

From Technologies to Solutions

Learning Dojo

Build a Great Web Experience with Simple Dojo and JavaScript Techniques

Peter Svensson

PACKT PUBLISHING

Learning Dojo

Build a Great Web Experience with Simple Dojo and JavaScript Techniques

Peter Svensson

PACKT PUBLISHING

BIRMINGHAM - MUMBAI

Learning Dojo

Copyright © 2008 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, Packt Publishing, nor its dealers or distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: November 2008

Production Reference: 1201108

Published by Packt Publishing Ltd.
32 Lincoln Road
Olton
Birmingham, B27 6PA, UK.

ISBN 978-1-847192-68-4

www.packtpub.com

Cover Image by Raghuram Ashok (raghuram@iiitb.ac.in)

Credits

Author

Peter Svensson

Reviewers

Eugene Lazutkin

Meenakshi Singh

Acquisition Editor

Rashmi Phadnis

Development Editor

Usha Iyer

Technical Editor

Dhiraj Bellani

Editorial Team Leader

Mithil Kulkarni

Project Manager

Abhijeet Deobhakta

Project Coordinator

Leena Purkait

Indexer

Rekha Nair

Proofreader

Angie Butcher

Production Coordinator

Aparna Bhagat

Cover Work

Aparna Bhagat

About the Author

Peter Svensson is an Ajax evangelist and front-end architect at Nethouse AB. He is the instigator of the Thin Server Architecture Working Group and a Dojo contributor. He's an avid JavaScript aficionado who really likes dynamic languages and would very much appreciate if the rest of the world would catch up really soon.

I'd like to thank my wonderful family; my son Adam for being the best pokémon master in the universe, my daughter Elsa for not sounding like a vildvittra most of the time, my wife Carina for being completely wonderful, and my mother Agneta for picking up our kids from school umpteen times a week.

About the Reviewer

Eugene Lazutkin is a veteran software developer based in Dallas, Texas. Working in the web development business for a long time, he is a firm believer in the power of Open Source, and an active participant in open-source projects. As a committer of the Dojo Toolkit, he is responsible for the Dojo DnD module (part of the Dojo Core), the `dojox.gfx` module (portable cross-platform browser-based vector graphics supporting SVG, VML, Silverlight, and Canvas), the charting module (based on `dojox.gfx`, it brings the power of interactive graphics to dynamic web applications), and numerous language-related enhancements (the functional programming module, the aspect-oriented programming module, and so on). While not programming for SitePen or working on improving Dojo, he writes technical articles that can be found at his company blog at `http://blog.sitepen.com/`, or in his personal blog at `http://lazutkin.com/`.

I want to thank my wife Svetlana, and my daughter Christina for putting up with my long hours, all Dojo contributors (`http://dojotoolkit.org/dojo-contributors`) for making this very book possible, and fellow committers for clarifying some fine points for me when I was reviewing this book.

Table of Contents

Preface 1

Chapter 1: Introduction to Dojo 5

Dojo usage 6

A sneak peek at Dojo 6

Dynamic charting 8

StackContainer wizard 10

What Dojo is (and isn't) 14

The Dojo foundation 15

How to participate 16

Feature overview 16

Dojo compared to other Ajax frameworks 21

Server-side compatibility 21

Reference sites, API listings, forums, and mailing list 22

Summary 22

Chapter 2: Useful JavaScript and Dojo Tricks 23

Array usage tips and gotchas 28

Functions as arguments 29

JavaScript's prototype-based OO 31

Using closures 33

OO in Dojo 35

Anatomy of a Dojo page 38

Basic Dojo functions 43

Intermediate Dojo functions 44

Logging, Firebug integration, and debugging 45

Event handling in Dojo 49

JSON 54

Summary 56

Chapter 3: Basic Dijit Knowledge 57
The two ways to create Dijits 59
How do I find my widget? 60
What's in the fridge? A smorgasbord of tasty Dijits 61
Dijit structure 64
Lifecycle 64
Templates 66
Attach points 67
Events 68
Extension points 69
Django Templating Language (DTL) 70
Variable substitution 71
Loops 72
Conditionals 72
Argument passing 72
A recommended widget structure 72
Themes 76
I18N 78
djConfig.locale 79
Programmatic locale formatting 80
Accessibility and graceful degradation 81
Summary 81
Chapter 4: Ajax Communication 83
Dojo Ajax features 83
The "Hello World" of Ajax 86
Errors 89
Posting data 89
The content parameter 92
Content-types 93
Script request/JSONP (dojo.io.script) 95
Remote procedure calls 96
Summary 98
Chapter 5: Forms 99
The Form Dijit 99
Generic properties for all Form Dijits 104
Textfields and validation 104
CheckBoxes, radio buttons, and toggle buttons 111
Spinners, sliders, and FilteringSelects 113
TimeTextBox 116
Buttons 117

Ajax integration **119**
Summary **120**
Chapter 6: Layout **121**
Basic Dojo layout facts **122**
ContentPane **122**
Container functions **124**
LayoutContainer 124
SplitContainer 124
StackContainer 125
TabContainer 130
AccordionContainer 131
BorderContainer 132
DragPane **135**
ExpandoPane **136**
FloatingPane **138**
GridContainer **140**
RadioGroup **145**
ResizeHandle **148**
RotatorContainer **150**
ScrollPane **152**
Compound example using layout **153**
The Toolbar 157
The BorderContainer 157
The AccordionContainer 157
The ScrollPane 158
One step forward—creating a widget **158**
Summary **165**
Chapter 7: Data, Trees, and Grids **167**
dojo.data and dojox.data **167**
Quick example **169**
Models **170**
Data sources, stores, and consumers **171**
dojo.data.Read 171
dojo.data.Write 174
dojo.data.Identity 175
dojo.data.Notification 175
Common functions for all APIs 176
A simple tree example **177**
Forests and trees **181**
Tree events 182

Tree styling 184
Tree drag and drop 187
Reading and writing with JsonRestStore 190
Overview of the Grid Dijit 194
A simple Grid 195
Grid views 197
Grid custom rendering 200
Grid sorting 202
Cell editing and styling 203
Summary 205

Chapter 8: Real-World Dojo 207
Using AOL's and Google's CDN 207
Dojo's build system and cross-domain loading 209
Using unit testing with Dojo Objective Harness (DOH) 212
Example 1: Site navigation scaffolding 215
Example 2: Multi-charting application 224
Example 3: CRUD application 233
Summary 243

Index 245

Preface

Dojo is arguably the most feature-packed Ajax toolkit available today, with a wide variety of functionality. It can be used sparsely to just add effects or styling to your web page, or it can be used for complex layouts, caching data stores, and interactive 2D graphics—the choice is yours. This book aims to give an overview of all major aspects of Dojo while at the same time giving lots of examples to copy for your own studies and websites.

What this book covers

Chapter 1 Introduction to Dojo: This chapter will give you an overview of the Dojo Ajax framework along with tips and tricks for usage of JavaScript and useful and generic Dojo functions.

Chapter 2 Useful JavaScript and Dojo Tricks: This chapter explains Dojo's approach to object-oriented programming with JavaScript.

Chapter 3 Basic Dijit Knowledge: Dijits is Dojo's name for Widgets. A typical widget is a colorpicker or a movable pane. This chapter introduces Dijits in general and gives a lot of examples on how to use them, how they interact, and even an introduction to writing your own.

Chapter 4 Ajax Communication: This chapter covers Ajax communication in several aspects, with examples, which also show the view from the server (in PHP code).

Chapter 5 Form-Related Dijits: This chapter focuses on serializing forms and powerful validation and Internationalization. It covers best practices when setting up Dojo forms and includes a run-down of available components.

Chapter 6 Layouts: The Layout Dijit organizes the content, be it other Dijits or plain HTML. This chapter shows a number of examples from simplifying layout in general with 'CSS-less' layout managers, to dynamic loading of content in content panes, and how to create a wizard widget with a StackContainer and a small amount of scripting.

Chapter 7 Data, Trees, and Grids: The datastores are perhaps the most complex area of Dojo, but they also give the programmer an unprecedented level of modularity. We'll go over a couple of simple examples, then describe the `dojo.data` interface, how to extend them, and how to use several different types of data in the same Dijit component.

Chapter 8 Real-World Dojo: This chapter is almost entirely examples, and focuses on giving the reader full-blown guides on how to create several types of applications with Dojo, and will also be a store for cut-and-paste problem solving when time is of essence.

What you need for this book

You need a computer connected to the Internet that regardless of operating system has a modern web browser (IE6, Firefox 1.5, Safari 1.3.2). Also, you need to know at least one programming language, preferably with a C-like syntax (C++, C#, Java). To follow all the examples, you need to download the latest source version of Dojo from `http://download.dojotoolkit.org/current-stable/`. At the time of writing of this book, that file was called `dojo-release-1.2.1-src.zip`.

Who this book is for

This book is for web developers with JavaScript knowledge, who want to use Dojo for developing dynamic Web 2.0 applications.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: " Here, we used the Dojo function `dojo.byId()`, which is a shorter way of getting a handle for an element, instead of writing `document.getElementById()`."

A block of code will be set as follows:

```
function foobar(a, b, userFunction)
{
   ....
   var x = doSomething(a, b);
   userFunction(x);
}
```

New terms and **important words** are introduced in a bold-type font. Words that you see on the screen, in menus or dialog boxes for example, appear in our text like this: "When you click one of them, though, you will always get the alert **Button 3 was clicked**".



Important notes appear in a box like this with the Notes icon on the left.



Tips and tricks appear like this with the Tips icon on the left.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book, what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply drop an email to `feedback@packtpub.com`, making sure to mention the book title in the subject of your message.

If there is a book that you need and would like to see us publish, please send us a note in the **SUGGEST A TITLE** form on `www.packtpub.com` or email `suggest@packtpub.com`.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code for the book

Visit `http://www.packtpub.com/files/code/2684_Code.zip` to directly download the example code.

The downloadable files contain instructions on how to use them.

Errata

Although we have taken every care to ensure the accuracy of our contents, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in text or code—we would be grateful if you would report this to us. By doing this you can save other readers from frustration, and help to improve subsequent versions of this book. If you find any errata, report them by visiting `http://www.packtpub.com/support`, selecting your book, clicking on the **let us know** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata added to the list of existing errata. The existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide the location address or website name immediately so we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with some aspect of the book, and we will do our best to address it.

1
Introduction to Dojo

For a budding Ajax developer, there is plenty to choose from. Even before the term was coined in 2005, there has been an abundance of JavaScript libraries available, most of them geared to particular tasks.

Lately, what came to be known as Ajax libraries or Ajax frameworks have taken on a more general nature. They are not only hand-crafted to hide the complexities of client-server communication, building sortable and dynamic tables or providing dynamic menus. The Ajax frameworks of today aspire to be one-stop shops doing everything from class-based inheritance and generic events to encryption, real-time 3D Graphics, and in-browser data buses.

Of all these frameworks, Dojo is perhaps the most generic, tracing its roots to not one but several different JavaScript libraries of yore. As you're reading this book, you have at least decided to take a look at what Dojo can offer, in terms of features, ease of use, and developmental complexity (or lack thereof). Maybe you have already decided to use Dojo, based on a special need that only Dojo can help you with, or for the general abundance of features.

The version of Dojo used in this book is post 1.1, up to 1.2RC1. Since Dojo is continually in rapid development, with a target of three months between every minor release, I wanted to include as many 'goodies' as possible during the writing of this book. Make sure to go to the website of this book to take part in any errata or updates made available since it went in print. Also, to dowload the examples, use the following URL: `http://www.packtpub.com/files/code/2684_Code.zip`.

Dojo usage

Dojo can be used a piece at a time. Just because you use Dojo to handle Ajax communication with the server, does not mean that you have to use it for event management of button clicks or drawing diagrams, but you can if you want. Dojo plays nice with other toolkits and technologies and can be used as little or as much as you would like.

The most common scenario is probably to use Dojo to add a specific function to an existing website. Examples of this are:

- An animated navigational menu for different pages of the site.
- Draggable 'windows' of information for a particular page.
- A sortable table of products.
- A combobox which gets all selectable items asynchronously from the server at runtime.

Perhaps the site is already using other toolkits like YUI or jQuery, but Dojo is chosen for a specific component that is not found in the toolkits already present.

More and more, Dojo is used to write true clients which are downloaded to the browser once, in the shape of an HTML file which refers to Dojo's JavaScript and CSS files, and which, after that only, communicates with the server using Ajax technology. This approach is very efficient when replacing or rewriting existing desktop applications to be used on the web.

The book will cover all these examples and many more, with three practical full-length examples in the last chapter, which are meant to be used as templates for your own needs.

A sneak peek at Dojo

Some Ajax frameworks provide functionality that can be used in other functions to manipulate the web page and achieve certain effects, whereas others mainly use specialized markup inside the page itself to pursue the same goals. Dojo is a mixture of both approaches.

Certainly, Dojo has a lot (if not, the majority) of functions designed to be run inside a script, or merely query, affect, or manage the web page from the outside. However, when using Dijits, as Dojo's widgets are called, there always exists the secondary (or perhaps primary) option of embedding the Dijit inside a 'normal' HTML element in the web page.

We'll start the book by quickly going over some small but hopefully explanatory examples, without going into too much detail for each This will help you to get your bearings around what Dojo can do for you and how hard it will be to get there.

Note that the source code and explanations will contain quite a lot of unfamiliar literals and references, if you are new to JavaScript and/or Dojo. The intention here is not to befuddle you with arcane notation, but merely to show not only what Dojo looks like, but also what the source code looks like, at a glance. Therefore, the explanations below are fairly short and focused more on describing what is being done rather than every single detail, which I promise you will find in coming chapters.

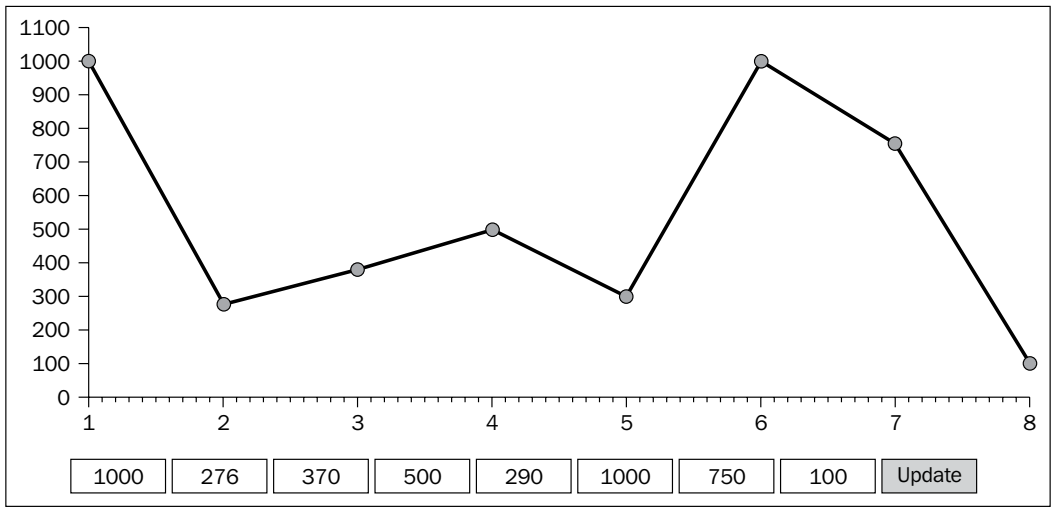
The base footprint of Dojo is about 25kb gzipped and it includes the following functionality:

- JSON encoding/decoding
- Browser detection
- Package loading
- Powerful Ajax support
- Unified events
- Animation support (including color animations)
- Asynchronous programming support
- High-performance CSS3 query engine
- Language utilities
- CSS style and positioning utilities
- Object-oriented programming (OOP) support
- Memory leak protection
- Firebug integration

Dynamic charting

To give you an example of a web page using Dojo, consider the dynamic charting page below.

Dojox has an extremely powerful charting package, which can not only sketch the single line chart shown below, but also pie charts, scatter chart, 3D projection, and much more. In this small example, I try to show how simple it can be to draw a fairly nice looking graph from just an array of values.




```
<script>

makeObjects = function()
{
    var arr = [1000, 276, 370, 500, 290, 1000, 750, 1100];
    var chart = new dojox.charting.Chart2D("test");
    chart.addAxis("x", {fixLower: "major", fixUpper: "major"});
    chart.addAxis("y", {vertical: true, fixLower: "major",
                        fixUpper: "minor", includeZero: true});
    chart.addPlot("default", {type: "StackedLines", markers: true,
                              shadows: {dx: 2, dy: 2, dw: 2}});
    chart.addSeries("My Values", arr, {stroke: {color: "blue",
                    width: 2}, fill: "white", marker: "m-3,0 c0,-4
                    6,-4 6,0 m-6,0 c0,4 6,4 6,0"});
    chart.render();

    function update()
    {
        var newseries = [];
        for(var i = 0; i < 8; i++)
```

```
            {
                var id = i+1;
                id = "v"+id;
                var x = dojo.byId(id).value;
                newseries.push(x);
            }
            chart.updateSeries("My Values", newseries);
            chart.render();
        }

        dojo.connect(dojo.byId("button"), "onclick", update);
    };
    dojo.addOnLoad(makeObjects);
    </script>
    </head>
    <body>
    <h1>Diagram</h1>
    <br/>
    <div id="test" style="width: 760px; height: 250px;"></div>
    <div style="margin-left:50px;">
    <input type="text" id="v1" size="8" value="1000"/>
    <input type="text" id="v2" size="8" value="276"/>
    <input type="text" id="v3" size="8" value="370"/>
    <input type="text" id="v4" size="8" value="500"/>
    <input type="text" id="v5" size="8" value="290"/>
    <input type="text" id="v6" size="8" value="1000"/>
    <input type="text" id="v7" size="8" value="750"/>
    <input type="text" id="v8" size="8" value="1100"/>
    <button id="button">Update</button>
    </div>
```

This first part of the source code shows a custom JavaScript function called `makeObjects`, which is run after the page is safely loaded, as the call to `dojo.addOnload` ascertains.

When called, it creates two objects: an array which is filled with eight integer values, and a `dojox.charting.Chart2D` object. The argument to the constructor is the DOM id of the element where we want the chart created, and the chart will automatically detect the dimension and fit inside the element provided.

The next two lines define how the `y-` and `x-`axis will be shown, and a definition of how the lines are to be plotted in the chart. This definition can become indefinitely complex, so I won't belabor you with the finer points until a later chapter. Feel free to poke around and change things to get other interesting results, if you please. You might then be interested to look into the source code of the test directory under charting, which can be found under your Dojo directory at: `dojox/charting/tests/test_chart2d.html`.

Then we add the actual data in the call to `addSeries`. Since we might want to plot several arrays of values on the same chart, it is possible to call `addSeries()` several times for a chart, and to give each 'series' a different look, hence the arguments (`{stroke: {color: "blue", width: 2}`, for instance) can vary between series in the same chart.

If you feel that the syntax of the calls look weird, just leave it at that for now since we'll be covering quite a lot of JavaScript object and array notation as well as some recurring use of them in different parts of Dojo in Chapter 2.

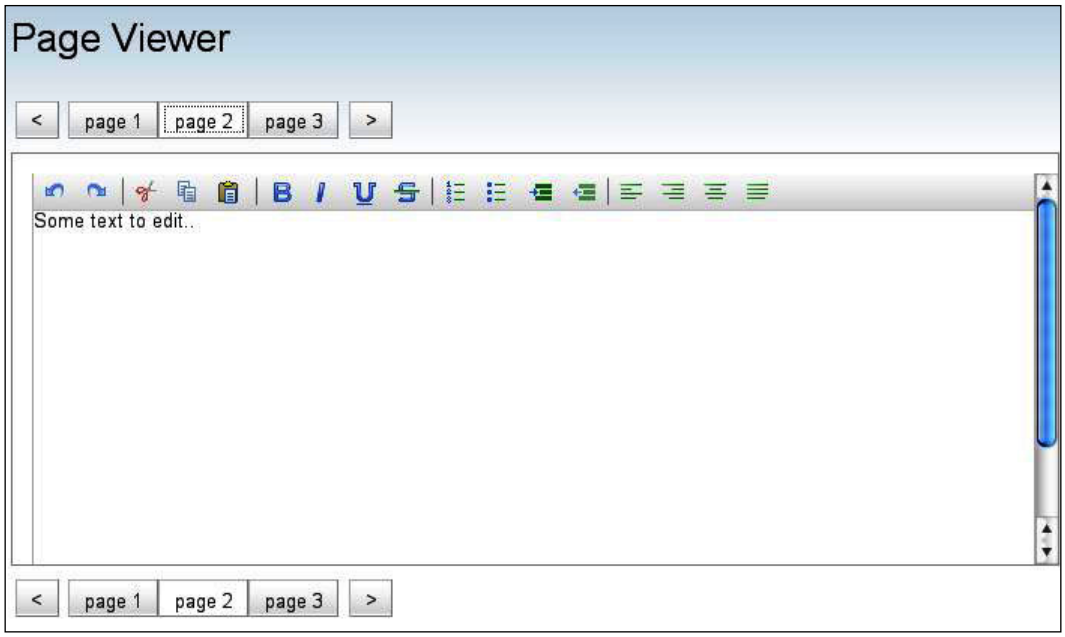
We also have a function defined called `update`, which is connected to a button element in the DOM at the end of the script section. This function just collects the data in the eight input fields and creates a new array, which we then update our already created chart object with. This is a very powerful feature, which lets you modify charting data in the browser, at user discretion, without resorting to Flash objects or heavy-handed and static server-side generation.

