the API service.

Introduction to I/O bind

The `dojo.io` package provides portable code for XMLHTTP and other, more complicated, transport mechanisms. Additionally, the "transports" that plug into it each provide their own logic to make each of them easier to use. The rest of this article will cover how the XMLHTTP transport from Dojo provides ways around the book-marking and back button problems.

Most of the magic of the `dojo.io` package is exposed through the `bind()` method. `dojo.io.bind()` is a generic asynchronous request API that wraps multiple transport layers (queues of iframes, XMLHTTP, mod_pubsub, LivePage, etc.). Dojo attempts to pick the best available transport for the request at hand, and in the provided package file, only XMLHTTP will ever be chosen since no other transports are rolled in. The API accepts a single anonymous object with known attributes of that object acting as function arguments. To make a request that returns raw text from a URL, you would call `bind()` like this:

```
dojo.io.bind({
    url: "http://foo.bar.com/sampleData.txt",
    load: function(type, data, evt){ /*do something w/ the data */ },
    mimetype: "text/plain"
});
```

That's all there is to it. You provide the location of the data you want to get and a callback function that you'd like to have called when you actually DO get the data. But what about if something goes wrong with the request? Just register an error handler too:

```
dojo.io.bind({
    url: "http://foo.bar.com/sampleData.txt",
    load: function(type, data, evt){ /*do something w/ the data */ },
    error: function(type, error){ /*do something w/ the error*/ },
    mimetype: "text/plain"
});
```

It's possible to also register just a single handler that will figure out what kind of event got passed and react accordingly instead of registering separate load and error handlers:

```
dojo.io.bind({
    url: "http://foo.bar.com/sampleData.txt",
    handle: function(type, data, evt){
        if(type == "load"){
            // do something with the data object
        }else if(type == "error"){
            // here, "data" is our error object
            // respond to the error here
        }else{
            // other types of events might get passed, handle them here
        }
    },
    mimetype: "text/plain"
});
```

One common idiom for dynamic content loading is (for performance reasons) to request a JavaScript literal string and then evaluate it. That's also baked into `bind`, just provide a different expected response type with the mimetype argument:

```
dojo.io.bind({
    url: "http://foo.bar.com/sampleData.js",
    load: function(type, evaldObj){ /* do something */ },
    mimetype: "text/javascript"
});
```

And if you want to be DARN SURE you're using the XMLHTTP transport, you can specify that too:

```
dojo.io.bind({
    url: "http://foo.bar.com/sampleData.js",
    load: function(type, evaldObj){ /* do something */ },
    mimetype: "text/plain", // get plain text, don't eval()
    transport: "XMLHTTPTransport"
});
```

Being a jack-of-all-trades, `bind()` also supports the submission of forms via a request (with the single caveat that it won't do file upload over XMLHTTP):

```
dojo.io.bind({
    url: "http://foo.bar.com/processForm.cgi",
    load: function(type, evaldObj){ /* do something */ },
    formNode: document.getElementById("formToSubmit")
});
```

Phew. Think that about covers the basics. Good thing you weren't planning on implementing all that stuff yourself, right?

RPC

As you have seen, Dojo provides powerful, yet simple, ways of performing a variety of I/O functions through the use of dojo.io.bind. However, during the

development of a typical application, a developer will have many I/O calls to make and will typically gravitate towards a common way of making those I/O calls on both the server and the client. This will often include defining functions that take some input and perform the appropriate request, as well as hooking that request to a callback function to process the results. In effect, the developer is required to implement a way of marshaling the request to the server in a way that it expects and then to have the client receive the contents in a way it expects. Dojo's RPC service aims to make this less error prone, easy to do, and require less code.

Remote Procedure Calls (RPC), also know as Remote Method Invocations, are a mainstay of the client/server development world. Essentially, RPC allows a developer to invoke a method on a remote host. Dojo provides a basic RPC client class that has been extended to provide access to JSON-RPC services and Yahoo services. It was designed so that it is also fairly trivial to implement custom RPC services.