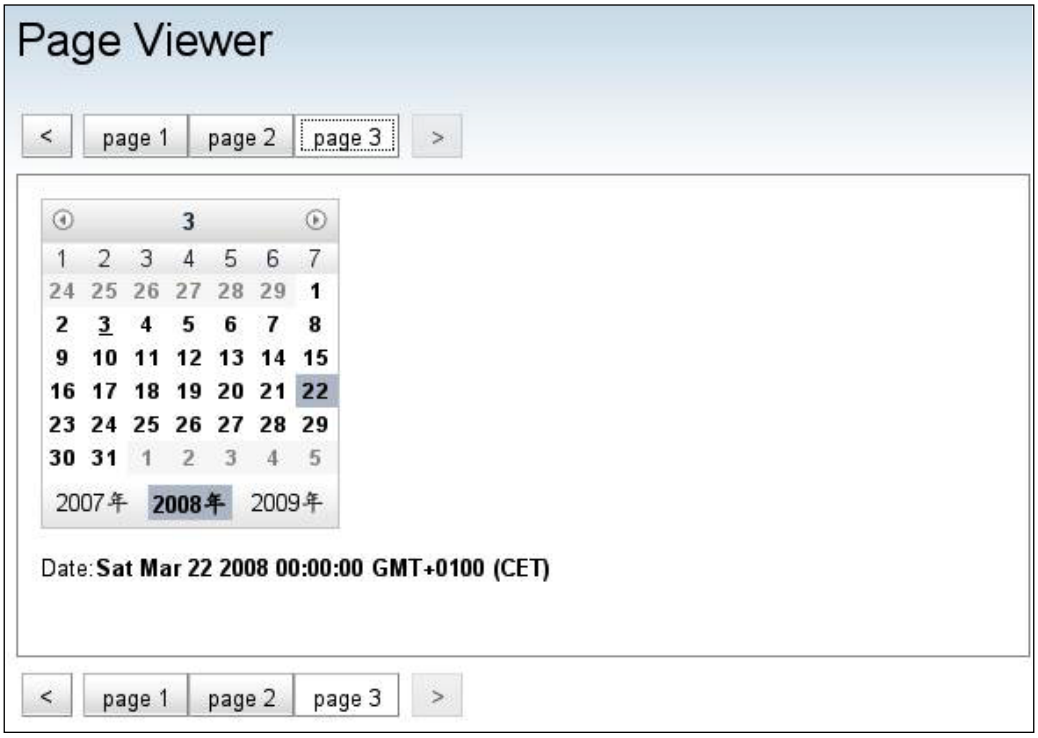
StackContainer wizard

The main feature of this example is the StackContainer. It is the base for the TabContainer which lets you arrange content on your page in different tabs that can be user-selectable.

However, many times you want to control how the different tabs are shown, and display them by other means than a tab button. In these cases, the StackContainer can be used to show a Wizard-like interface with different views or states.




Page Viewer
< page 1 page 2 page 3 >
Some text to edit..
< page 1 page 2 page 3 >


Page Viewer
< page 1 page 2 page 3 >
3
1 2 3 4 5 6 7
24 25 26 27 28 29 1
2 3 4 5 6 7 8
9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30 31 1 2 3 4 5
2007年 2008年 2009年
Date:Sat Mar 22 2008 00:00:00 GMT+0100 (CET)
< page 1 page 2 page 3 >

```
<head>
      <script>
          function selected(page)
          {
              console.debug("page selected " + page.id);
              var widget=dijit.byId("myStackContainer");
              dijit.byId("previous").setDisabled(page.isFirstChild);
              dijit.byId("next").setDisabled(page.isLastChild);
              dijit.byId("previous2").setDisabled(page.isFirstChild);
              dijit.byId("next2").setDisabled(page.isLastChild);
          }
          function setColor(color)
          {
              console.log("setColor called with "+color);
              var theSpan = dojo.byId("outputSpan");
              theSpan.style.color = color;
              theSpan.innerHTML = color;
          }
          function myHandler(id, what)
          {
              dojo.byId('calendarSpan').innerHTML = what;
          }
             dojo.subscribe("myStackContainer-selectChild", selected);

      </script>
</head>
<body class="tundra">
<h1 class="testTitle">Page Viewer</h1>
<button dojoType="dijit.form.Button" id="previous"
onClick="dijit.byId('myStackContainer').back()">&lt;</button>
<span dojoType="dijit.layout.StackController" containerId="myStackCont
ainer"></span>
<button dojoType="dijit.form.Button" id="next"
onClick="dijit.byId('myStackContainer').forward()">&gt;</button>
<div id="myStackContainer" dojoType="dijit.layout.StackContainer"
style="width: 90%; border: 1px solid #9b9b9b; height: 20em; margin:
0.5em 0 0.5em 0; padding: 0.5em;">
<div id="page1" dojoType="dijit.layout.ContentPane" title="page 1">
<div dojoType="dijit.ColorPalette" onChange="setColor(this.value);">
</div>
<div>
Test color is: <span id="outputSpan"
style="height:15px;width:100px;">
</span>.
```

```
</div>
</div>
<div id="page2" dojoType="dijit.layout.ContentPane" title="page 2">
<div dojoType="dijit.Editor" >
Some text to edit..
</div>
</div>
<div id="page3" dojoType="dijit.layout.ContentPane" title="page 3">
<input id="calendar3" dojoType="dijit._Calendar"
onChange="myHandler(this.id,arguments[0])" >
<br/>
Date:<b><span id="calendarSpan"></span></b>
</div>
</div>
<button dojoType="dijit.form.Button" id="previous2" onClick="dijit.
byId('myStackContainer').back()">&lt;</button>
<span dojoType="dijit.layout.StackController" containerId="myStackCont
ainer"></span>
<button dojoType="dijit.form.Button" id="next2" onClick="dijit.
byId('myStackContainer').forward()">&gt;</button>
</body>
```

You can also see that the StackContainer is defined using a special markup property called `DojoType`.

```
   <div id="myStackContainer" dojoType="dijit.layout.
StackContainer" ...>
```

All Dijits are declared this way when created in the markup of the page itself. All Dijits (and many supporting objects) can be defined in one of the two ways—using the markup property or using a programmatic instantiation inside a script.

When declarative definition is used, Dojo uses its parser to parse the entire page, once it has finished loading. Then Dojo creates any Dijits that are referenced in tags with the `dojoType` parameter. Any arguments needed are put verbatim as custom properties after the `dojoType` declaration (or before, it doesn't really matter, but I hope you get the point).

Though not needed, you can also choose to use the StackController, which, given a containerId property that contains the id of our StackContainer, creates a simple interface that lets the user choose which part of the Stack to show.

Now over to what's actually being shown. You might react to the fact that the whole stack is just a series of nested divs. The outermost div is the StackContainer, and the content-carrying divs are defined as ContentPanes.

ContentPanes show up a lot when you're doing Dojo layout. The definition of a ContentPane is basically something that only holds content, with no fancy stuff. They're used in LayoutContainers and are the things that the containers arrange.

To have something to show you other than text, I chose three widgets that are very graphic and at the same time very easy to create: the ColorPicker, the Dojo text editor, and the Calendar widget, configured for chinese on a lark.

The functions defined in the script tag are management for the StackContainer buttons, and callback functions for the calendar and colorpicker, which display their current values.

What Dojo is (and isn't)

Dojo is a client-side JavaScript framework. That means that it is primarily designed to run inside a web browser, even though parts of Dojo can (and do) run inside some of the existing server-side JavaScript frameworks [`http://en.wikipedia.org/wiki/Server-side_JavaScript`].

On the other hand, some parts of Dojo, particularly the Ajax and data store functionality, support certain formats to be provided on the server-side. In the source distribution of Dojo, there are several sample server-side implementations to show the intrepid developer how to go about creating his or her own, although these implementations are not considered as parts of Dojo itself.

Dojo is furthermore casting a wide net, with a wide range of functionality. Most probably you will only use some kinds of functions in a single web application, but there is a lot of gold in the tunnels.

Much work has been made to make the different parts use similar or the same interfaces, use the same names for similar functionality, and to let them be invoked in the same way.

Still, there are many parts of Dojo that are specialized and separate from the rest of the framework (like the offline storage or encryption modules), as well as quite many which are interrelated. (Drag and drop functionality can be used together with the Tree widget, for example.)

When you look at the `dojo` directory, you find the following four subdirectories:

- `dojo`: Here lies all the basic and 'official' Dojo functionality that is not directly tied to widgets.
- `dijit`: Dijits is what Dojo calls its widgets. So this directory is filled to the brim with cool and useful widgets of all kinds.
- `dojox`: `dojox` is Dojo's response to area 51. Present here are new and experimental features provided as-is, with not too much documentation.
- `util`: The `util` directory contains diverse things that are not really used inside a web-page like build-script for customizing your own version of Dojo, logotypes, and such.

The Dojo foundation

The Dojo foundation, modeled after the Open Source Foundation, is a 501c6 not-for-profit legal entity set up for the administration of the Dojo Ajax framework.

Among the members of the foundation, we find such prestigious companies as IBM, Sun, Bea, and many others. The foundation not only manages Dojo, but several related projects such as OpenRecord, cometd (a server-side stand-alone implementation of the "reverse-ajax" Bayeux protocol) and DWR (Direct Web Remoting).

For all projects, the foundation has several rules, which make certain that the software it manages remain open source. Chief of these is the contributor license agreement (CLA) for both individuals and corporations who wish to donate resources (mostly source code fixes or features) to the foundation. For more on this, please visit `http://www.dojotoolkit.org/foundation`.

To force individuals who just want to participate to sign an agreement first might seem a little odd, but the reasons for this are mainly to guarantee, for instance, that the employer of an individual contributor later can claim that parts of Dojo actually belong to them, and related issues. So, from a consumer perspective, this is a very wise decision.

Furthermore, Dojo is available in two different licenses: the Academic Free License 2.1, and the BSD license, both of which are extremely permissible and royalty-free.

How to participate

Many parts of the Dojo framework have been donated from users or organizations over the years. If you have created something that you feel would be a good contribution to Dojo (or one of the other projects), use the forums on the site to announce your intentions.

Most probably, you'll first start to use Dojo for some very specific purposes, and later start to test some cool-looking parts in your spare time. Before you know it, you'll have found the need to add some extra features or maybe you've found the rare bug! In either case, you'll probably post about your findings in the forums, which can be found at `http://dojotoolkit.org/forum`.

In general, if you have a question or a problem with Dojo, it is always recommended to use Google to search for an answer first. Then choose one of the following options, with Forums being the preferred alternative:

- Post the question to the correct forum at `http://dojotoolkit.org/forum`.
- Use IRC to ask your question on the #dojo channel `in irc.freenode.net`.
- Join the mailing-list `dojo-interest`, which can be joined at `http://dojotoolkit.org/mailman/listinfo/dojo-interest`.

Always make sure that you have checked for typos in your code, and that you have a clear explanation of what you want to achieve. It is very hard to answer questions of the type "How do I get names and addresses out of my Oracle database with Dojo?", since that leaves more than a fair amount of work on the shoulders of the poor person trying to make heads or tails of your question.

Feature overview

The comittee of Dojo is focused on making it an easy-to-build web application and that's fortunate considering the sheer size of the framework.

Even though Dojo is mind-bogglingly huge , the basic runtime is merely 24kB in size, and each page uses the `require()` function to load only those parts of Dojo which are needed, making Dojo use only as many resources as it needs to in any given page.

The following list is by no mean exhaustive, but does cover the major "selling points" that Dojo offers. This book will not cover each item in depth, or indeed at all, but it is good to know that something that you need exists to begin with. That way you can ask about it in the forums later, or just browse the tests in the source distribution.

The list of features, then, with no further ado:

Feature	Comment
Namespacing	Dojo tries its utmost to tread very light inside the browser. One of the basic tenets is that **everything** should have a namespaced name, so that the name does not collide with some other library loaded before or after. This is actually quite a big issue which has caused a large amount of grief in the past. This means, for example that the function for looking up an element based on its id is called `dojo.byId(x)` in Dojo, whereas some other frameworks have shorter names, due to lack of namespacing.
Object-Orientation	Dojo provides functions for creating OO-like inheritance on top of JavaScript's Prototype-based inheritance, as well as encapsulation and mixin classes.
Powerful Widget system.	The system used inside Dojo for defining Dijits, Dojo's Widgets, is open and extensible. This means that it is really simple to define a new widget which is based upon an existing one. Say that you want to create a 'Wizard' widget. You could then create a new widget which inherits from StackContainer, and only add logic for the handling of merchandise and communication with the Payment Service of your choice. The widget definition system itself is also very expressible and simple to use. Each widget defined can use Dojo's autoloading features (`dojo.require()`) and can be modularized with its own CSS file(s) and HTML template file, where simple variable substitution creates customizable looks with very little effort. I would like to think that if I had to list the top-five "killer applications" in Dojo, the Widget system would be pretty near to the top.
Event system	Dojo's event system began, by necessity, as all other Ajax frameworks do, by hiding the intricacies event handling difference between different browsers. (OK, differences in Internet Explorer) But nowadays, Dojo lets you manage almost anything as an event. You can, for instance, let a function call be an event so that you can make Dojo call your own custom function anytime another function is called. Another really nice feature is the publish/subscribe message bus which most 'official' widgets use to declare state changes to other parts of the system, or to the interested function in general. This makes it very simple to add some custom functionality when a tab is selected, menu chosen, or a draggable item dropped, without having to define your own widget class.

Feature	Comment
Drag and Drop	Speaking of Drag and Drop, Dojo has that in spades. Not only is it very simple to define any element on the page as draggable or as a drop target, it is also possible to restrict certain drop targets to only accept certain draggables. It is also possible to use Tree nodes together with Drag and Drop, so you can drag a node from one tree and drop it on a branch of another, effectively moving or copying it. The Drag and Drop system also has ample support for custom avatars for draggable items, which shows them in alternate color, pictures, or animation, as you wish, before, during, or after the drag.
`dojo.data`	`dojo.data` is a generic data layer between components that present complex data in any form and the data sources themselves. This sounds pretty academic, but is really a very neat idea. Using `dojo.data` means that if you can write your own `dojo.data` adapter for your custom data that comes out of your server, you can plug it right into the Tree or Grid components. Out-of-the-box Dojo provides an ItemFileReadStore and an ItemFileWriteStore, which can be used for accessing files containing data in JavaScript JSON notation on the server, or locally (while testing). There also exists experimental data stores in the `dojox` package called CsvStore, HtmlStore, XmlStore, PicasaStore, and FlickStore, to name a few. Most probably you will begin by making your server handle the format of the ItemFile*Stores or the XmlStore, and with some luck you will not need to do more than that.
XHR (Ajax)	This one is maybe too obvious to even fit in a list of features. I mean, this **is** an Ajax framework, isn't it? Of course, it has Ajax functionality! But the kind of functionality it has is a bit above and beyond just getting stuff from the server asynchronously. Dojo's Ajax functions can also define which kind of data is transported; various JSON variants, text or XML, and one of my favorite features is the ability to simulate form posting, without having to use a form at all, with the `postData` argument.

Feature	Comment
Internationalization (I18n)	All relevant Dijits in Dojo have support for internationalization. Dojo has support for setting the locale for the whole page or for individual widgets. The locale choice will then define how the widget formats date, time, and currencies, as well as management of text in bi-directional languages such as Arabic or Hebrew. Since locale definitions are stored in separate files, this system can be extended and reused for new widgets needing other domain-specific formats or strings.
Simple animations	Using slides, wipes, and fades for elements is a good way to get users' attention. The `dojo.fx` module lets you use single effects, properly timed, and also allows you to chain them together, to be played later or repeated.
Debugging	Dojo has its own debugging system, which is critical when you are forced to develop a browser which lacks a proper debugging environment. `dojo.debug` displays any console.log statements you put in your code in a separate div inside the browser window, in pretty much the same style as the firebug JavaScript debugger.
OpenAjax	OpenAjax is an organization that is devoted to standardizing the way Ajax frameworks interact with the browser environment and with each other. Much of this has to do with providing a common point of registration in the browser environment, so that namespace conflicts can be resolved or at least made visible. OpenAjax also has several standards for event management and are working on common widgets specifications. Naturally, Dojo has support for this and follows the developments closely.
Analytics	The `dojox.analytics` package tracks mouseovers, clicks, and related information and sends it back to a predefined URL at regular intervals. This can be used to analyze how your application is really used and how to improve the user interface.
Gfx	`dojox.gfx` is a very comprehensible 2D graphics package. It offers such functionality as beziers, transformations, gradients, line styles, polygons, fills, and text plotting.

Feature	Comment
Charting	`dojox.charting` continues to build upon the `dojox.gfx` foundation and provides a staggering amount of riches for creating static and dynamic charts of various types. Among the kind of charts that can be created, we find lines, pies, scatter, and areas. Also, most charts can combine several series into the same chart for comparisons. There is also quite a lot of magic for managing x and y-axis plotting, color schemes, and line types. If you start digging in this package, I guarantee lots of fun and many an afternoon flashing by in minutes. Also, don't be afraid to ask specific questions in the forums, since the documentation tends to be a wee bit scarce.
Gfx3d	Not being satisfied with mere 2D graphics, Dojo also sports a capable 3D graphics package, with everything from quads, cameras, matrix operations, and scenes. At the moment, it can't really be considered blindingly fast, but as the Tamarin JIT engine gets included into Firefox 3 (and its descendants), it will start to become a real alternative to Flash-based 3D charts and effects.
Encoding	The `dojox.encoding` package provides a lot of hard-to-recreate functionality like compressions algorithms (lzw, splay), encryption algorithms (Blowfish and MD5 digests), and general encoding (base64, etc.).
XML Parsing	When you need this, you need it bad. If you need to parse or recurse down an XML structure provided by the server, you can just pour the text representation into the XML Parser, which will give you a JavaScript JSON object in return.
RPC	An extension of the basic XHR Ajax features in Dojo, the RPC support, makes it easy to create functions that wrap server calls and hides complex calls and callbacks. Using the Simple Method Description format (SMD), a RPC function can be created on the fly using a server-provided file.
Edge-cached by AOL and Google	AOL and Google have agreed to host released versions of Dojo on their CDN (Content Delivery network), to be used by anyone. That means you can create rich Dojo applications inside a single web page, use the pages locally, mail them to someone, or put them on a server, without having to ship and pack up the files of the framework itself.

Dojo compared to other Ajax frameworks

Dojo is by far the largest Ajax framework today, in terms of features and functionality. But that is not always a deciding factor in which framework to choose. A smaller, focused framework might be a good choice for certain occasions. Below is a list of other Ajax frameworks or toolkits and how they stack up against Dojo.

Prototype:

Prototype is a fairly small framework, which historically has been fairly easy to use, and is used very much together with Ruby on Rails. It does not play nice with the common namespace, and collisions with other frameworks can occur. It has its own OO framework with classes and inheritance but lacks support for widgets and anything that is not related to Ajax or DOM interaction.

jQuery:

JQuery is a very fast framework which is excellent for modifying and adding features to an existing page. This is due to its Xpath/CSS query language, which makes it easy to operate on several elements at once. It does not contain any widget system, but has a fairly capable UI branch with a large number of plugins, from treetables to slideshows. Since version 0.9, Dojo also has XPATH query functionality in the form of dojo.query.

YUI:

Yahoo User Interface is controlled by Yahoo, but still open source. It is one of the most polished toolkits out there with a strong focus on layout and user interface widgets. YUI also has basic element animations and a fairly good data source abstraction, in the same spirit as dojo.data. It is also publicly cached on Yahoo's CDN.

EXT:

Ext has many similarities with YUI. It is very polished, maybe the most good-looking Ajax framework, even if Dojo is not far behind. Like YUI and Dojo, it has a powerful data abstraction, which separates presentational widgets from the source of the data, making it easy to change presentation method.

Server-side compatibility

Dojo is, of course, compatible with any server technology you can imagine. In the "Hello World" example in the online "Dojo Book", example servers to use to handle the Ajax transfers are shown in PHP, ASP, ColdFusion, JSP, and Perl.

Since Ajax in itself separates the technology used on the client (The Browser) and the server, it allows removal of some of the complexity of the server-side framework. Using Dojo, you can put the entire View in the traditional MVC sense on the client, and get a dramatically slimmed-down server as a result.

Having said that, Dojo has, of course, preferences in what kind of data it likes, especially for feeding its data sources, but as you will find out later, it is not all that difficult to write your own data source if your server talks in funny ways.

Reference sites, API listings, forums, and mailing list

The starting point of any exploration of Dojo is the main site at `http://dojotoolkit.org/`. The next important link to remember is the forums, where questions are answered in a furious pace, day and night. It is located at `http://dojotoolkit.org/forum`.

Fairly new and with a plethora of screencasts and tutorials is Dojo Campus at `http://dojocampus.org/`.

And the two tabs on your browser that you will use the most will probably point to the auto-generated API documentation at `http://dojotoolkit.org/docs/api`, and the online Dojo Book at `http://dojotoolkit.org/book/dojo-book-1-0`.

The main mailing list can be found at `http://dojotoolkit.org/mailman/listinfo/dojo-interest`, but does not have as much traffic as the forums.

Summary

- Dojo is an Ajax toolkit written in JavaScript providing a huge amount of functionalities.
- Dojo is open-source and has a large, welcoming community.
- Dojo is easy to get started with.

2
Useful JavaScript and Dojo Tricks

The history of JavaScript is a bit convoluted. It was first released as LiveScript by Netscape, but was later renamed to JavaScript in the December 1995 release of the Netscape Navigator 2.0B3 browser, which also had support for Java applets. This is thought to have been a deciding factor for the name change.

Unfortunately, JavaScript and Java have very little in common, and the name has led to quite a lot of confusion over the years. JavaScript 1.1 was standardized by the European Computer Manufacturers Association (ECMA) (see `http://www.ecma-international.org/memento/index.html`).

Microsoft has created its own variant of JavaScript called JScript. JavaScript 1.5 and JScript 5.5 are conformant with ECMAScript version 3.

For more information about the history of JavaScript, see the wiki page at `http://en.wikipedia.org/wiki/JavaScript`.

It was actually used on the server in one of its earliest incantations by Netscape, but server-side JavaScript has been violently out of fashion until recently, where a couple of very high-profile server-side JavaScript projects have garnered a lot of attention.

Some good examples of server-side JavaScript frameworks and services are: Steve Yegge's Rhino on Rails at `http://google-code-updates.blogspot.com/2008/01/interview-with-steve-yegge-on-rhino-on.html`, Sun's Phobos at `https://phobos.dev.java.net/tutorials/usedojo/`, and Aptana's Jaxer at `http://www.aptana.com/jaxer`, 10gen JavaScript cloud at `http://10gen.com`, and AppJet at `http://www.appjet.com`.

At the time of writing this book, Dojo was being used tentatively in several server-side JavaScript projects. Being the most advanced framework (in terms of architectural features, not necessarily related to visual widgets), it is a fair bet that the use of Dojo on the server side will increase.

It is even possible that you're reading this book right now because you're involved in a project using Dojo on the server side exclusively.

But the focus right now is on the browser. JavaScript is, by history, tied to the Document Object Model or DOM that defines the web page that is loaded in the browser. Mostly, if not all elements and their properties that can be defined in HTML and CSS can be created just as well inside a snippet of JavaScript code.

Most probably you have written some JavaScript prior to reading this book. It is very easy to write JavaScript in a very simple-minded manner, where you use few if any of its time-saving features and stay within the beaten path. With some luck, you'll have some more tools to work with when working on your next project.

The following sections assume that you are familiar with some kind of C-like programing language. JavaScript's C-like syntax is a two-sided coin. Since it looks similar to many other languages, it is easy to pick up and use, but that illusory familiarity can also be a hinderance to learn the more advance concepts of JavaScript.

Here, we do not give you a complete run-down of all possible functions in JavaScript or a total list of properties for all elements. The following are just the most common time-savers in JavaScript (and later, in Dojo itself) that can make your daily coding simpler. For a good introduction and good reference to JavaScript as a language, go to `http://www.w3schools.com/js/js_intro.asp`.

In the examples encountered in this (and other chapter(s)), I make frequent use of `console.log()`. This object is not part of JavaScript per se, but is a kind of a de facto object which both the Venkman and Firebug JavaScript debuggers make good use of. So to try out the examples, I suggest that you install either one of the debuggers and create a page with just one large <script></script> tag, where you can write the examples and then watch the output on the console of the debugger.

If you have none of these installed, they can be donwloaded at the following places:

Firebug: `http://getfirebug.com/`.

Venkman: `http://www.mozilla.org/projects/venkman/`.

Most things that you'll encounter in JavaScript are objects. If you see something and aren't really sure what it is – it is most probably an object. Arrays are objects, and even functions are objects. There are several ways to create an object. One way to create an object is the following:

```
var myobj = new Object();
```

But you could just as well use the object literal notation, which is much more compact:

```
var myobj = {};
```

Just as variables of any kind are created when and where they are defined, object properties can be smacked on whenever you need to. To declare a box object with the four properties, viz. top, left, width, and height, you could write the following:

```
var box = {};
box.top = 17;
box.left = 42;
box.width = 100;
box.height = 100;
```

But the object literal notation can be even more compact. The above could (and often is) be written like this, in one go:

```
var box = {top: 17, left: 42, width: 100, height: 100};
```

In another language, this object declaration would have necessitated a class declaration, possibly an interface to go with it, and a lot of boilerplate code. In JavaScript, you can simply define an arbitrarily complicated object in one or two lines as a return value to a function.

There is one danger with object literal notation, which you must keep in mind since it is used almost everywhere when using Dojo. **You must never leave a dangling comma when using the object literal notation!**

This is easy to do, especially when prototyping where you try out a lot of things at different times, by commenting out certain parts of a definition. Consider the code below, which defines the arguments to the programmatic creation of a Grid. You will learn this later in the book, but the common way of creating Dojo widgets (Dijits) is to have a property object as the first argument, with a possible second DOM node argument which defines where to put the widget:

```
var gridnode = dojo.byId(this.id+"_grid");
var grid = new dojox.Grid(
{
   structure:layout,
   model:model
}, gridnode);
```

If you, for some reason would like to omit the last property (`model`), you might want to comment out the last line, so that the call to create a new grid component will now look like this:

```
var grid = new dojox.Grid(
{
  structure: layout, <---------------------- Here be danger
   //model: model
}, gridnode);
```

A possibly better notation, which lets you spot these kinds of errors easier or hopefully, avoid them altogether, is to insert the comma at the beginning of each line instead.

```
var args =
{
   foo: 17
   ,bar: 4711
   ,baz: "Is that a function in you pocket or are you just happy to
see me?".
}
```

When referring to object properties, you can use either of two methods: dot notation or bracket notation, except in a special case, where you must use bracket notation. If we need to access the `top` property of the object `box` in the above example, we could either write the following:

```
var thetop = box.top;
```

or

```
var thetop = box['top'];
```

The special case here is when you have a property which contains a space, like `foo bar`, which must be quoted.The dynamic insertion of properties on objects makes it possible to use any object as a key to an array.That was exactly what happened in the example where we defined the box object. Each property could be accessed in bracket notation, even when setting the value:

```
box['top'] = 17;
```

Since the dynamic addition of properties on object is not limited to objects you create yourself, you can use this feature for debugging purposes, when accessing DOM elements.

What is meant by that is that you can easily 'insert' your own data to existing objects you are using, regardless of whether you have created them yourself, as you will clearly see below in just a couple of lines.

Often you create elements dynamically and you only provide an element that is used by another function (maybe inside Dojo) in some way and would like to be sure that it really is _your_ element. To add a custom property to any object, do the following:

```
var myelement = document.createElement('div');
myelement.foobar = 17;
```

You will now be able to track 'your' element in Firebug and in your own debugging more easily. Checking for the existence of the `foobar` property.

A related issue is where to store metadata for your own functions which are closely tied to the DOM element which you are using. The first idea that comes to mind might be to create a hashtable object indexed on the DOM id of the element, like this:

```
var mytable = {};
function myfunc(element)
{
    var myprop = 17;
    mytable[element.id] = myprop;
}
```

And then when calling another function with the element, you need to get your metadata:

```
function getMyMetadata(element)
{
    var metadata = mytable[element.id];
    ...
}
```

Instead of using the hashtable, you could instead just tuck the metadata on the DOM element so that whenever you use the element, you have your metadata as a property. The above two functions might be re-written in the following way:

```
function myfunc(element)
{
    element.myprop = 17;
}
function getMyMetadata(element)
{
    var metadata = element.myprop;
}
```

A good thing to remember is that an object, of course, can contain other objects, so that the following notation is fairly common when building arguments for use in Dojo widgets:

```
var draw=
{
   stroke:
   {
      color: "blue", width: 2
   },
   fill: "lightblue"
}
```

Array usage tips and gotchas

When we speak of arrays, we normally think of nice, numerically indexed arrays. JavaScript has those, with a standard (and some non-standard) repertoire of functions that goes with the territory. A normal array definition goes like this:

```
var foo = ['foo', 'bar', 'baz, 'frotz, 'quatz'];
```

Where `foo`[0] == `'foo'`.

You can of course use a common loop to access all the elements in a classical fashion, like this:

```
for(var i = 0; i < foo.length; i++)
{
   console.log(foo[i]);
}
```

What you might not know is that you can treat an array as a stack, at no extra cost, with the functions `push()` and `pop()` respectively. A destructive walk of the array (which removes one item at a time from the end) would look like this:

```
while(foo.length > 0)
{
   console.log(foo.pop());
}
```



A stack is a data structure which stores data as a stack of plates. When you 'push' data it gets put on top of the stack, when you 'pop' you take away the data formerly put on top of the stack.

The `push()` function also operates at the end of an array, but instead of removing an object, it appends an object instead. Push and Pop are functions which I heartily recommend and which are easy to overlook.

So if you have an array `x = [1, 2, 3, 4, 5]` and then execute a statement like this:

```
x.push(6);
```

The array will now look like this: `[1, 2, 3, 4, 5, 6]`.

Whereas `pop()` will remove and return the last element. This is nothing revolutionary in itself. What I'd like you to remember, though, is that the powerful idea here is that you can create an array in a classical fashion, and at certain occasions use it as a stack.

So, if you again take the array `x = [1, 2, 3, 4, 5]` and then execute the following statement:

```
var y = x.shift();
```

The variable `y` will now contain the number 1 and the array will have been modified to display `[2, 3, 4, 5]`. This means that by using `push()` and `shift()` you'll have a queue in no time, which still behaves as an array when requested. Remembering this will save you a lot of unnecessary extra variables and/or loop unwinds in the future, if you haven't used it yet.

Functions as arguments

As functions are first-class objects in JavaScript, a common trick is to use functions as arguments to other functions. You will see quite a lot of this when working with Dojo, hopefully doing it yourself soon enough. The common use of this functionality is to define a function, which is then referred to by name:

```
<script>
function handleClick(e)
{
    console.log("Element which generated click;"+e.target.id);
}
dojo.connect(dojo.byId('mybutton'), 'onclick', handleClick);
</script>
<button id="mybutton">Foo</button>
```

Here, we used the dojo function `dojo.byId()`, which is a shorter way of getting a handle for an element, instead of writing `document.getElementById()`.

As you can see, the `dojo.connect` function, in its most simplest form, causes the function passed as the third argument to be called when the event occurs.

Sometimes you'll feel silly for creating a whole function for something that you'd really like to just do right there and then, which is where the embedded function declaration comes in handy. The above even code would then look like this:

```
dojo.connect(dojo.byId('mybutton'), 'onclick', function(e)
{
   console.log("Element which generated click;"+e.target.id);
});
```

The tricky part here is to remember to close the last parenthesis, since we're actually in the middle of a function call, even though it looks like a definition on a glance.

You are also free to collect function references in arrays or hashtables. Since functions are first-class objects in JavaScript, they can be used as any other variables.

Suppose that you write a function where you want to use a callback. That function might look something like this:

```
function foobar(a, b, userFunction)
{
   ....
   var x = doSomething(a, b);
   userFunction(x);
}
```

As you can see, the use of a callback function is very simple and immediate.

As a function is an object in JavaScript and object can contain other objects, this can be (and is!) used to arrange functions in 'namespace' hierarchies.

Imagine that you're introducing a new functionality in a web page (without the use of Dojo), a shopping cart, for instance.

The page in question has lots of hairy legacy JavaScript functions and objects for a variety of nefarious functionalities, much of which have very generic names.

What you don't want in a situation like this is to inadvertently use the same name for some of your stuff that will collide in subtle but dangerous ways with all the other things on the page.

What you can do then is to create just one object and tuck the rest of your code underneath it, like this:

```
var scart = {};
scart.foo = 17;
scart.bar = 4711;
scart.init = function()
{
}
```

This is only useful if you have just one shopping cart on the page, which might actually be the case. However, you might want to create a component that you can put on several places on the page, like a ticker, chart, or data grid. That means that we'll have to dive into JavaScript's prototyped object model.

JavaScript's prototype-based OO

This is not particularly complicated. It's probably the most down-to-earth object oriented model you've heard of, and it mainly involves searching about for stuff.

In many object-oriented languages, you have the concept of classes and instances. The first is a generic definition of an object and the latter is an instance according to the definition. In JavaScript, we only have instances.

Some instances can act as blueprints for other instances. The object-model of Java is easier to grok, since they separate concerns a bit better, but I hope that JavaScript's ease and elegance will make it possible to understand its slightly hairier object orientation.

To create something akin to the class definitions, you can use a special property that JavaScript keeps an eye out for, an object–prototype. The prototype property is an object where you can define all variables and functions that a new instance of that object shall have.

```
var chart = function(name)
{
   this.name = name || this.defaultname;
}
chart.prototype.defaultname = "Mario Chart";
chart.prototype.draw = function()
{
   ...
}
var mychart = new chart("Earnings 2008");
var yourchart = new chart();
```

The first thing that happens here is that the function chart is defined. Inside it we use the 'this' reference to set a variable. The variable `this` refers to the current object. As we're going to see further on, it is not always so easy to know what the current object is when using a lot of callbacks. But for now, you can safely assume that this does what it looks like it does.

Then we define two properties for the `prototype` property of the function. These are used when a new instance of the object is created using the `new` operator. Strictly speaking, what happens is that the new instance gets a reference to the original object's (function's) prototype property.

Two different chart objects get created, and for each one, the two properties `defaultname` and `draw` are readily accessible. But something else happened here, didn't it? The first instantiation gave an argument, whereas the other did not.

The constructor function of the `chart` object does take an argument, but what JavaScript does when a defined argument is not provided in a function call is not throw a fit (exception) but silently set that argument to `undefined`.

The next thing that happens in the function is that we set the `name` property for the current object. If none is defined before, it becomes defined now. That's how it works in JavaScript.

Also, the `||` operator tests whether the argument on the left side is defined or not. If it is, that value is returned, otherwise the value on the right hand is returned. This makes it handy in situations like this, where you want to fall back on a default value if a parameter or argument is not defined.

This, in turn, means that the property `mychart.name` will be `Earnings 2008`, but the property `yourchart.name` will be `Mario Chart`.

The funny thing with prototypes, though, is that they're not blueprints but references. This means that you can easily change the value for a property across several objects at once.

```
 var obj = function(){};
obj.prototype.x = 0;
obj.prototype.setX = function(value)
{
   x = value;// Bug
}
 var a = new obj();
a.setX(1);
      var b = new obj();
b.setX(2);
      var c = new obj();
    c.setX(3);
      obj.prototype.x = 17;
      console.log(a.x);
```

The output of the logging will be **17**, as it will for all three objects. When we create a new object using the `new` operator, the new object refers to the original function object's prototypes.

So, when we access a variable inside the context of the new object and that variable is not defined, JavaScript searches up the prototype chain until it finds a variable with that name, or if not, it creates it for the local object.

If we had written the setter function like this instead:

```
this.x = value;
```

The console log would have reported **1**, since the variable name search that JavaScript does would hit a match immediately on the current object, so it would not continue to search up to the next prototype.

Using closures

A closure in JavaScript isn't something that you can create or use directly, like an array. One way of describing closures is that they are the scope of functions that are retained even after the function has returned.

```
function foo()
{
   var x = 17;
}
foo();
console.log("x inside foo = "+foo.x);
```

You might think that the result of the output would be an error, or **undefined**, but it will actually display **17**, due to the power of closures.

Some ground rules for using closures:

- Each time a function is called, a closure is created.
- All variables declared during that function call are added to the closure.
- All variables inside the closure can still be accessed after the function has returned.
- 'inner' functions that are called within one closure inherit the references from the outside closure.

This can be demonstrated using a loop inside a function:

```
function makeButtons(n)
{
        for(var i = 0; i < n; i++)
        {
        var b = document.createElement('button');
      b.innerHTML = i;
      var foo = document.getElementById("foo");
      foo.appendChild(b);
      b.onclick = function()
        {
        alert("Button "+b.innerHTML+" was clicked");
        }
        }
}
```

The function assumes that there's a div somewhere on the page where it can add the buttons it creates. What happens when then function is called with the argument 4, is the four buttons will be created (obviously) in the page, named 0 through 3. When you click one of them, though, you will always get the alert **Button 3 was clicked**.

In a language without closures (possibly still Java), similar code would have resulted in an exception or error of some kind, when the `onclick` function tried to refer to the variable `b`, which had gone out of scope a long time ago at the time the button was clicked.

But as JavaScript has closures, this works remarkably well. The anonymous inner function declared as a handle for the `onclick` function will, when called, have access to all variable references from the outside closure, including the variable `b`, even though the `makeButtons` function has returned happily tens of seconds before the button itself is clicked.

But the message doesn't say **Button 0 was clicked** when you click the first button, and the reason for that is not that the closures aren't working properly, but that they can't work miracles.

The anonymous function does indeed succeed in getting hold of the variable `b`, but `b` gets overwritten three times during the loop, so the content of `b` when the button is clicked is of course the last button created.

To make the following code work, we need to create a closure to protect the current value of `b` for each loop. Utilizing the facts that a closure is created when a function is called, and that we can create anonymous functions in JavaScript, a solution to the problem would be:

```
(function()
{
   b.onclick = function()
 {
  alert("Button "+b.innerHTML+" was clicked");
  }
})();
```

A new anonymous function gets declared for each loop. It inherits all declared variables from the outer closure (function call). Since we want it to be called immediately, we have to wrap it in parantheses, and then call (using `()`) the paranthetical function.

This style of coding is also called 'self-invoking functions'. Since each closure is created for a specific loop, the `b` variable will retain the value it had when the closure was created, and the alert will correctly say which button was clicked.

OO in Dojo

Dojo has built its own inheritance-based system of object-orientation, on top of the prototype-based object orientation inherent in JavaScript. The system is used mainly for inheritance, to make it easy to write a base class for Buttons, and extend that to specific types of Buttons later on.

It also makes it simpler to convert code written in Java or other classically object-oriented languages. In contrast to many other object-oriented approaches, Dojo allows for multiple inheritance through *mixins*, which lets you copy the behavior from any number of base classes.

To declare a class using Dojo, you use the `dojo.declare` function:

```
dojo.declare ("MyNewClass", superclass,
{
});
```

Note that everything above is just a one-line function call which is broken down to several lines for pretty printing purposes. That's why theres a trailing left parenthesis at the end of the closing bracket.

It is very easy to forget to close the parenthesis, especially when you're writing your first classes and have them long enough that you can't see the beginning and end at the same time. You've been warned.

The `dojo.declare` function takes three arguments:

- A String, which is the name of the new class to be created.
- An object, which is the reference to the existing superclass. **This cannot be a String!**
- An object literal, which defines the body of the class in name-value pairs separated by commas.

The object literal has some tricks up its sleeve as well. There's a magic function to be declared called "constructor", which takes any arguments to the newly created instance and manages initialization and anything else that needs to be done at object creation.

Also, all variables that need to be available in class function using `this` references should be explicitly declared inside the object literal, given default values, and so on.

Let's say that we create a new class from scratch with no superclass, having one variable and just a constructor. It would look like this:

```
dojo.declare ("MyNewClass", null,
{
   constructor: function(foo)
   {
      this.foo = foo;
   }
});
```

When creating a new object of the type `MyNewClass`, you use the normal JavaScript new keyword, like this:

```
var a = new MyNewClass(4711);
```

Where `a.foo == 4711`.

If you want to have static variables that exist only in the class and share the same state across all instances, you can define them inside a block that is class-addressable. You can use any name or names you want, but 'static' might not be a bad choice.

```
dojo.declare ("MyOtherClass", null,
{
   constructor: function(foo)
   {
      this.foo = foo;
```

```
        this.static.objects++;
        console.log("We have now created "+this.static.object+"
MyOtherClass instances");
    },
    static:
    {
        objects: 0
    }
});
```

The `static` property can be accessed (as seen above) using `this.static.` inside instance functions, and from the outside using `MyOtherClass.static`. Pretty simple in all, really.

But to return to subclassing, Dojo has chosen some very sensible basic defaults when it comes to overloading and the like. Let's say we have two classes, where one extends the other, like this:

```
dojo.declare("Ship", null,
{
    name: "-none-", // default value
    displacement: 0, // Not needed, since it will be created by the
    //constructor. Declared here as well for explicitness.
    armament: {},
    constructor: function(n, w)
    {
        this.name = n ? n : this.name;
        this.displacement = w;
        this.armament.turrets = 4;
    },
    scanSurroundings: function()
    {
        // Scan visual surroundings
    }
});
dojo.declare("Submarine", Ship,
{
    constructor: function(n, w)
    {
        this.armament.torpedoes = 16;
    },
    scanSurroundings: function()
    {
        this.inherited(arguments);
        // Scan sonar afterwards
    }
});
```

The first class, `Ship`, defines some basic things that all ships have in common like turrets, and a function to scan the surroundings. The second class defines a submarine, which wants to reuse the functionality in the `Ship` class, but add some things and modify others.