Let's pretend that we have a little application that we want to make some server calls with. For simplicity's sake, we'll say the methods we want the server to do are add(x,y) and subtract(x,y). Without using anything special, like an RPC client, we might do something like this:

```
add = function(x,y) {
        request = {x: x, y: y};
        dojo.io.bind({
                url: "add.php",
                load: onAddResults,
                mimetype: "text/plain",
                content: request
        });
}
subtract = function(x,y) {
        request = {x: x, y: y};

        dojo.io.bind({
                url: "subtract",
                load: onSubtractResults,
                mimetype: "text/plain"
                content: request
        });
}
```

As you can see, this isn't particularly difficult. However, this is quite the simple application, despite our every attempt to make it complicated by having the server add or subtract two numbers instead of performing these operations in the client in the first place. What happens if our application is not so simple and has 30 different requests to make? I guess we would have to just write this same code over and over for each different request; each time making a request object, specifying URLs, potentially validating parameter types, and so on. This is simply error prone and boring to write.

Dojo's RPC clients simplify this whole process by taking a simple definition of the remote methods and application needs and generating client side functions to call these methods. A developer need only write this definition, and initialize a RPC client object and then all of these remote methods are available for the developer to use as normal.

The definition file, called a Simple Method Description (SMD) file, is a simple JSON string that defines a URL that will process the RPC requests, any methods available at that URL, and the parameters those methods take. The definition for our example above might look like this:

```
{
        "serviceType": "JSON-RPC",
        "serviceURL": "rpcProcessor.php",
        "methods":[
                {
                        "name": "add",
                        "parameters":[
                                {"name": "x"},
                                {"name": "y"}
                        ]
                },
                {
                        "name": "subtract",
                        "parameters":[
                                {"name": "x"},
                                {"name": "y"}
                        ]
                }
        ]
}
```

Once the definition has been created, the code its pretty simple. The definition can be supplied either as a URL to retrieve it, a JSON string,Ã‚Â or a JavaScript object.

```
var myObject = new dojo.rpc.JsonService("http://localhost/definition.smd")
var myObject = new dojo.rpc.JsonService({smdStr: definitionJSON});
var myObject = new dojo.rpc.JsonService({smdObj: definition});
```

Thats it! Now all thats left is to call the method.

```
myObject.add(3,5);
```

I'll bet you are saying to yourself, "Nice try, but I want to get the results of the add method, not just call it." You are correct, but that is also simple to achieve. Recall that we are making asynchronous calls to the server. While we could make the request synchronous, it would likely provide for a bad user experience because it would block the user interface during the call. Instead, the return value of the myObject.add() call, is a deferred object. The deferred object, something that might be familiar to users of Twisted Python or MochiKit, allows a developer to attach one or more callbacks and errbacks to the resultant data event. Our simple example can be expanded as such:

```
var myDeferred = myObject.add(3,5);
myDeferred.addCallback(myCallbackMethod);
```

or more succinctly:

```
var myDeferred = myObject.add(3,5).addCallback(myCallbackMethod);
```

As you can see, we've added myCallbackMethod as a callback for the deferred object returned from myObject.add(). In this case myCallbackMethod will be

called with parameter with a value of 8. Likewise, an errback method can be attached to the deferred object to process an errors returned from the server. We can add as many callbacks and errbacks to our deferred object as we want and they will be called in the order that they were connected to the deferred object.

This discussion has revolved around using dojo.rpc.JsonService, which is Dojo's JSON-RPC client. In addition to JsonService, Dojo offers an RPC client for connecting to Yahoo services, dojo.rpc.YahooService. The syntax and call structure is identical. While Dojo is currently limited to these two RPC clients, the design of the dojo.rpc.RpcService base class, which is inherited by dojo.rpc.JsonClient and dojo.rpc.YahooService allows a developer to easily customize and extend dojo.rpc.RpcService, to create services that meets their specific needs. These customizations will be discussed later in Part II when we discuss how to get the most out of Dojo.

Transports

dojo.io.bind and related functions can communicate with the server using various methods, called transports.Ã‚Â Each has certain limitations, so you should pick the transport that works correctly for your situation.

The default transport is XMLHttp.

IFrame I/O

The IFrame I/O transport is useful because it can upload files to the server.Ã‚Â Example usage:

The response type from the above URL can be text, html, or JS/JSON.

IframeIO responses need to be a little different from the ones that are sent back from XMLHttpRequest responses. Because an iframe is used, the only reliable, cross-browser way of knowing when the response is loaded is to use an HTML document as the return type.

If the return type (specified by the mimetype) is text/plain, text/javascript or text/json, then the server response should be an HTML page that has a

```
 element. The data
that you want
```