Note that there is no reference to the superclass constructor inside the constructor of the `submarine` class. The reason for that is that the superclass' constructor **is always called** for a subclass, and it is also always called **before the constructor of the subclass is executed**.

However, this only applies to the constructor. So, if you override a superclass function in a subclass (that Submarine does) and still want to call the superclass' function before you extend its functionality, you'll have to use `this.inherited`.

The `arguments` variable is always created by the Javascript runtime whenever a function is called. It is an object which contains all variables passed to the function. It has nothing to do with Dojo.

If you want your class to inherit variables and functions from more than one class, you simply use an array of class references as the second argument to `dojo.declare`. All classes in the array get applied to your new class.

As an example, consider that you have already defined the classes `Mammal` and `FlyingCreature`, you might declare a `Bat` class like this:

```
dojo.declare("Bat", [Mammal, FlyingCreature],
{
   // Class body
});
```

We'll be covering Widgets and how to manipulate them later, but right now you can see the that it does indeed use an array as the second argument to `dojo.declare`.

Anatomy of a Dojo page

Dojo has one peculiarity compared to most other Ajax frameworks: it needs its entire src/ catalog. That means that you can't (like with many other popular Ajax toolkits or frameworks) just copy the base file, `dojo.js`, to your web directory, include it, and call it a day – you must copy the whole distribution, which is several megabytes.

Dojo consists of a base package, which can be zipped down to just over 25K, and contains a lot of nice features, like the CSS query engine, event management, and so on.

The rest of Dojo's functionality—the Dijit widget system, the visionary and cutting-edge dojox stuff, and everything else, is loaded as required either by the web page or by the components themselves.

When you download a pack of Dojo, you will see a directory structure looking a bit like this:

```
dojo
dijit
dojox
util
```

The `util` directory doesn't contain anything that you'll access from your web page, but has some nifty things. It has build scripts (using Java) that can build customized files for your site and shrinksafe that can be used to compress Dojo or other JavaScript files.

The main file, which must be loaded before anything else can take place is `dojo.js`, which is inside the `dojo` directory.

A minimal web page using Dojo might look like this:

```
<html>
   <head>
      <title>Minimal Dojo Page</title>
      <style>
      @import "dojo/resources/dojo.css";
      @import "dijit/themes/tundra/tundra.css";
      </style>
      <script type="text/javascript">
         djConfig=
         {
            parseOnLoad: true
         }
      </script>
      <script type="text/javascript" src="dojo/dojo.js"></script>
      <script type="text/javascript">
         dojo.require("dojo.parser");
         dojo.require("dijit.form.DateTextBox");
      </script>
   </head>
   <body class="tundra">
      <input type="text" name="date1" value="2008-12-30"
        dojoType="dijit.form.DateTextBox"/>
   </body>
</html>
```

The first thing you might notice in the above code is the import of two custom stylesheets that Dojo use. The first, `dojo.css`, is a small and useful stylesheet which is derived from Yahoo's reset CSS. The other import is one of Dojo's themes, called Tundra.

If you skip forward quickly to the body tag, you see that it has got the class `tundra`, referring to the theme included above. Many of Dojo's widgets require a theme to be loaded, and also, more importantly, require the theme class to be set *on the body tag*.

This might raise issues when using Dojo in parts of a website where you don't have or want to have control of the styling of the body tag.

The reason that you can't use the theme on a div further down the hierarchy is that Dojo frequently creates decorative elements for widgets directly under the body, rather than near or around the element where the widget is created. This is done in order to avoid inference.

The next part of the page is the small script that defines a variable called `djConfig`. As the name suggests, it contains any configuration options you might want to add before loading Dojo.

The most common is the one we see here: `parseOnLoad: true`. This configuration option causes Dojo to scan or parse the whole page after it is properly loaded, in search for Dojo components which then are created on the spot.

In the next script statement, Dojo itself is loaded, taking care of the options in the `djConfig` variable (if present).

Concluding our script sections, we have two `dojo.require` statements. Hopefully I have managed to explain to you that this is the way you get Dojo to load only the stuff you need for your page.

The first line loads the Dojo parser, which parses the page, looking for 'dojoType' tags in all elements.

This is needed because we just configured Dojo to parse the page when loaded. The second line loads the definition for a widget that gets used in the page; a DateTextBox which displays a nice-looking calendar for entering dates.



In the HTML markup, there's only one element defined – a div. Here, you see Dojo's elegant use of tag properties to declaratively create widgets and other components.

Don't worry, though. Everything that you see declared this way can also be created programmatically inside a script, with the same property names and the same argument formats. This will be covered in detail later.

The main property used is `dojoType`, which tells the parser which component should be created and replaces the element where it is defined. You might think that this would be enough, and that the required statement in the beginning of the page is somewhat superfluous. However, the required statements correspond to file names in the Dojo file hierarchy, but a file might sometimes define more than one component. It's therefore necessary to be explicit with which files to load.

After the `dojoType` definition comes zero or more arguments to the component being created. Let's take a look again at the specific line:

```
<input type="text"     name="date1" value="2008-12-30"
dojoType="dijit.form.DateTextBox"/>
```

The order of the properties doesn't matter, of course, as long as they're there. This specific widget takes a couple of arguments, but neither name nor value are arguments that it uses. Rather, the name is the standard DOM element name and the value is the standard value for the input element.

However, the DateTextBox does take a parameter named `strict`, which if set to true makes it reject entries with extra whitespace or abbreviations for days or months. If we wanted to set that, it would be used just as another property; `strict = "true"`.

But the parser is even more useful. Let's say that you've written a class that is not a Dojo widget but still needs to access some part of the DOM, or that you are converting some other JavaScript code into a Dojo class which has the same need, but you still don't want to make it a full Dijit.

This would in itself be kind of odd. I would really recommend to create your own widget when doing anything which will show up on the page as you get so much for free, but just for the sake of the argument, let's say that you don't.

You can use `dojoType` to create an instance of that class as well, and all properties defined inside the tag will be passed as instantiated variables to your class.

To be more precise, the parser collects all the properties in the markup and populates an object with it. It then assumes that your constructor looks like this:

```
constructor: function(args, node)
{
}
```



In the above code example, `args` is the passed object where all the properties can be accessed like in any object, and `node` is a reference to the HTML DOM node that had the `dojoType=xxx` definition.

To wrap up with an example, you could have a page like this:

```
<html>
   <head>
      <title>Simple Dojo Parser Example</title>
      <script type="text/javascript">
          djConfig=
         {
            parseOnLoad: true
         }
      </script>
      <script type="text/javascript" src="dojo-1.1b1/dojo/dojo.js">
      </script>
      <script type="text/javascript">
         dojo.require("dojo.parser");
      </script>
      <script>
         dojo.declare("fooclass", null,
         {
            foo: 17,
            constructor: function(args, node)
            {
               this.foo = args.foo;
               console.log("Got instantiated inside node "+node.id+",
                           args is..");
               console.dir(args);
            }
         });
      </script>
   </head>
   <body>
      <div id="mynode" dojoType="fooclass"  foo="4711" bar="baz">
      </div>
   </body>
</html>
```

The important part here is that the class is defined in the head, so that it is found by the parser, when the body of the page is parsed. If you have firebug installed and the firebug console up, you will see output that looks something like this:



Note here that even though we use the bar property in the `div` element, it does not get passed to the constructor. You have to explicitly define what properties your class shall accept from the parser. In our case, this is only the variable `foo`.

Basic Dojo functions

Dojo has an abundance of functionalities, of which only a larger subset will be covered in this book. I have strived to show funcitonality that is most relevant to everyday usage, rather than a full catalog.

The function `dojo.byId(arg)` returns the HTML DOM element with the argument id. If you have an element `<div id='foo'></div>`, it could be accessed using `dojo.byId("foo")`.

The function `dojo.isArray(arg)` returns true if the argument is an array. Together with `dojo.isObject(arg)`, it's very useful in separating arrays from objects. The reason this is useful is that arrays are objects and you have to jump through hoops to be otherwise certain that you actually have an array.

The function `dojo.trim(arg)` trims whitespace characters from both sides of the passed argument string. You need this. Really.

The function `dojo.addOnload(arg)` adds a function to be executed when the page has finished loading and all elements defined in the HTML page have been created as DOM elements. You can add any number of functions using this function, which will be executed in sequence. It is not guaranteed that CSS and pictures have finished loading, in contrast to Widgets, which will have all CSS and pictures in their markup templates loaded before their constructors are called.

The function `dojo.forEach(arg, func)` is used to loop over each element in an array. Each item is passed to the function as its single argument. It is used like this:

```
dojo.forEach(myArray, function(e)
{
   // do something to the item in the e variable
});
```

To get a complete view of all the packages and functions that make up Dojo, go to the API documentation page at `http://api.dojotoolkit.org/`.

Intermediate Dojo functions

The following functions are related in:

The function `dojo.filter` takes two arguments: an array and a filtering function and returns an array. Suppose that you want to do the following:

```
function makeNewArray(arg_arr, ceiling)
{
   var new_arr = [];
   for(var i = 0; i < arg_arr.length; i++)
   {
      var element = arg_arr[i];
      if (element > ceiling)
      {
         new_arr.push(element);
       }
   }
   return new_arr;
}
```

The same code can be expressed like this using the filter function:

```
function makeNewArray(arg_arr, ceiling)
{
    return dojo.filter(arg_arr, function(x){ return x > ceiling; })
}
```

What happened now? Most importantly, all intermediate index variables and temporary variables evaporated, reducing the risk for errors drastically.

In some more detail, `dojo.filter` applies the function argument on every element of the array. If the function returns `true`, `dojo.filter` will add the element to an invisible return value array. If it returns numerical zero it does not.

It will also coerce other values to boolean values, where a numerial 0 will be treated as false, and any other number as true. If a string is returned, " " will be treated as false, and a non-empty string as true.

When all elements in the array have been processed, the invisible argument array is returned.

Also, the code is much shorter this way.

If you want to get the array of 1s and 0s from the comparative function directly, you can use `dojo.map` instead. It takes the same arguments otherwise.

The function `dojo.map` works like this: It takes an array as the first argument and a function as the second, then calls the function repeatedly, once for every item in the array. It then collects all the return values in a new array, which is then returned.

```
var new arr = dojo.map([1, 2, 3], function(x){ return x * 10; })
```

The variable `arr` will now contain [10, 20, 30].

If you need to check the elements of an array for certain conditions, use `dojo.every` and `dojo.some`.

You might want to check if an array of values contain any large deviations, say over 1000 or so. Then you can use `dojo.every` like this:

```
if(dojo.every(arg_arr, function(x) { return x < 1000)})
{
   // everything is OK, no value was over 1000
}
```

The function `dojo.some` does the same thing but in reverse. This means that if any of the repeated calls to the function argument returns `true`, `dojo.some` returns `true` as well.

Logging, Firebug integration, and debugging

As previously stated, you can turn on debugging inside the `djConfig` variable, like this:

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

What does this mean? And also, does this mean that we can't debug at all unless `isDebug` is set to true? The latter is, of course, not true. The debug flag is there to give the developer (you, that is) a live logging window, so that you can get information out from your running code. This has historically been crucial for browsers where the vendors have made **true** JavaScript debugging very hard, if not impossible, to implement.

What `isDebug` does is to make sure that you get a `<div>` somewhere on the screen that adds your debug output, line by line. It also gives you some additional features as logging levels and dumping of objects.

If you have Firebug installed in your browser, Dojo will detect and use the Firebug console instead of `<div>`. This also means that you will have to open the Firebug debug console manually when you're using Firefox + Firebug. But you're already doing that anyway in that case, aren't you?

The different logging levels available are:

```
console.log      - normal text
console.debug    - normal text
console.info     - blue info icon and normal text
console.warn     - yellow warning icon and black text on blue
                   background
console.error    - red error icon and red text on yellow background
```

One of the most powerful and useful features of debugging in Dojo (or, actually even without Dojo with Firebug) is the `console.dir()` call.

The function `console.dir` analyzes the passed argument and tries to dump its content in the debug console. If you have an array called `my_arr` and use the following statements:

```
console.log("Updating chart with new data");
console.dir(my_arr);
```

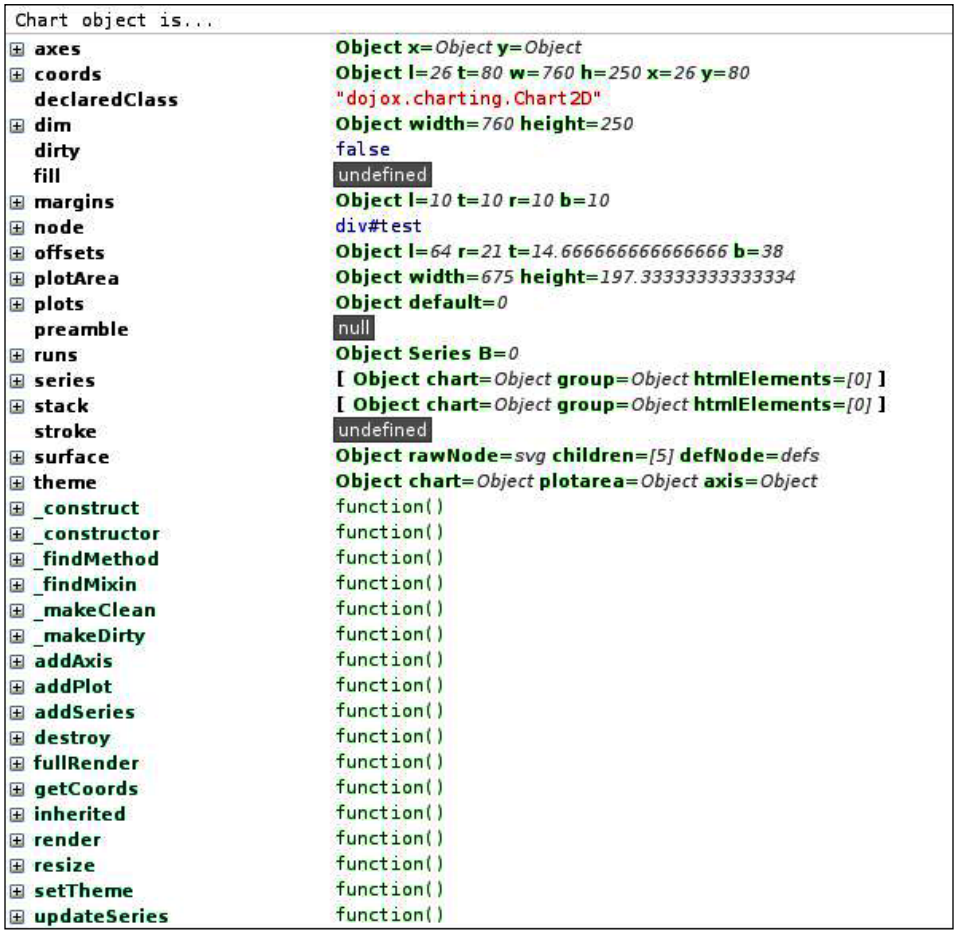
You might see the following output:



For the next example, let's say that we want to use a fairly complicated object that we don't have documentation for. One good start could be to dump it in the console and get an overview of which properties and other objects it contains.

```
console.log("Chart object is...");
console.dir(chart);
```

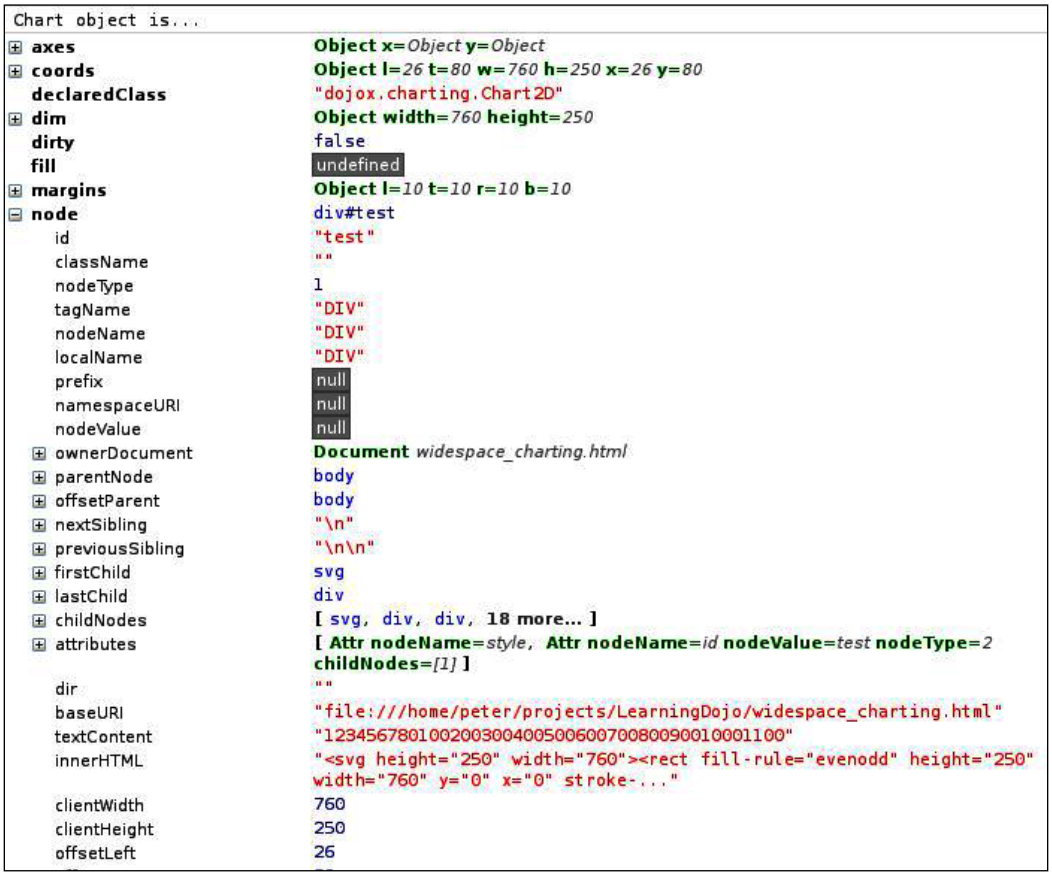
This will give the following output (We're dumping the chart object we created in Chapter 1):



You see that we get quite a good list of properties. The black properties listed first are scalars (Strings, numbers, and so on) or objects. The second part of the dump lists all functions available in the object. The functions are not normally dumped in their entirety, but it gives a good overview nonetheless.

The right side of the dump gives a quick look inside any property. If it is a number, string, or an array, it is listed completely (unless the array is very long). Even objects have some sub-properties listed, unless, as with arrays, they are too plentiful, in which case they're abbreviated.

Any properties which are not scalar can be drilled down into by clicking the **+** sign at the left side of the property. This will result in an automatic `console.dir()` call for the property and possibly new drill-down properties. Let's see what happens when we drill down to the `node` property of the charting object we have dumped:



The picture has been somewhat cut to focus on the essentials. We see that the 'node' property is indeed an HTML DOM node, and most probably the DOM node in the web page where the chart has been given leave to plot itself.

We also note the id of the DOM node and the parentNode of the node, among other things. What you see when you dump an object in this manner depends of course on what kind of an object it is. I strongly recommend you to use `console.dir()` extensively when you start to create your own first small test applications for everything from arrays and Dojo classes to your own objects.

It will give you a much greater understanding of how JavaScript works and most probably make things you've read here and elsewhere connect finally when you can see their effects directly.

Event handling in Dojo

Dojo has very good event support. One of the most classic problems when managing events is the huge differences between everything else and Internet Explorer, leading to a lot of extraneous testing code which has been both hard to write and to maintain.

One of the best things with Dojo's event system is the normalized event object which gets passed on all normal browser events. A normalized event is an event object that looks the same regardless of which type of event it was or which browser it occurred in.

Dojo also has functionality that extends its event model to include function calls as events as well. This means that just as you can hook up your function to be called when the "onclick" event fires on a HTML DOM Button element, you can instead hook it up to be called *when another function is called*. This makes it surprisingly simple to extend functionality, even for objects that do not provide for it explicitly.

Let's say that you have a button that you want to connect a function to. Dojo lets you use the `dojo.connect` function to do it like this:

```
connection = dojo.connect(dojo.byId('mybutton'), 'onclick', foo);
```

And when you need to remove the connection from the element, to avoid memory leaks and the such, you use `dojo.disconnect`:

```
dojo.disconnect(connection);
```

The function that gets called when the event occurs must take one argument, which is the normalized event object.

```
function foo(event)
{
   // Do something with the event argument
}
```

The event object contains a lot of properties, with the following being the most commonly used:

- `currentTarget`: The DOM node that generated the event. Most probably what you want to act upon.
- `layerX` and `layerY`: Relative x and y of the mouse pointer to the target.
- `pageX` and `pageY`: x and y co-ordinates of the mouse pointer to the current window.
- `relatedTarget`: For events like `onmouseout`, the element which the mouse moved out to.
- `charCode`: A unified code for the character being pressed, if any.

The charcodes are defined in `dojo.keys` and are provided here as a quick reference:

- `BACKSPACE`
- `TAB`
- `CLEAR`
- `ENTER`
- `SHIFT`
- `CTRL`
- `ALT`
- `PAUSE`
- `CAPS_LOCK`
- `ESCAPE`
- `SPACE`
- `PAGE_UP`
- `PAGE_DOWN`
- `END`
- `HOME`
- `LEFT_ARROW`
- `UP_ARROW`
- `RIGHT_ARROW`
- `DOWN_ARROW`
- `INSERT`
- `DELETE`
- `HELP`
- `LEFT_WINDOW`

- RIGHT_WINDOW
- SELECT
- NUMPAD_0
- NUMPAD_1
- NUMPAD_2
- NUMPAD_3
- NUMPAD_4
- NUMPAD_5
- NUMPAD_6
- NUMPAD_7
- NUMPAD_8
- NUMPAD_9
- NUMPAD_MULTIPLY
- NUMPAD_PLUS
- NUMPAD_ENTER
- NUMPAD_MINUS
- NUMPAD_PERIOD
- NUMPAD_DIVIDE
- F1
- F2
- F3
- F4
- F5
- F6
- F7
- F8
- F9
- F10
- F11
- F12
- F13
- F14
- F15
- NUM_LOCK
- SCROLL_LOCK

So to connect an event handler function which listens the *ENTER* key on a textfield, you might write the following:

```
<script>
dojo.addOnLoad(function()
{
   function keyHandler(e)
   {
      if (e.charCode == dojo.keys.ENTER)
      {
         console.log("keyHandler called. ENTER pressed");
      }
   }
   dojo.connect(dojo.byId('mytextfield'), 'onkeypress', keyHandler);
}
</script>
<label for="mytextfield" >Text:</label>
<input type="text" id="mytextfield">
```

In all modern web browsers, events (like a mouse click) are first sent to the innermost element in the DOM hierarchy where it occurred, to see if there has been any handlers defined for it. For example, a mouse click on a table will be sent to an image inside a table cell first.

If no handlers are found, the event is then sent to the next element higher up in the DOM hierarchy, in this case the table cell. This is called 'Bubbling' and takes place until either the event is stopped by a function call, or the event reaches the top of the document.

To stop an event from being caught by other handlers higher up, you can use one of the following two Dojo functions:

- `event.preventDefault()` stops this actual event from doing what it normally would do. Send a form or open a page.
- `event.stopPropagation()` stops the event from bubbling.

You can also call `dojo.stopEvent(event)` from anywhere to get both the functions in one go.

The function `dojo.connect` can also be used for connecting event handlers to function calls.

Suppose you have a function `foo`. Suppose also that you very much would want your own function `bar` to be called every time `foo` is called.

```
// foo is defined globally somewhere in the page.
function bar()
{
   console.log("Bar called!");
}
dojo.connect(foo, bar);
```

And that's it!

Sometimes you have different contexts for the respective functions. The function `foo` might live inside its own important object, and your own `bar` function might be defined inside an instantiated widget. You can then provide the contexts alongside the functions in the call, like this:

```
dojo.connect("foo", importantObject, "bar", myInstantiatedWidget);
```

There is one more way to use events in Dojo, and that is the 'pubsub' or publish/subscribe system. Don't let the name scare you off. I guarantee that it is an absolute breeze to both understand and use, and it will save you so much time in simplifying your code you'll feel silly. OK, maybe not you, but that's how I felt.

Pubsub (as you wll see it commonly refered to in documentation and source code) is what is normally referred to as a message bus. It is also the most simple message bus imaginable. What it does is that it allows you to send a message (an object) to all listeners of a channel (a string).

To listen to a channel, we can do the following:

```
function myFunction(arg)
{
   console.log("myFunction got called with these args..");
   console.dir(args);
}
handle = dojo.subscribe("myTopic", myFunction);
```

To get Dojo to call our handler function, we need to publish something to the channel as well:

```
   dojo.publish("myTopic", [0, 1, 2, 3]);
```

This means that we can connect any number of functions, at our leisure, to listen to a channel, so that whenever anything gets published to it, all will be called.

The main issue here is that you can decide at runtime what to listen to, and you can do it dynamically. Another important fact is that two functions that are arbitrarily deep inside their own class and object hierarchies can still communicate without adding any cruft or changing their behavior (much).

JSON

JSON is a format for sending JavaScript objects between client and servers. The acronym stands for **Javascript Object Notation**, and was first proposed by Douglas Crockford in the 3rd ECMAScript definition (see `http://www.json.org/`).

The goal with JSON is to provide an alternative to XML as data-interchange format, which can be both simple to read by humans and simple to parse by machines.

It is basically a very simple object literal format where arrays can be used to group sub-items, with some additional conversion constraints for wire transfer.

A typical JSON object might look like this (Example source from the `json.org` site):

```
{
    "glossary": {
      "title": "example glossary",
        "GlossDiv": {
          "title": "S",
            "GlossList": {
              "GlossEntry": {
                "ID": "SGML",
                "SortAs": "SGML",
                "GlossTerm": "Standard Generalized Markup Language",
                "Acronym": "SGML",
                "Abbrev": "ISO 8879:1986",
                "GlossDef": {
                  "para": "A meta-markup language, used to create
                   markup languages such as DocBook.",
                 "GlossSeeAlso": ["GML", "XML"]
                    },
                 "GlossSee": "markup"
                }
            }
        }
    }
}
```

You recognize the 'standard' object literal format, where an object is defined (using `{}`), and where each property inside the object is separated by a comma, and whose name and value are separated by a colon. All names and values are considered string values and should be double-quoted, with escaping of quotes that appear inside strings themselves.

Note also that some sub-items are defined as objects (`"GlossList"`) and some as arrays (`"GlossSeeAlso"`).

Dojo has great support (of course) for reading and writing JSON data. The base Dojo object has several specialized functions for dealing with JSON.

The following lines are taken directly from the online API documentation at `http://dojotoolkit.org/`, which have the following format for describing functions:

- Arguments are separated by commas (just like in real life).
- Each argument has one or more possible types after a separating colon.
- Sometimes an argument can be of several types, which are then separated by a vertical bar.
- Some arguments are optional, which will then have their types names appended with a question mark.

```
dojo.formToJson(formNode: |Booleandojo.formToJson(formNode: |dojo.
formToJson(formNode: |, prettyPrint: dojo.formToJson(formNode: |
```

This code takes as its first argument either a string defining an Id of a DOM node form element, or a direct reference to the DOM node itself. The second optional argument is a boolean which defines whether the JSON output should be pretty-printed (human-readable) or not.

```
dojo.fromJsonStringdojo.fromJson(json:
```

This code segment takes an object literal JSON string and returns an object.

```
dojo.toJsonObjectdojo.toJson(it: Booleandojo.toJson(it: ,
prettyPrint: Stringdojo.toJson(it: , prettyPrint: ?, _indentStr:
```

This segment of the code works like `dojo.formToJson` but on any object instead. This function `prettyPrint` works the same, and the `_indentStr` optional argument is the string to use for indentation.

Summary

- JavaScript can represent objects in a format called "object literal", which is used extensively by Dojo.
- Remember to never use dangling commas in object literals.
- Functions are first-class objects in JavaScript and are frequently used as arguments to other functions.
- Closures are persistent memory created on each function call, which remains even after the function has returned.
- Dojo includes its own object-oriented class system built upon JavaScript's prototype-based class system, which supports multiple inheritance.
- Dojo provides a parser which can search the page for Dojo widgets (Dijits) defined in any DOM element, and create them.
- Firebug is very good to use for testing web pages. A basic knowledge of Firebug can be very helpful.
- Dojo has a normalized event system, which means that all event objects look the same and use the same codes, regardless of which browser they occur in.
- JSON is a standard format for representing JavaScript objects when sending them to a server or receiving them from a server.

3
Basic Dijit Knowledge

A Dijit or Dojo Widget is an instance of a Dojo class that lives in the page and usually has a DOM node associated with it. Examples are `FloatingPane`, `Tree`, `NumberSpinner`, and `ValidationtextBox`.

The basic premise of a Dijit is that you should be able to create any number of widgets of the same type on the same page, regardless of where on the page you choose to put them. Certain Dijits have special requirements, naturally, such as the Layout Containers, which assume that they are surrounding `ContentPane` Dijits.

In general, Dijits are very versatile and much time has been spent to assure portability and modularization of the Dijit system. Best of all, this is accessible to you as a developer as well.

All Dijits can be subclassed to change parts of their behavior, and then used as the original Dijits, or you can create your own Dijits from scratch and include existing Dijits (Forms, buttons, calendars, and so on) in a hierarchical manner.

All Dijits can be created in either of the following two ways:

- Using the `dojoType` markup property inside selected tags in the HTML page.
- Programmatic creation inside any JavaScript.

For instance, if you want to have a `ColorPalette` in your page, you can write the following:

```
<div dojoType="dijit.ColorPalette"></div>
```

But you also need to load the required Dojo packages, which consist of the `ColorPalette` and any other things it needs. This is generally done in a script statement in the `<head>` part of the HTML page, along with any CSS resources and the `djConfig` declaration. So a complete example would look like this:

```
<html>
   <head>
      <title>ColorPalette</title>
      <style>
        @import "dojo-1.1b1/dojo/resources/dojo.css";
        @import "dojo-1.1b1/dijit/themes/tundra/tundra.css";
      </style>
      <script type="text/javascript">
          djConfig=
         {
            parseOnLoad: true
         }
      </script>
      <script type="text/javascript" src="dojo-1.1b1/dojo/dojo.js">
      </script>
      <script type="text/javascript">
         dojo.require("dojo.parser");
         dojo.require("dijit.ColorPalette");
      </script>
   </head>
   <body class="tundra">
      <div dojoType="dijit.ColorPalette"></div>
   </body>
</html>
```

Obviously, this shows a simple color palette, which can be told to call a function when a choice has been made. But if we start from the top, I've chosen to include two CSS files in the `<style>` tag.

The first one, `dojo.css`, is a `reset.css`, which gives lists, table elements, and various other things their defaults. The file itself is quite small and well commented.

The second file is called `tundra.css` and is a wrapper around lots of other stylesheets, some are generic for the theme it represents, but most are specific for widgets or widget families. Themes will be covered more later in this chapter.

The two ways to create Dijits

So putting a Dojo widget in your page is very simple. If you would have want the `ColorPalette` dynamically in a script instead, remove the highlighted line just before the closing body tag and instead write the following:

```
<script>
   new dijit.ColorPalette({}, dojo.byId('myPalette'));
</script>
```

This seems fairly easy, but what's up with the empty object literal ({}) as the first argument? Well, as some Dijits take few arguments and others more, all arguments to a Dijit get stuffed into the first argument and the others, the last argument is (if needed) the DOM node which the Dijit shall replace with its own content somewhere in the page.

The default is, for all Dijits, that if we only give one argument to the constructor, this will be taken as the DOM node where the Dijit is to be created.

Let's see how to create a more complex Dijit in our page, a NumberSpinner. This will create a NumberSpinner that is set at the value '200' and which has '500' as a maximum, showing no decimals. To create this NumberSpinner dynamically, we would write the following:

```
<input type="text" name="date1" value="2008-12-30"
dojoType="dijit.form.DateTextBox"/>
```

One rather peculiar feature of markup-instantiation of Dijits is that you can use almost any kind of tag for the Dijit.

The Dijit will replace the element with its own template when it is initialized. Certain Dijits work in a more complicated fashion and do not replace child nodes of the element where they're defined, but wrap them instead.

However, each Dijit has support for template HTML which will be inserted, with variable substitutions whenever that Dijit is put in the page. This is a very powerful feature, since when you start creating your own widgets, you will have an excellent system in place already which constrains where things will be put and how they are called.

This means that when you finish your super-complicated graph drawing widget and your client or boss wants three more just like it on the same page, you just slap up three more tags which have the `dojoType` defining your widget.

How do I find my widget?

You already know that you can use `dojo.byId('foo')` as a shorter version of `document.getElementById('foo')`. If you still think that `dojo.byId` is too long, you can create a shorthand function like this:

```
var $ = dojo.byId;
```

And then use `$('foo')` instead of `dojo.byId` for simple DOM node lookup.

But Dijits also seem to have an id. Are those the same as the ids of the DOM node they reside in or what? Well, the answer is both yes and no.

All created Dijit widgets have a unique id. That id can be the same string as the id that defines the DOM node where they're created, but it doesn't have to be.

Suppose that you create a Dijit like this:

```
<div id='foo' dojoType='dijit._Calendar'></div>
```

The created Dijit will have the same Dijit id as the id of the DOM node it was created in, because no others were given. But can you define another id for the widget than for its DOM node? Sure thing. There's a magic attribute called `widgetId`. So we could do the following:

```
<div id='foo' dojoType='dijit._Calendar' widgetId='bar'></div>
```

This would give the widget the id of `'bar'`.

But, really, what is the point? Why would we care the Widget / Dijit has some kind of obscure id? All we really need is the DOM node, right? Not at all. Sure, you might want to reach out and do bad things to the DOM node of a widget, but that object will not be the widget and have none of its functions.

If you want to grab hold of a widget instance after it is created, you need to know its widget id, so you can call the functions defined in the widget. So it's almost its entire reason to exist!

So how do I get hold of a widget obejct now that I have its id? By using `dijit.byId()`. These two functions look pretty similar, so here is a clear and easy to find (when browsing the book) explanation:

- `dojo.byId()`: Returns the DOM node for the given id.
- `dijit.byId()`: Returns the widget object for the given widget id.

Just one more thing. What happens if we create a widget and don't give either a DOM or widget id? Does the created widget still get an id? How do we get at it?

Yes, the widget will get a generated id, if we write the following:

```
<div dojoType='dijit._Calendar'></div>
```

The widget will get a widget id like this: `dijit__Calendar_0`. The id will be the string of the file or namespace path down to the `.js` file which declares the widget, with `/` exchanged to `_`, and with a static widget counter attached to the end.

What's in the fridge? A smorgasbord of tasty Dijits

Most Dijits reside under the `dijit` subdirectory of your Dojo installation. Some reside under `dojox`, though, because they are in beta, or recently contributed to Dojo and not yet completely tested. Here is a fairly thorough list (for Dojo 1.1.0) that you can use as an inspiring starting point:

Dijit	Comment
`dijit.ColorPalette`	A simple color picker. For more advanced use, I recommend the recently introduced `dojox.ColorPicker`, mentioned further down.
`dijit.Dialog`	A modal dialog widget which is initially hidden, together with its content. Will "dim" the rest of the screen when shown.
`dijit.Editor`	An advanced editor, which sports a large toolbar and support indents, lists, fonts, and more. It can return its content as either HTML or text.
`diit.InlineEditBox`	The famous inline editing widget. Only shows a small editing area for parts of a text when called for, otherwise hides.
`dijit.Menu`	Normally used as either a vertical or horizontal pop-up contextual menu, but can also be used "statically" meaning it is shown all the time, where you please.
`dijit.ProgressBar`	A progress bar which can be updated in increments or played as an animation until stopped. Good for Ajax calls, and so on.
`dijit.TitlePane`	Is a pane which can be collapsed (except for the title bar) either by a button on the title bar or programatically. It is similar to the AccordionContainer layout manager, except that it only manages one pane.
`dijit.ToolBar`	The same toolbar that the editor uses. Great when you're developing desktop-like web applications. Great for drop-down menus, buttons, or ComboBoxes.

Dijit	Comment
`dijit.ToolTip`	A classical tooltip that displays a hovering box of content on a mouseover event, for any element. Can contain arbitrary content and even lazily load content from a URL.
`dijit.Tree`	A tree that uses the `dojo.data` data model to build a collapsible tree which can be configured to call a function when any element in the tree is clicked.
`dijit.form.Form`	A very useful form wrapper which can override default actions and serialize all elements as an associative array, among other things.
`dijit.form. Button`	Contains the variants Button, DropDownButton, and ComboButton which provide a nice-looking Button widget, and two variations of a Button containing a Menu.
`dijit.form. CheckBox`	Contains Dojo styled versions of Radio button and CheckBox. Also provides a ToggleButton, a Button element which can be set or unset (pressed in or not).
`dijit.form. CurrencyTextBox, DateTextBox, NumberTextBox and TimeTextBox`	These variations of the classic INPUT text tag, provide specialized input dialogs (for date and time selection) and specific constraints for string, time, date, and monetary input values. This means that they will give user feedback that the number was out of range, for example.
`dijit. FilteringSelect`	Is a SELECT tag which can be populated dynamically and is `dojo.data` enabled, so that it can be configured using Ajax from a URL on the server.
`dijit. MultiSelect`	A variant of FilteringSelect which can accept multiple choices.
`dijit. TextArea, dijit. SimpleTextArea`	TextArea wraps the tag with the same name providing resizing for the text contained, whereas SimpleTextArea stays the same size.
`dijit.Slider, dijit.Spinner`	Two Dijits which allow either dragging a marker (Slider) or using up/down buttons (Spinner) to enter numeric values.
`dijit.layout. ContentPane`	Basic layout tile for other content, for the Container classes. It is usually recommended to set its `title` property, which is used in various containers, when presenting the Pane as minimized.
`dijit.layout. AccordionContainer`	Orders a number of ContentPanes so that only one is visible at a time, with the others shown minimized. It uses an easing motion to switch between ContentPanes.
`dijit.layout. BorderContainer`	Orders the ContentPanes it surrounds according to tag properties set on them, like region="left", "top", and so on.
`dijit.layout. TabContainer`	Orders the ContentPanes it surrounds into Tabs, where each Tab presents the title property of each ContentPane. Pressing the tab icon reveals the corresponding ContentPane and hides the others.

Dijit	Comment
`dijit.layout.StackContainer`	Works just like the tabContainer, but shows no tabs. Instead which ContentPane is displayed is controlled programmatically.
`dojox.Grid`	A highly configurable table component with sorting, summary rows, custom rendering and sorting, and cell editors. Uses `dojo.data` for reading and writing (on cell editor changes) to and from the server. Can be found in a "data-free" version called Virtual grid which can be populated independently.
`dojox.layout.DragPane`	Makes its content draggable back and forth, bringing in different parts if larger than the pane.
`dojox.layout.ExpandoPane`	Works like a ContentPane, but can be expanded or contracted just like a TitlePane.
`dojox.layout.FloatingPane`	A non-modal floating window. Can sport buttons like minimize, maximize, and/or close. Can be dragged around, resized, and minimized to provide a default generated docking element. Good for jaw-dropping effects. Head-smackingly simple to use.
`dojox.layout.RadioGroup`	Works like TabContainer, except it uses onmouseover sensitive buttons to wipe in and out different ContentPanes.
`dojox.layout.ResizeHandle`	Really useful little widget. Takes a DOM node as the property to which it attaches itself and provides resizability. Nice.
`dojox.layout.ScrollPane`	Organizes all the elements it surrounds and creates horizontal or vertical stripes which roll the elements back or forth in the viewport, depending on which side of the ScrollPane the mouse is over. Reminiscent of an analog sliding scale.
`dojox.widget.ColorPicker`	This is a classical high-resolution color picker, in the vein of Photoshop. If you need rgb values with saturation, hue, and so on, this is for you.
`dojox.widget.FileInput`	A widget which wraps the input type="file" chooser, and can send the file asynchronously with Ajax, without disturbing the loaded page.
`dojox.widget.FisheyeLite`	Creates a horizontal or vertical "fisheye lens" effect much like the Mac's program bar. As the mouse pointer gets closer to an element, its size increases, and then decreases again as it leaves the element.
`dojox.widget.Loader`	A discrete widget that listens to global xhr (Ajax) events and provides feedback if there are outstanding requests from the server. Is used in an example to change the mouse pointer when "loading".
`dojox.widget.MultiComboBox`	Accepts multiple values with Ajax-enabled drop-down choice list for each.

Dijit	Comment
`dojox.widget.Rating`	You've been looking for this. Seriously. It shows stars that can be grayed out or starred depending on where the user clicks. Think about `amazon.com`. Choices are sent to the URL of choice in big Ajax style.
`dojox.widget.SortList`	A simple alternative to the Grid, if all you want to do is to list elements in a sorted order.
`dojox.widget.Toaster`	The Toaster slides in its contents from either of the screens four sides. Useful for shopping-carts or menu-like effects.
`dojox.widget.Wizard`	A variation on the StackContainer theme, where a user is presented a series of choices, with buttons advancing to the next "screen" in the wizard each time.

The Dijit string listed in the first row can be used in a `dojo.require("");` statement. At times, several Dijits are provided in one requirement. Some Dijits have been left out, considered to be superfluous or of minor importance.

This is only a list of Dijits, though. There are quite a lot of other components that live in the page that we'll get to in later chapters.

The `Dojo Tree` and `Grid Dijits` are complex and powerful enough to call for their own chapter, later on in this book.

Also, several components (not only Tree and Grid, but also `ComboBox` and `SortableList`, among others) can consume data stores, which will be covered separately.

Dijit structure

The most basic definition of a Dijit is that it is a Dojo class which is associated with a certain DOM element in a web page. It is, in any other aspects, a regular Dojo class. The `_Widget` base class (that is the ultimate root class of all Dijits) does add a number of extra handle properties and functions to all Dijits, such as the variable `domNode` which refers to the DOM node which the Dijit is associated with.

All Dijits also have a lifecycle (described below) where Dojo ensures that certain functions on the Dijit are called at various times. There is for example one function that will be called before the Dijit is actually displayed and another one, which is called before any child Dijits are created, and so on.

Lifecycle

Regardless of whether a widget gets instantiated by the parser searching the page (or parts of it) for widgets to create, or by you creating it programmatically, certain functions get called at certain times.

In the base widget class `dijit._Widget`, we see the following comments:

```
create: function(/*Object?*/params, /*DomNode|String*/srcNodeRef){
   //summary:
   //Kick off the lifecycle of a widget
   //description:
   //To understand the process by which widgets are instantiated,
     it
   //is critical to understand what other methods create calls and
   //which of them you'll want to override. Of course, adventurous
   //developers could override create entirely, but this should
   //only be done as a last resort.
   //
   //Below is a list of the methods that are called, in the order
   //they are fired, along with notes about what they do and if
    /when
   //you should over-ride them in your widget:
   //
   //* postMixInProperties:
   //* a stub function that you can over-ride to modify
   //variables that may have been naively assigned by
   //mixInProperties
   //* widget is added to manager object here
   //* buildRendering:
   //* Subclasses use this method to handle all UI initialization
   //Sets this.domNode.  Templated widgets do this automatically
   //and otherwise it just uses the source dom node.
   //* postCreate:
   //* a stub function that you can over-ride to modify take
   //actions once the widget has been placed in the UI
```

The `postMixInProperties` function is called right after Dojo has mixed in all functions and properties of the widgets' superclasses. Most widgets have the classes `_Widget` and `_Templated` mixed in, or subclasses of them. This is the function where you set important properties of the widget before it is displayed.

The function `buildRendering` is then called, which normally builds the UI and sets the `domNode` property of the widget, to be used in functions as `'this.domNode'`. The variable `_Templated` overrides this to build the widget according to a template string or file. Override this only if you've read the source code a couple of times. And understand it.

The function `postCreate` is next, which is provided as a stub function which is called once the widget is visually inserted in the page.

The function `startUp` is next, which gets called for all widgets once all widgets in the page have been rendered properly, but before any child widgets have been created, so that any necessary properties can be messaged first.

The function `uninitialize` gets called when the widget is being destroyed, usually due to a call to `destroyRecursive` on the widget itself or to a parent widget.

Now here comes an important fact: You never need to implement any of these functions, unless you feel that you need to. All are implemented fully, or as stubs for your convenience, in superclasses of all widgets.

Templates

The whole point with a widget is that it should be visible. All widgets comprise at least one HTML element. The element which makes up a widget can be created programmatically in an overridden function, but for most purposes you get much cleaner code by separating the looks from the logic, so to speak.

Dojo actually supports two different templating langauges: the simple default and the Django Template Language (DTL, implemented originally in the Python Web framework of the same name).

Note that if you're using the first (standard) templating language, you're only allowed one element in the template! This is not as bad as it sounds. What it means is that you can only have one element with any number of children.

What will get you into trouble is if you have more than one element in your template parallel, or side-by-side, then only the first will get parsed.

The first supports simple variable substitutions, using `${..}` syntax. For instance, you can define a HTML template snippet for a widget that includes a div tag which gets a custom id based upon the id given to the widget using the following:

```
<div id="${id}_custom"></div>.
```

That id would be the same property that is set as id="..." in the page where the widget is created and which also can be accessed using `this.id` inside the functions of the widget.

So all properties being set in the widget, by instantiation or dynamically (but before the template is rendered) can be used in this way inside the template string or file.

For a concrete example, look at the template for `dijit.form.TextBox`:

```
<input class="dijit dijitReset dijitLeft"
dojoAttachPoint='textbox,focusNode'
name="${name}"
dojoAttachEvent='onmouseenter:_onMouse,onmouseleave:_onMouse,onfocus:_
onMouse,onblur:_onMouse,onkeypress:_onKeyPress'
autocomplete="off"
type="${type}"
/>
```

Inside the `dijit/form/TextBox.js` file, there are no definitions for these variables, however. But if we walk the class hierarchy we see that TextBox derives from a class called `dijit.form._FormValueWidget`. This class, in turn, inherits from `dijit.form._FormWidget` (and both superclasses are defined in the file with the same name.) Finally, we get to the variables:

```
...
   //Name used when submitting form; same as "name" attribute or
   //plain HTML elements
   name: "",
   // alt: String
   //Corresponds to the native HTML <input> element's attribute.
   alt: "",
   // value: String
   //Corresponds to the native HTML <input> element's attribute.
   value: "",
   // type: String
   //Corresponds to the native HTML <input> element's attribute.
   type: "text",
...
```

So now you see that there is no magic underlying basic templating in Dojo. As long as you have a variable defined in a widget, even if it comes from a superclass or mixin (as you remember, a mixin is another (or more) multiple superclass, or a superclass added at runtime), it can be referred to in the template for the widget with the ${} syntax.

Attach points

Templates might contain fairly complex things, and it is not at all unusual to have the need to access DOM nodes inside the widget template after it has been instantiated. You might want to show or hide a dialog, or maybe add items dynamically to a list.

Either way, the quick and dirty way of doing this (as shown above) is to give the DOM node you want to access inside the widget template a dynamic id based upon the id of the widget, like this:

```
<div id="{id}_foo">
```

One could then use the following code in the actual JavaScript of the widget:

```
var x = dojo.byId(this.id+"_foo");
```

And `x` would always resolve to that particular node in the template, regardless of how many widgets of the same kind has been created in the page.

As it turns out, Dojo has excellent support for this, and it is called "Attach Points".

An Attach Point is a name you can give to any DOM node or Dijit in the template. Dojo will automatically scan the template for Attach Points and then create contextual `this` variables for the Dijit with the same names.

Consider a template that uses Attach Points instead, which looks like this:

```
<div><div dojoAttachPoint='foo'>XXXX</div><div dojoAttachPoint='bar'>
YYYY</div></div>
```

What will happen when the Dijit is rendered in the page is that two new variables will be instantiated in the context of the Dijit, with the names 'foo' and 'bar', being references to the DOM nodes in question.

In the `postCreate` function of the widget, one could then have a line which reads:

```
this.foo.style.background = "#f94444";
```

This will change the background of the DOM node in the template, without having to know what DOM id it actually has, and without colliding with any other Dijits of the same (or other) type.

Events

But there were some other funny things in the template for TextBox, weren't there? There was a property called `dojoAttachEvent`, which is used expressly for the same reasons that one would use `dojo.connect` inside a script.

Let's look closer at those events:

```
dojoAttachEvent='onmouseenter:_onMouse...' ...
```

It is a series of comma-separated event names and function names pairs. In the case above, the event `onmouseenter` for the DOM node is connected to the function `_onMouse` in the `TextBox` widget class. This would otherwise be the prime reason for using Attach Points, so the code gets cleaner still.

Extension points

An extension point is what Dojo calls a function override inside the usual page. Several widgets aren't particularly useful unless you tell them what to do, such as the Button family. One way of telling a Button what to do is to create a small JavaScript somewhere in the page and use `dojo.connect`.

Another way could be to create you own widget which subclasses the Button widget, and then override the `onClick` function defined inside that widget using an extension point. We begin by checking out parts of the code in `dijit.Button.js`:

```
_onClick: function(/*Event*/ e){
   // summary: internal function to handle click actions
   if(this.disabled || this.readOnly)
     {
      dojo.stopEvent(e); // needed for checkbox
      return false;
     }
   this._clicked(); // widget click actions
   return this.onClick(e); // user click actions
},
```

This function is wired up to take care of click events for the widget. The last line of the widget calls another internal function, onClick, and returns the result of that. onClick looks like this:

```
onClick: function(/*Event*/ e){
   //summary: user callback for when button is clicked
   //if type="submit", return true to perform submit
   return true;
},
```

To override this function by using an extension point, we can put the following in our page when declaring the Button widget:

```
<script>
   function myfunc(e)
    {
      //Do something really clever with the event argument
   }
</script>
<div dojoType="dijit.form.Button" onClick="myfunc"></div>
```

Django Templating Language (DTL)

DTL is quite another order of templating language. Here you can define more complex widget markups using both loops and conditionals.

Where Dojo's classical templating system relied on variable substitutions, leaving the widget to create more complex dynamical markup in special functions and then put them in the template, DTL gives us much more power, or conversely allows our logic to become smaller.

To use DTL inside a widget, you must subclass dojox.dtl.Template and dojox.dtl.HtmlTemplate. The former manages all simple text substitutions, and the latter manages all other complex DOM manipulation.

So when defining a widget, instead of writing this:

```
dojo.declare("mycorp.charting.chart1", [ dijit._Widget,
dijit._Templated],{...
```

You'd instead write the following:

```
dojo.declare("mycorp.charting.chart1", [ dijit._Widget,
dojox.dtl._Templated],
{...
```

For example, we have a widget which reads data from the server using `dojo.xhrGet` and then puts all items read in a simple but sortable SortList.

Using a standard templating language, we might have built a template that looked something like this:

```
templateString="<ul dojoType="dojox.widget.SortList"></ul>";
```

And then have some widget code that looks like this:

```
// To be called when the server responds with data
handleData: function(data)
{
   var obj = eval("("+data+")");
   dojo.forEach(obj, function(o)
   {
      var li = document.createElement('li');
     li.innerHTML = "" + o;
      this.domNode.appendChild(li);
   });
}
```

Instead, with DTL, we can change the template too look like this instead:

```
templateString="<ul dojoType="dojox.widget.SortList">{% for item in
obj %}<li>{{ item }}</li>{% endfor %}</ul>";
```

The code will be changed to only set the `obj` variable, which is used in the template, and the template itself then manages looping and variable substitution all on its own.

```
// changed to use DTL functionality instead. Obj is now an instance
//variable.
handleData: function(data)
{
   this.obj = eval("("+data+")");
}
```

To actually see the rendered data, DTL requires an extra step. You must at one point, when your variables are set up correctly, call the render function from `dojox.dtl._Templated`. But since your widget has mixed that class in, it is part of the 'this' context.

```
handleData: function(data)
{
   this.obj = eval("("+data+")");
   this.render();
}
```

However, `dojox.widget.SortList` is actually able to load its contents dynamically from a `dojo.data store` without any intervening widgets at all. I only use it as a daft but simple example on how one usually can go about creating dynamic content in a widget.

First, let's take a look at the features of DTL before we dive down into a longer example.

Variable substitution

Variables that we want to display from the widget are marked up differently.

```
{{person_name}}
```

instead of

```
${person_name}
```

Loops

The loops in DTL are reminiscent of server-based templating languages of yore, especially ASP and JSP.

Of course, these days the server need not do any templating or web magic at all, it only sends us data which gets rendered every which way in the Dojo client.

The variable `item_list` showing up in the example below are variables that exist in the 'this' context of the widget, and can be put there using widget parameter passing 'the usual way', just so you know.

```
<ul>
{% for item in item_list %}
<li>{{ item }}</li>
{% endfor %}
</ul>
```

Conditionals

```
{% if ordered_warranty %}
<p>Your warranty information will be included in the packaging.</p>
{% else %}
<p>No warranty for you!</p>
{% endif %}
```

Argument passing

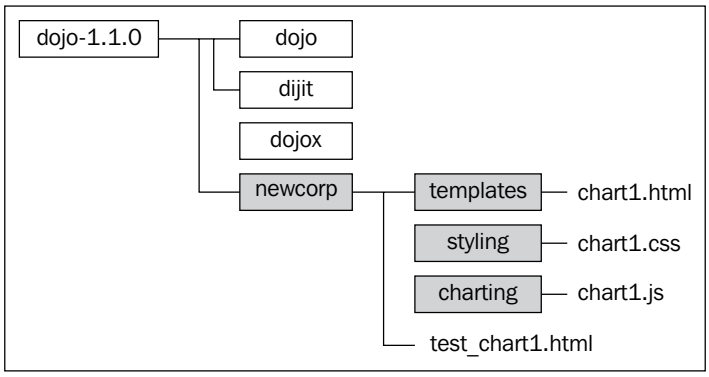
As you remember from Chapter 2, a Dijit is just a Dojo class with some special properties. One of those properties is that if the Dijit class declares an explicit class variable, that variable will automatically be detected and initialized if present in the Dijit argument object

A recommended widget structure

The function `dojo.require` is a very powerful JavaScript loader. By creating a separate directory parallel to the Dojo directories, you will be able to load you own widgets (including having them auto-loading any of their dependencies) using just one `dojo.require` statement in your page.

There's another way to point out where to load classes from (this does of course not only relate to widgets, but we'll get to that in a minute).

This is how you might want to set things up for a new customer or project.



This is just an example, and you might want to set things up differently for a variety of reasons. We'll take each non-standard directory and file at a time and describe the reasons for where they're put.

File/Directory	Meaning
`newcorp`	This is the name of your new project. The reason for putting this directory here is not only to make it simpler to use `dojo.require` for your project, but also to encapsulate all of your functionality in one folder.
`templates`	Have at least one separate folder for your templates. As you'll see, you can use an inline string in the widget to avoid putting the widget template in a separate file. I advise against it, at least in the beginning. A separate file is great to pass around, and to change separately. Sometimes you might choose to create separate subdirectories in the `templates` folder. This is helpful if you have many templates that yet can be divided into groups of widgets.
`styling`	Here I put the css files that are unique for each widget. Note that Dojo does not auto-load these css files, so you will still have to include them manually into your main html file. Even so, I have come to rely upon having a separate directory for styling, to further modularize things from each other and to be able to focus on just the current widget.
`charting`	First things first here; charting is not a magic name in Dojo, it's just a sample name of the first widget created for the `newcorp` project. It could have been named `foo` instead. But following the structure of the rest of Dojo, each area of functionality has its own subdirectory. If you were to create another component for newcorp that was a simple shopping cart, the next directory might be named `sc` or `cart`, parallel to this one.

File/Directory	Meaning
`test_chart1.html`	I usually put my test files in the root directory of the project. This is not completely normal, and Dojo usually provides a `test` subdirectory.
`Chart1.html`	The HTML template for the `mycorp.charting.chart1` widget
`Chart1.css`	The CSS styles for the same
`Chart1.js`	The actual widget class which declares the widget and points out the template file.

No matter how you choose to arrange the files, there are only two places which look up all the other things:

- The prime HTML file which actually loads the widget which you've created.
- The `.js` file that defines the widget.

The parts which are critical here come usually in the beginning of the widget. Let's look at a small example which fits into the scenario described above: the `chart1.js` widget:

```
dojo.provide("mycorp.charting.chart1");
dojo.require("dijit._Templated");
dojo.require("dijit._Widget");
dojo.require("dojox.charting.Chart2D");
dojo.declare("mycorp.charting.chart1", [ dijit._Widget, dijit._
Templated],
{
   templatePath: dojo.moduleUrl("mycorp","templates/chart1.html"),
   widgetsInTemplate: true,
   startup: function()
   {
      console.log("startup for mycorp.charting.chart1 called.");
      var ch = this.corpchart;
      console.log("corpchart "+ch+" has id "+ch.id);
      var chart1 = new dojox.charting.Chart2D(this.corpchart.id);
      chart1.addSeries("Series A", [1, 2, 5, 4, 1, 3, 2,3,2,4,3,4],
      {stroke: {color: "red", width: 2}});
      chart1.addSeries("Series B", [7, 6, 3, 5, 4, 6, 5,6,7,5,2,5],
      {stroke: {color: "blue", width: 2}});
      chart1.render();
    }
});
```

The first line of the widget declaration is `dojo.provide`. That call must contain a string that exactly references where in the namespace this widget (or class, actually) exists. Since this widget exists in the file `mycorp/charting/chart1.js`, the string will be `mycorp.charting.chart1`.

Then follows a number of required statements. The big upside with having required statements inside the widget definition is that it leaves the main HTML page relatively uncluttered. In fact, with some luck, all the script references you'll need is the one for `dojo.js`, and then a required statement for your widget. The rest of the Dojo classes will be pulled in dynamically as their required statements get discovered.

Let's have a look at the test page for our class:

```
<html>
<head><title>Chart1 Test</title></head>
   <style type="text/css">
      @import "../../../dojo/resources/dojo.css";
      @import "styling/chart1.css";
   </style>
    <script>
        var djConfig={ parseOnLoad: true};
    </script>
    <script type="text/javascript" src="../dojo/dojo.js"></script>
    <script>
        dojo.require("dojo.parser");
        dojo.require("mycorp.charting.chart1");
    </script>
<body>
    <div dojoType="mycorp.charting.chart1"></div>
</body>
</html>
```

We include two CSS files: the Dojo reset styles and the specific styles for our widget. It can be a good idea to create versioned CSS files for each package (or project) that combine all styles into one file, for regular milestones. However, for everyday coding, it is generally better to have separate files.

After that we get the `djConfig` declaration which only tells the parser to begin parsing the page for Dijits immediately after load.

Then we pull in the main dojo loader—`dojo.js`, and require first the parser, and then our widget.

The body of the test file is minimal, since all special markup for the widget is folded down into its template.

The CSS file is also very small and looks like this:

```
.mycorp_chart
{
   height: 200px;
   width: 400px;
}
```

And the template itself looks like this:

```
<div dojoAttachPoint="corpchart" class="mycorp_chart"></div>
```

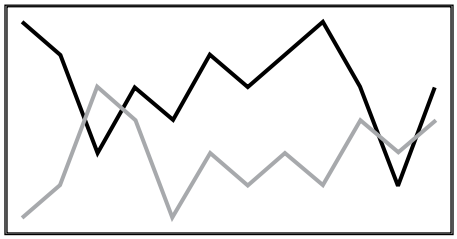
Using the `dojoAttachPoint` property, a varaible gets created inside our widget, which we can refer to as `this.corpchart` and which is a reference to the DOM node in question.

If you try out the example and use firebug, you will see the following in the firebug console:

```
gfx renderer = svg
startup for mycorp.charting.chart1 called.
corpchart [object HTMLDivElement] has id mycorp_charting_chart1_0
```

The id of the widget has been set to the namespace path of the widget, with _ replacing /, and a static counter tucked at the end.

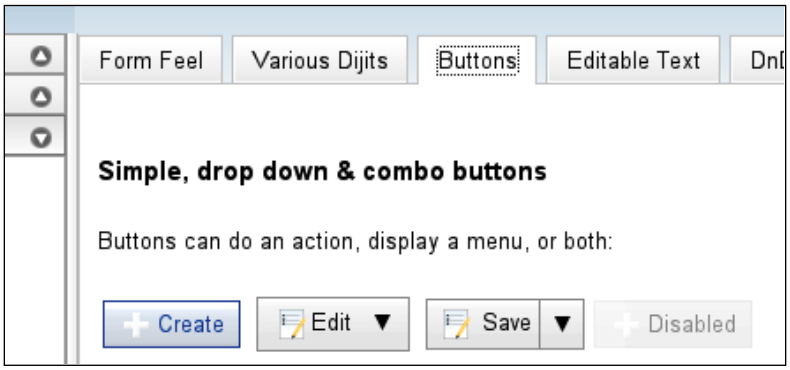
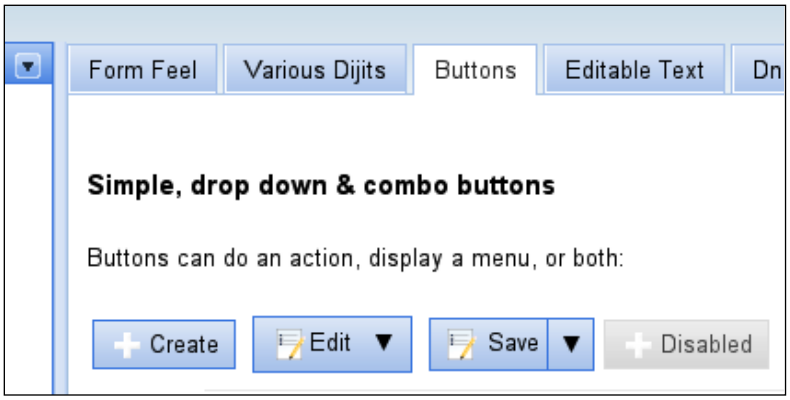
The chart displayed will look approximately like this:



Themes

Themes are css files which make it possible to change the look and feel of Dijits with just a change in a base css class.

Dojo 0.9 came with the standard "tundra" theme with a Ext-like simplicity and elegance. Now in version 1.1 and counting we have the following themes (Images courtesy of `dojotoolkit.org`):

Theme	Looks
Tundra	
Soria	
Noir (Under construction. Use at your own discretion.)	

There's also a fourth standard theme called "a11y", which is a theme for the visually impaired, with larger fonts, greater contrast, and so on.

Themes are set for Dojo widgets in the following way:

- Include the css file for the theme of your choices.
- Set the class of the body tag to the name of the theme.

The first point goes without saying, but the second might raise a couple of eyebrows. I have had clients that have flat out refused to touch the class of the body tag, concerned that the inclusion of the Dojo css files might break something further down in their design. This is a valid concern. It is a bit odd to use themes this way.

The reason for this is that a lot of complex widgets put all or parts of their elements directly under the body tag, so as not to interfere with the rest of the markup on the page.

Stuff are then placed or moved to the correct place on certain events, or just placed correctly from the beginning, but the element in question is still a child node of `document.body`.

This in turn means that if you don't put the theme class on the body tag, you'll end up with calendar widgets that look like dirt, and so on.

It would be unwise to recommend anyone to go against the expressed wishes of a client or project leader and just slap on the theme class anyway on the body tag using the following technique:

```
var theme = 'tundra';
if(!dojo.hasClass(dojo.body(),theme))
{
     dojo.addClass(dojo.body(),theme);
}
```

But it has been known in parts of Sweden to totally rock and make life much easier.

I18N

Dojo uses Internationalization in all the official widgets under the dojo and dijit hierarchies. Only functionality in the experimental dojox hierarchy is exempt from I18N compliance.

Dojo uses specific JavaScript files for each function and language, so that it is comparatively simple to add translations and handling for locales which are not yet supported.

djConfig.locale

By setting the `djConfig.locale` variable, you can change the look and feel of Dojos widgets to better conform with other locales than en_us.

All Dijits that have some kind of semantic responsibility can and will show different behavior when a locale other than default is set.

Under the `dijit` folder in the Dojo directories, there's a folder called `nls`. In this folder, Dojo has put JavaScript files containing an object literal with resource strings for the classes with the same names.

As an example, there's a file called `loading.js` with the following content:

```
({
   loadingState: "Loading...",
   errorState: "Sorry, an error occurred"
})
```

We see that it contains two keys, `loadingState` and `errorState`, with default English values. Under the `nls` directory, there are quite a lot of subdirectories. They are named after the country codes they manage; `cs`, `de`, `fr`, `pt`, `ru`, and so on.

If we check in the `de` subdirectory, there's a `loading-js` file there as well, with the following content:

```
({
   loadingState: "Wird geladen...",
   errorState: "Es ist ein Fehler aufgetreten."
})
```

So, clearly, the `nls` system uses default resources in the base directory, with substitutable files with the same names and key-names, but with locale-corresponding content. Because these are JavaScript files, they can be managed and loaded lazily using the same techniques as the `dojo.require` loader does.

Every major Dijit or Dijit group has its own `nls` subdirectory. Under `dijit.form`, there's another `nls` directory, where we can see the same structure as that for the general `dijit` directory. A small number of default files—`ComboBox.js`, `TextArea.js` and `validate.js`, are replicated in the subdirectory of each language.

If no `djConfig.locale` is configured, Dojo assumes that the locale to be used is the one configured for the browser. The function `djConfig.extraLocale` can be used to set one or more extra locales. Several locales can then be used for different widgets on the page.

Programmatic locale formatting

Using the following objects, you can leverage Dojos local-sensitivity to get dates, currency, and more in current locale format:

- `dojo.number.regexp`
- `dojo.number.parse`
- `dojo.date.locale.regexp`
- `dojo.date.locale.parse`

For example, you can use the following to convert a date into different formats:

```
dojo.require("dojo.date.locale");
var d = new Date(2008,5,11,19,58);
var datestring = dojo.date.locale.format(d);
var ch_datestring = dojo.date.locale.format(d, {locale:'zh-cn'});
```

There's also a convenient system object called `dojo.currency`, which can be used to parse to and from different currency formats and locale specific formatting (which character to use for comma and such). It is used in the following manner:

```
var dollarstring = dojo.currency.format(123.45, {currency: "USD",
locale: "en-us"})
```

If you want to do the opposite, like parsing a numerical value from a correctly formatted currency string, you do this:

```
var value = dojo.currency.parse("$123.45", {currency: "USD", locale:
"en-us"})
```

While we're on the subject, another really useful object is `dojo.number`. Using this, you can format to and from a certain amount of digits, decimals, or templating strings, including locale specific support for commas.

```
dojo.number.format(123, {pattern: "0000"})
```

This gives the string "0123", for instance, whereas

```
dojo.number.format(-1234567.89, {pattern: "#,##,##0.000##"})
```

gives the string "-12,34,567.890".

Accessibility and graceful degradation

Dojo fully supports **Accessible Rich Internet Applications (ARIA)**, which is a standard by the **W3C Web Accessibility Initiative (WAI)**.

In practice, this means that all widgets are degradable and do not require a mouse to operate. Many elements inside the elements are tagged with "wai-roles", which can be used by the rendering browser to enable accessibility functionality for lower resolutions and more.

Firefox can use the roles defined in Dojo's widget templates to explain the page to a screen-reader for the visually impaired, and the roles can also be used for screen-scrapers and to allow for greater transparency towards indexing by search engines.

Summary

- A Dijit is a Dojo widget, and the terms can be used interchangeably.
- Dojo has a large number of useful widgets which are simple to use and easy to extend.
- A Dijit is a Dojo class which inherits from `dijit._Widget`.
- A Dijit can use templates, either as inline strings or using external HTML template files.
- Dojo can use both simple templating and the **Django Template Language (DTL)**.
- Using attach and extension points, it becomes manageable to extend the functionality of a Dijit.
- Using Dojo's own file structure for custom widgets leverages the class loading features greatly.
- Dojo supports Internationalization for Dijits and provides structure for custom i18n endeavors.
- A number of simple formatting functions are provided for dates, currencies, and number formatting.

4
Ajax Communication

Ajax is the very backbone of an Ajax toolkit, providing the developer with useful methods to take full control of the JavaScript client's communication with the server it was loaded from.

In recent years, new ways of transporting data have been implemented that do not require the data to be located on the same server. Even though those methods of transport are of another kind (`dojo.io.script`, below, for example), they are still managed in a similar manner in Dojo.

Dojo also provides some interesting abstractions on top of Ajax communications, such as remote procedure calls and data stores (which will be covered in a chapter of their own).

Dojo Ajax features

XHR is an acronym for XmlHttpRequest. It doesn't actually force you to transfer only XML data, though it was meant for that from the beginning. XHR techniques is the reason Ajax is called Ajax. Ajax – Asynchronous JavaScript And XML.

The idea behind Ajax is to make it possible for scripts inside the web page to send and retrieve data to and from the server without having to reload the page. A side-effect of this is that the page can retain state between data transfers without having to rely on cookies (although the methods can be combined).

In practice, what happens is that the browser opens a new iframe with zero height and width. Each iframe is capable of loading and posting data independently of each other, and so the accesses of the 'invisible' iframe will not affect the main page to be reloaded.

This results in a page whose JavaScript functions are able to post a form and get a reply, which in turn updates only part of the page, flicker-free, so to speak.

The function `dojo.xhr` implements this functionality in Dojo. It is included in the dojo base package, so it is always accessible to your scripts, without the need for any `dojo.require()` statements.

What is good to know about `dojo.xhr` is that it is generally asynchronous. This means that you will have to pass it a callback function as an argument, which will be called at some point in the future.

It is simple to make the basic mistake of writing code that makes sequential assumptions, which will result in undefined and null values all over the place.

Let's check out a small example, from the perspective of the programmer. Let's assume that you create a variable which you want to contain the content of a small text file that exists on the server.

You then create an anonymous inline function as the callback, which sets the variable inside the `dojo.xhr` call, and when the call is done, you try to use the variable. Simple, eh?

The code might look like this:

```
var x;
dojo.xhr("Get",
{
     url: "http://yourserver/yourdata.txt",
     handleAs: "text",
     timeout: 10000,
     load: function(data, ioArgs)
  {
       x = data;
  }
});
console.log("x == "+x);
```

The `dojo.xhr` method takes quite a few arguments. The first argument is the HTTP method to use for communicating with the server. "Get" is the default method used by browsers when accessing web pages, and if no method is mentioned when discussing XHR, you can safely assume that "Get" is implied.

The second argument is an object literal carrying the bulk of the argument parameters. Here, we have the two most important ones: `url` and `load`. We also define a timeout variable which speaks more or less for itself. It's the timeout in milliseconds, after which the function fails with a timeout error.

We'll see later how we can specify error handlers to manage the timeout directly.

The `url` parameter is of course responsible for defining where data is to be loaded from (or in the case of `POST`, posted to).

The `handleAs` parameter defines what kind of content is being transferred. You might think (in a philosophical sense, rightly so) that this is a fine business of the underlying system, but since both browser and server need to negotiate acceptable content-types in the HTTP header, this is actually crucial and can lead to many weird errors if mismanaged.

The `load` parameter defines the callback function, which `dojo.xhr` will call upon a successful retrieval of data from the server.

We could just as well have defined the function beforehand and passed the name of it as a reference, which would have looked like this:

```
var x;
var myCallback = function(data, ioArgs)
{
   x = data;
}
dojo.xhr("Get",
{
     url: "http://yourserver/yourdata.txt",
     handleAs: "text",
     timeout: 10000,
     load: myCallback
});
console.log("x == "+x);
```

The above code is equivalent to the first.

You might note that the signature of the callback function takes two parameters. Not only the actual data delivered, but also a strange one called `ioArgs`. This second parameter contains all the arguments passed to the server in the call, for debugging purposes. You might use this information to match what the server returned with what you sent to it.

Let us return to the basic mistake I just described. It is in the above code, and it's clearly visible in the second variant, but not so easy to spot, in the first.

The danger here is that you might assume that the variable `x` is set before it actually is. When the `dojo.xhr` call is made, it returns immediately. As soon as it returns, the code happily trundles along to the `console.log` statement.

This statement will print out **x == undefined**, since x is not set to anything. Some 150 milliseconds later, or so, the server will return the data, and the callback function will be called, which indeed will set x to the desired state, but long after we've tried to use it.

So, to be certain that we work with x only when the server has replied, we can declare the load function inline, like this:

```
var x;
dojo.xhr("Get",
{
     url: "http://yourserver/yourdata.txt",
     handleAs: "text",
     timeout: 10000,
     load: function(data, ioArgs)
  {
       x = data;
      console.log("x == "+x);  // Act upon the data, only in the
      //callback function.
  }
});
```

The "Hello World" of Ajax

Since Ajax communication implies a server-side component, let's check out a complete round-trip example using a minimal Ajax client and a minimal PHP server, which you can use as a "safe base" for further experimentation.

A simple Dojo page using Ajax looks like this:

```
<html>
   <head>
      <title> Chapter 4 XHR test</title>
      <style type="text/css">
      @import "http://o.aolcdn.com/dojo/1.1.0/dojo/
               resources/dojo.css";
      @import "http://o.aolcdn.com/dojo/1.1.0/dijit/themes/dijit.css";
      @import "http://o.aolcdn.com/dojo/1.1.0/dijit/
               themes/tundra/tundra.css";
      </style>
      <script>
      var djConfig=
      {
          isDebug: true,
          parseOnLoad: true
```

```
        };
        </script>
        <script type="text/javascript" src="http://o.aolcdn.com/
                     dojo/1.1.0/dojo/dojo.xd.js">
        </script>
        <script language="JavaScript" type="text/javascript">
           dojo.require("dojo.parser");
           dojo.require("dijit.form.Button");
           function _onClick()
           {
              var arg = dojo.byId('inp').value;
              dojo.xhr("Get",
              {
                 url:"server_ch4.php?arg="+arg,
                 handleAs: "text",
                 load: function(data, ioArgs)
                 {
                    var ta = dojo.byId('ta');
                    ta.value = ta.value + data;
                 }
              });
           }
        </script>
     </head>
     <body>
        <input type="text" id="inp">
        <div dojoType="dijit.form.Button" id="button"
                       onClick="_onClick();">Send</div>
        <textarea id="ta"></textarea>
     </body>
  </html>
```

The PHP server (including crude but effective logging) looks like this:

```
  <?php
     $log = "";
     $logstart = getcwd();
     function logit($it)
     {
        //---------------------------------------------------------------
        global $log;
        global $logstart;
        $log = $log.$it;
        $fp2 = fopen($logstart."/meh.log", "a");
        $d = date("Y-m-d H:i:s");
```

```
        fwrite($fp2, $d.">".$it."\n");
        fclose($fp2);
        //---------------------------------------------------------------
    }
    $arg = $_GET["arg"];
    logit("server got arg = $arg");
    echo "Server says. You wrote '$arg'";
?>
```

The not-so-exceedingly stylish page should look like this after a round-trip to the server:



The HTML page uses the AOL CDN to load Dojo, which is always recommended for mockups, easy porting, and "mailability" of proof of concepts.

The first part of the page requires the parser and the slightly extraneous `dijit.form.Button` which the parser instantiates for us.

Then a function called `_onClick` is defined, which is meant to be called when the button is clicked. As you can see, it simply takes the value of the text input field and builds a custom URL inside the `dojo.xhr` call.

When the `xhr` call returns with a reply from the server, the `load:` callback function gets called. The `load:` function appends the received textual data from the server to the contents of the textarea in the page's body.

The server is very simple and simply echoes the string passed to it from the client, with some additional info.

This is the most simple and basic round-trip example one can make with Dojo and Ajax. Obviously, it would be possible to have several parts of the page making calls to the server at different points in time, as well as having a server which generated information according to an algorithm or searched for information in a database.

In fact, quite a number of Dojo Dijits are "Ajax-aware", which makes it super-simple to write a simple server-side script which generates data for them. For instance, there's the `dijit.form.ComboBox` which can be configured to be populated directly from the server.

These "Ajax-aware" components are a little bit more complicated, though, than our example above. They use the `dojo.data` model which separates the data access into different layers and makes it possible to use one data source to feed both a tree or a table grid. We'll look into `dojo.data` more in later chapters.

Errors

Just as the `dojo.xhr` function can take a 'load' callback function argument which gets called when the server returns data in an orderly manner, an options 'error' argument of the same type can be provided, and it looks like this:

```
error: function(e, ioArgs)
{
   console.log("An error occurred: "+e);
}
```

The most common error is `timeout`, but others (like `server error 500`) are possible.

Posting data

When comparing the two most popular HTTP methods, `GET` and `POST`, it is not unusual to hear that `POST` sends data "out of band". This does not mean that `POST` somehow sends information by email or using a special radio transmitter in the computer.

Actually, the differences between `GET` and `POST` are primarily that `GET` "sends" its data as part of the URL it uses to access documents on the server. You have seen the long, ampersand-separated strings that look like this:

```
server.php?foo=bar&baz=17&hejbaberiba=xxxyyyzzz
```

`POST` URLs, on the other hand, look like this:

```
server.php
```

So where is the data? The data is transmitted in the HTTP body, using pretty much the same format, but the data is now properly and explicitly transmitted to the server, whereas `GET` more or less uses a "hack" to transfer data, since it actually is only about retrieving documents. `POST` also has the benefit of being able to send non-character data and to circumvent to fairly low size limit for `GET` requests.

Note also that Dojo has special short-hand functions to all the possible HTTP methods; `dojo.xhrGet`, `dojo.xhrPost`, `dojo.xhrPut`, and `dojo.xhrDelete` are all present in some variants.

So, `POST` is the more "proper" way to send data from the client to the server, but at the end of the day the server is usually happy just to get the data at all. Traditionally, `POST` was used to send form data to the server, and still does, if you use the classical form tag together with the `action=""` attribute.

Here is an example of a simple client page and server which send a form using `dojo.xhr`:

```
<html>
   <head>
      <title> Chapter 4 XHR POST test</title>
      <style type="text/css">
      @import "http://o.aolcdn.com/dojo/1.1.0/dojo/
               resources/dojo.css";
      @import "http://o.aolcdn.com/dojo/1.1.0/dijit/themes/dijit.css";
     @import "http://o.aolcdn.com/dojo/1.1.0/dijit/themes/
              tundra/tundra.css";
      </style>
      <script>
      var djConfig=
      {
          isDebug: true,
          parseOnLoad: true
      };
      </script>
      <script type="text/javascript" src="http://o.aolcdn.com/
                   dojo/1.1.0/dojo/dojo.xd.js">
      </script>
      <script language="JavaScript" type="text/javascript">
         dojo.require("dojo.parser");
         dojo.require("dijit.form.Button");
         function _onClick()
         {
            console.log("Sending data to server");
            dojo.xhrPost(
            {
               url:"server2_ch4.php",
               handleAs: "text",
               load: function(data, ioArgs)
               {
                  var ta = dojo.byId('ta');
                  ta.value = ta.value + data;
               },
               form: 'theform'
            });
         }
      </script>
```

```
    </head>
    <body>
       <form id="theform" style="width:300px;">
          <label for="name" style="float:left;">Name:</label><input
           type="text" id="name" name="name" style="float:right;"><br/>
          <label for="age" style="float:left;">Age:</label><input
           type="text" id="age" name="age" style="float:right;"><br/>
          <label for="shoesize" style="float:left;">Shoe Size:
          </label><input type="text" id="shoesize" name="shoesize"
           style="float:right;"><br/>
          <label for="weight" style="float:left;">Weight:</label>
          <input type="text" id="weight" name="weight"
           style="float:right;"><br/>
       </form>
       <div dojoType="dijit.form.Button" id="button"
        onClick="_onClick();">Send</div><br/>
       <textarea id="ta" cols="40" rows="10"></textarea>
    </body>
</html>
```

This time, the styling is a bit improved to get proper rows of labels and input fields. You can also see the use of the `form` parameter, which takes the id of a form element whose fields are to be serialized and sent to the server.

Due to some bugs in the 1.1 release, this example uses the specialized function `dojo.xhrPost`, which in all other aspects other then not having to define which method to use, takes the same parameters as `dojo.xhr`.

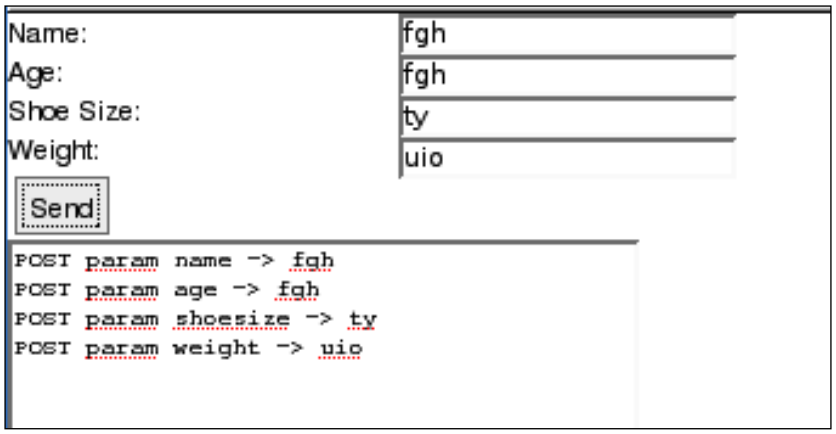
Here is the PHP server that handles the `POST` from our Dojo client:

```
<?php
   $log = "";
   $logstart = getcwd();

   function logit($it)
   {
      //---------------------------------------------------------------
      global $log;
      global $logstart;
      $log = $log.$it;
      $fp2 = fopen($logstart."/meh.log", "a");
      $d = date("Y-m-d H:i:s");
      fwrite($fp2, $d.">".$it."\n");
      fclose($fp2);
      //---------------------------------------------------------------
   }
```

```
        logit("Server reached");
        foreach($_GET as $key =>$value)
        {
            $x = "GET param $key -> $value\n";
           logit($x);
           echo($x);
               }
        foreach($_POST as $key =>$value)
        {
            $x = "POST param $key -> $value\n";
           logit($x);
           echo($x);
               }
    ?>
```

The PHP script enumerates all `GET` and `POST` data sent to it, and logs and echoes back those it finds. The result of using the script can be seen in the following figure:



The content parameter

Sometimes, you need to send `POST` data to a service of some kind, but you don't have a form to pluck the data from. Dojo provides a very useful parameter to the call called `content`. The argument is an object literal with a list of name and value pairs like this:

```
{name: 'Foo Foosson', age: '4711'}
```

It can be used in conjunction with an existing `form` parameter or (possibly more common) stand-alone.

Content-types

In the above examples, the content-type has been set to `text` because that is what we have been passing back and forth between Dojo client and PHP. The block of code in the file `xhr.js` that manages content-types is a very lucid read, and explains a lot of stuff up front.

```
dojo._contentHandlers = {
      "text": function(xhr){ return xhr.responseText; },
      "json": function(xhr){
         if(!dojo.config.usePlainJson)
         {
            console.warn("Consider using mimetype:text/json-comment-
            filtered" + " to avoid potential security issues with
            JSON endpoints" + " (use djConfig.usePlainJson=true to
            turn off this message)");
         }
         return _d.fromJson(xhr.responseText);
      },
      "json-comment-filtered": function(xhr)
         {
         // NOTE: we provide the json-comment-filtered option as one
         //solution to the "JavaScript Hijacking" issue noted by
         //Fortify and others. It is not appropriate for all
         //circumstances.
         var value = xhr.responseText;
         var cStartIdx = value.indexOf("\/*");
         var cEndIdx = value.lastIndexOf("*\/");
         if(cStartIdx == -1 || cEndIdx == -1)
         {
            throw new Error("JSON was not comment filtered");
         }
         return _d.fromJson(value.substring(cStartIdx+2, cEndIdx));
      },
      "javascript": function(xhr){
         // FIXME: try Moz and IE specific eval variants?
         return _d.eval(xhr.responseText);
      },
      "xml": function(xhr){
         var result = xhr.responseXML;
         if(_d.isIE && (!result || window.location.protocol ==
            "file:"))
         {
            _d.forEach(["MSXML2", "Microsoft", "MSXML", "MSXML3"],
            function(prefix){
               try{
```

```
                    var dom = new ActiveXObject(prefix + ".XMLDOM");
                    dom.async = false;
                    dom.loadXML(xhr.responseText);
                    result = dom;
                 }catch(e){ /* Not available. Squelch and try next
                  one. */ }
              });
           }
           return result; // DOMDocument
        }
    };
    dojo._contentHandlers["json-comment-optional"] = function(xhr){
        var handlers = _d._contentHandlers;
        try{
           return handlers["json-comment-filtered"](xhr);
        }catch(e){
           return handlers["json"](xhr);
        }
    };
```

We see that the `text` content-type is associated with a function which just returns the raw text passed to it, nothing strange there.

The next type is called `json`, after the JavaScript Object Notation, which means that the raw data gets translated to a JavaScript object, which is then returned. This is very powerful because it lets the server create arbitrarily complex JavaScript objects, functions, and all, as return values of queries from the Dojo client.

Allowing a client to request data indiscriminately from the server leads to the possibility of cross-site scripting vulnerabilities, unless you take additional precautions. Therefore, Dojo encourages developers to use comment-hardened json, which is called `json-comment-filtered`.

The content-type `javascript` is not normally used. It makes Dojo evaluate the returned text as it is. To be on the safe side, I would always recommend using a message-passing paradigm instead.

The `xml` content-type detects any browser irregularities and selects the browser-provided data structure which represents the data as a hierarchical XML DOM object.

It should be noted that sending XML data between client and server is excessively processor-intensive on the client side, and if you are sending larger quantities of data, please consider sending it as plain text or json.

Finally, we have `json-comment-optional`, which tries to use commented json, but falls back to 'standard' json if comments are not present.

Script request/JSONP (dojo.io.script)

One of the most basic tenets of browser security is the same-domain policy, which enforces the rule that a page loaded from one domain may not connect to any other domain. This rule is enforced by all major browsers, so that a malicious script that might have been injected at some point in the server cannot stealthily connect to a remote server and transmit sensitive details about the user (or worse).

There are, in fact, three exceptions to this rule, and those are CSS files, image files (gif, png, jpeg, and so on), and JavaScript files. That's right, it's perfectly legal to load an HTML page from domain x and have it in turn load JavaScript files from domainy.

In fact, without that feature, it wouldn't be possible to load and use AOL's CDN to load Dojo in the examples in this book.

But maybe you know this and have tried to use Dojo's XHR methods to load resources on remote servers which are encoded as JSON like the Yahoo Search API. Dojo does allow that, but provides another (but similar) API for it.

The call to `dojo.io.script.get` can look like this (taken from the unit test):

```
dojo.io.script.get({
         url: "../_base/timeout.php",
         callbackParamName: "callback",
         content: {Foo: 'Bar'},
         timeout: 500,
         handleAs: "json",
         preventCache: true,
         handle: function(response, ioArgs)
         {
            if(response instanceof Error &&
               response.dojoType == "timeout")
           {
               console.debug("FOO OK TEST");
               d.callback(true);
            }
             else{
               console.debug("FOO FAIL TEST");
               d.errback(false);
            }
         }
```

Note that the equivalent of `dojo.xhrGet`'s `load` parameter for callback is called `handle` here, but apart from that, most things are the same. We even have the opportunity to use fake form elements with `content`.

Remote procedure calls

The XHR functions are versatile and very powerful, but can be cumbersome when what you want to do is not to transfer data, but just call a function. As long as the function can be invoked with a HTTP request, XHR is fine for that also, but Dojo's RPC functionality provides a better alternative.

When using `dojox.rpc`, you create an object which has stub functions that take arguments and construct XHR requests behind the scenes when called.

Another good thing about `dojox.rpc` is that it hides the implementation so that it becomes a natural dividing line between different parts of your application.

It might seem a bit roundabout to follow the exact `dojox.rpc` specification just to get a little separation between layers in your code, and that's of course arguable. However, the most common use case for `dojox.rpc` is using a remote procedure call that has already been made for you.

For instance, Dojo contains several ready-to-go services in `dojox.rpc` format, hiding their exact URL building, parameter quirks, and so on. They only expose a number of functions which might be called like any other functions.

Look at these snippets from the yahoo example in the `dojox.rpc.demo` folder in the Dojo source distribution:

```
var mu = dojo.moduleUrl("dojox.rpc.SMDLibrary", "yahoo.smd");
this.yahoo = new dojox.rpc.Service(mu);
...
this.yahoo.webSearch({query: e.target.value }).
addCallback(this, function(results)
         {
            console.debug(results.ResultSet);
            this.results = results.ResultSet;
            this.render();
         });
```

Three things happen here:

- A file is referenced using `dojo.moduleUrl`, which is a way to reference a resource relative to where `dojo.js` was loaded from. We might as well have used a URL string, but using `dojo.moduleUrl` makes it more portable.
- A new `dojox.rpc`.service object is created, using the file referenced.
- The `dojox.rpc` object (`this.yahoo`) exposes a function called `webSearch` which takes an object literal (`{}`) containing function parameters and a callback function to call when the service has replied, as arguments.

The first point is a bit odd. What's in this file? Is it needed? Yes and No. You can create `dojox.rpc` objects from scratch, but to separate things further, Dojo utilizes a JSON file format called SMD. SMD stands for **Simple Method Description**.

A SMD file is a bit like a WSDL file (**Web Services Description Language**), which defines Web Services, only readable.

It contains a header with general options, followed by a list of what functions to expose and which URLs and arguments they're determined by.

The entry for `webSearch` in the file `dojox/rpc/SMDlibrary` looks like this:

```
webSearch: {
         target: "http://search.yahooapis.com/WebSearchService/V1/
webSearch",
         parameters: [
            { name: "query", type: "string", optional: false,
              "default": "" }, // must be less than 1kb
            { name: "region", type: "string", optional: true,
              "default": "us" },
            { name: "type", type: "string", optional: true, "default":
              "any" }, // can be "all", "any", "phrase"
            { name: "results", type: "integer", optional: true,
              "default": 10 }, // max 100
            { name: "start", type: "integer", optional: true,
              "default": 1 },
            { name: "format", type: "string", optional: true,
              "default": "any" }, // can be "any", "html", "msword",
            //"pdf", "ppt", "rss", "txt", "xls"
            { name: "adult_ok", type: "boolean", optional: true,
              "default": null },
            { name: "similar_ok", type: "boolean", optional: true,
              "default": null },
            { name: "language", type: "string", optional: true,
              "default": null },
            { name: "country", type: "string", optional: true,
              "default": null },
            { name: "site", type: "string", optional: true,
              "default": null },
            { name: "subscription", type: "string", optional: true,
              "default": null },
            { name: "license", type: "string", optional: true,
              "default": "any" } // can be "any", "cc_any",
            //"cc_commercial", "cc_modifiable"
         ]
      },
```

Note that all parameters are not only named, but also types with possible default values. This is an important first step in providing tool and IDE support for Dojo, but also makes the SMD portable for use in other non-JavaScript systems.

If you look at the file in its entirety, you'll also see functions named `trafficData`, `artistSearch`, and many more. In fact, that SMD file is a `dojox.rpc` implementation of all Yahoo's public web services. Similar definitions have been made recently for Google search and several others.

Summary

- Programming with Ajax requires a knowledge of event-driven programming.
- Dojo provides APIs for all HTTP methods: `GET`, `POST`, `PUT`, and `DELETE`.
- It is possible to send all the values of a form in a Dojo Ajax call by defining the DOM id of the form element, using the `form` property in a `dojo.xhr` call.
- It is also possible to 'fake' form postings and insert arbitrary data into a `POST` Ajax call.
- Dojo also has a JSON-RPC remote procedure call API, which lets you define any HTTP URL as a function.
- Dojo's RPC API can be encoded in a SMD file, which can be used to create a RPC function.

5
Forms

Forms are an integral part of any web application. Initially used to send an aggregate amount of data from the browser to the server and then cause a new page to be shown or generated from the server, Dojo has a special Form component which makes it possible to manage data transfer and state changes in the page in a more discrete manner. Together with this form component comes a multitude of form elements, some just augmented variants of their HTML element relatives like ComboBox or TextArea, some of a more complicated nature like the HorizontalSlide or NumberSpinner, with more getting added every month.

The Form family of widgets is, of course, very much tied to Ajax communication, both when dynamically getting the content of a ComboBox from the server and when sending the values of all fields inside a form. But as you will see, most components also have good support for different kinds of simple validations and default warning prompts when their content or choices are outside the given bounds.

The Form Dijit

The Form element was introduced in Netscape Navigator 2.0 as an advanced way to send data back to the server. The basic idea was to let the form enclose a number of data-entry fields of various flavors and send them all at the same time.

The Form element has been a useful element and is still used in its standard form by many sites today. As time has passed, though, the Form element has become something of a problem child when used in Ajax-riddled pages. This is because its default use breaks the convenient single-page application mold and forces a page reload upon submission.

Let's take a look at a simple form and highlight some of the most important features, that any Ajax framework needs to subvert or override to ensure a smooth ride.

```
<FORM action="http://foo.bar.com/myhandler.php" method="post">
    <P>
    <LABEL for="firstname">First name: </LABEL>
              <INPUT type="text" id="firstname"><BR>
    <LABEL for="lastname">Last name: </LABEL>
              <INPUT type="text" id="lastname"><BR>
    <LABEL for="email">email: </LABEL>
              <INPUT type="text" id="email"><BR>
    <INPUT type="radio" name="sex" value="Male"> Male<BR>
    <INPUT type="radio" name="sex" value="Female"> Female<BR>
    <INPUT type="submit" value="Send"> <INPUT type="reset">
    </P>
 </FORM>
```

You see here that two of the possible properties of the `Form` tag are `action` and `method`. The `action` attribute contains the URL where the data contained in the controls of the form is to be sent. Usually, or classically, this is a URL which defines a service of some kind on a server capable of consuming the data.

The `method` property defines which HTTP method is to be used when sending the data. The word `get` means that the data can only be printable ASCII, and will be appended to the URL as series of `&` separated key-value pairs, which will themselves be separated from the base URL by a `?`.

The URL generated by the above form using `method="get"` would look like this:

```
http://foo.bar.com/myhandler.php?firstname=lelle&lastname=
mifflund&email=lelle@mifflund.com..
```

If `post` was used, however, the data sent in the controls might be binary (as in the case of file upload) or of any textual encoding, but most importantly the data will not be sent as part of the URL but in a separate form body.

If you're building a single-page application (which is the most reasonable choice when using an Ajax framework like Dojo), you want to avoid lengthly page loads as much as possible.

This means that we must wire the form to not post its data to the server, but to a JavaScript function instead, which can then send the information at its leisure using Ajax.

Dojo has a whole category devoted to form handling and controls, called `dijit.form`. The main component is called `dijit.form.form`. The following functions can be called on a form Dijit:

- `getValues()` - Returns an associative array of all values of the form's controls indexed by name.
- `setValues()` - Does the opposite, taking an associative array as an argument.
- `isValid()` - Calls `isValid()` on each Dojo control and returns `true` if all controls validate their respective criteria.
- `submit()` - Submits the form.
- `execute()` - Not used, but called when the form is submitted, so that it can be overridden either programmatically or by using an extension point in a template.

Let's take a look at a minimal example of using the form Dijit, which also introduces a couple of Dojo control Dijits, which will be defined later.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/
  TR/html4/strict.dtd">
<html>
<head>
   <title>Chapter 5 Form Test</title>
   <style>
      @import "dojo-1.1.0/dojo/resources/dojo.css";
      @import "dojo-1.1.0/dijit/tests/css/dijitTests.css";
   </style>
   <script type="text/javascript" src="dojo-1.1.0/dojo/dojo.js"
      djConfig="isDebug: true, parseOnLoad: true"></script>
   <script type="text/javascript">
      dojo.require("dijit.form.Form");
      dojo.require("dijit.form.CheckBox");
      dojo.require("dijit.form.TextBox");
      dojo.require("dojo.parser");// scan page for widgets and
      //instantiate them
   </script>
   <script>
      dojo.addOnLoad(function()
      {
         dojo.connect(dojo.byId("formtest"), "onsubmit", function(e)
         {
            e.preventDefault();
           var id = e.currentTarget.id;
```

```
            console.log("submit for "+id+" contained values;");
            var formwidget  = dijit.byId(id);
            var values = formwidget.getValues();
            for(var key in values)
            {
               var value = values[key];
               console.log(key+" -> "+value);
            }
         });
      });
   </script>
</head>
<body>
   <form id="formtest" dojoType="dijit.form.Form">
      <input type="text" name="textbox1" value="something"
       dojoType="dijit.form.TextBox" />
      <input type="checkbox" name="checkbox1" value="foo"
       dojotype="dijit.form.CheckBox" checked="checked" value="foo" />
      <input type="submit"/>
   </form>
</body>
</html>
```

This example uses the `dojo.addOnLoad` function which makes sure that all resources have been loaded before executing the function argument.

The function uses `dojo.connect` to connect the `onsubmit` event of the form Dijit to an anonymous event handler. As described in Chapter 2, the argument passed to event handlers by Dojo's event system is the same for all events across all browsers, which can cut production time significantly).

The most common problem when using forms and Ajax together is that the form tends to be POSTed to the server, which in the default case means that the page gets reloaded. If this happens very quickly, it can lead to a lot of confusion.

So the first thing we need to do in our form handler is to prevent the event from going about its normal business by calling `preventDefault()` on the event object.

We do this before we do anything else, since any error in our code might otherwise prevent the statement from being reached and any debug messages will be wiped out.

We could also have used `e.stopPropagation()` or `dojo.stopEvent(e)`. Then we use the `currentTarget` property of the dojo event which refers to the DOM element which generated the event.

You might think that it should be the button element which we clicked, but in fact, the `onsubmit` event is generated by the form itself, so it is the form element that `currentTarget` refers to.

Using the `id` property of `currentTarget`, we use `dijit.byId()` to get the object reference to the Dijit:

```
var formwidget  = dijit.byId(id);
```



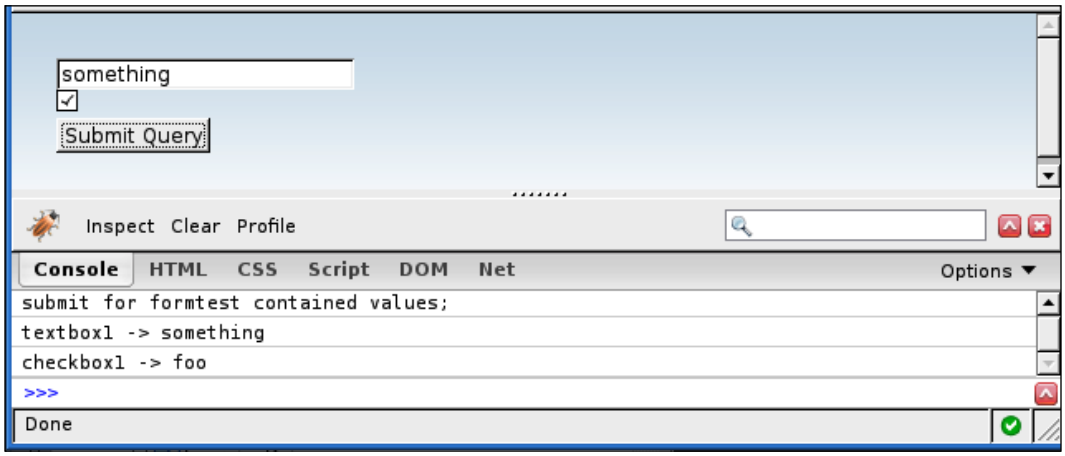
Note that it is easy to intermingle `dojo.byId` and `dijit.byId`. The former gets you the DOM element for the given id, and the latter gets you the Dojo Dijit with the given id.

Dijit ids are just identifiers and need not be the same as the DOM ids of the elements they are contained in, but tend to be the same or similar in everyday use.

Then, we get the list of controls inside the form by calling `getValues()`:

```
var values = formwidget.getValues();
```

Then follows a simple iteration of all keys and values in the returned array, which you can see the result of in the following picture:



Generic properties for all Form Dijits

All Form control Dijits have the following properties:

- `disabled` - If this control shows up as grayed out / disabled.
- `intermediateChanges` - If the control fires `onChange` events all the time or only upon request.
- `tabIndex` - When the user hits the *TAB* key, all controls in the form will be jumped to in the order of this index.

The following functions can be called on each control Dijit:

- `focus()` - Switches focus to this control in the form.
- `getValue()` - Gets the (may be user-entered) value of this control.
- `setValue()` - The opposite function, setting the value.
- `reset()` - Resets the value to the default or initial value.
- `undo()` - Undoes the last value change for the Dijit.
- `SetAttribute()` - Sets specific attributes like `disabled`, `readonly`, and so on.

Textfields and validation

One very common task for JavaScript programmers is to write something quick that validates if what the user entered is really a date or contains a minimum of characters, and so on.

Dojo has a number of special controls just for this purpose, letting you not only check if the entered values comply to your standards but also gives automatic feedback to the user with very little configuration.

The control Dijits also are I18N-aware, so that you can set a locale property for many of them (where it makes sense), so that a `CurrencyTextBox` knows which character is used as a comma-separator for different countries.

The currently available validation-aware Dijits are:

- `CurrencyTextBox`
- `DateTextBox`
- `NumberTextBox`
- `ValidationTextBox`
- `TextBox`

Let's begin by checking out one of the provided test cases in the Dojo source, for validation: `test_validate.html`. This file can be found under `dijit/tests/form` in your Dojo source directory. It contains a multitude of tests and is a very good source for 'snippets' when you're learning.

First, there's the regular `TextBox`, which doesn't do much validation, but is so similar to the others that I include it here anyway. It is used for formatting user input.

```
<input id="q01" type="text" name="firstname" value="testing testing"
   style="width: 700px;" tabIndex=2
   dojoType="dijit.form.TextBox"
   trim="true"
   onfocus="console.log('user onfocus handler')"
   onblur="console.log('user onblur handler')"
   onChange="dojo.byId('oc1').value=arguments[0]"
   intermediateChanges="true"
   propercase="true" />
```

The event handlers `onfocus`, `onblur`, and others are actually defined in the base form widget class for your convenience, so defining handlers for them does what you would expect. Note also the properties `trim` and `propercase`. Here is a sample output of running the Dijit:



This might not seem particularly interesting until I say that the string I actually entered above was " **foobar baz**", with two leading space characters. What the Dijit does is to ensure that the value of the TextBox will always be trimmed of leading and trailing whitespace and that leading characters in words will be capitalized.

Our next example is the `NumberTextBox`, which is very useful for constraining numbers entered by the users. Here is the official example:

```
<div class="testExample">
   <input id="q03" type="text" name="age" tabIndex=1
   dojoType="dijit.form.NumberTextBox"
   promptMessage="(optional) Enter an age between 0 and 120"
   maxLength="3"
   class="small"
   constraints="{places:0,min:0,max:120}"
```

```
        onChange="console.debug('onChange fired for widget id = ' + this.id
        + ' with value = ' + arguments[0]);"
        tooltipPosition="above, below"/>
    </div>
```

Of note here are the properties `maxLength`, `promptMessage`, and `constraints`. The first two are fairly self-explanatory, but the `constraints` property is a rabbit-hole of epic proportions. The property `constraints` can contain a large number of different things, depending on which component it is used in.

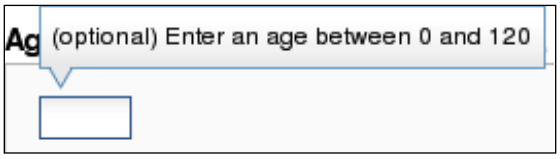
Another property used here that might come in handy is the `tooltipPosition` property, which either sets any tooltips to show above or below the widget (depending on what other things are shown in the vicinity), or before or after.

The number formatting rules are defined in the file `dojo/number.js`, which has the following constraints in addition to `max` and `min`:

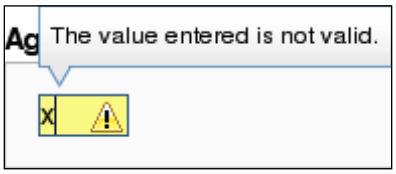
```
//pattern: String?
//override [formatting pattern](http://www.unicode.org/
//reports/tr35/#Number_Format_Patterns)
//with this string
//type: String?
//choose a format type based on the locale from the following:
//decimal, scientific, percent, currency. decimal by default.
//places: Number?
//fixed number of decimal places to show.  This overrides any
//information in the provided pattern.
//round: Number?
//5 rounds to nearest .5; 0 rounds to nearest whole (default). -1
//means don't round.
//currency: String?
//an [ISO4217](http://en.wikipedia.org/wiki/ISO_4217) currency
//code, a three letter sequence like "USD"
//symbol: String?
//localized currency symbol
//locale: String?
//override the locale used to determine formatting rules
```

The constraints can be divided up into `pattern` (the first) and property (the rest) constraints. With a property constraint, you can define the locale to be **USD**, but you could also define a pattern constraint that specified the format of numbers, say, to two decimal places by defining pattern to be `####.##`.

Going back to the current example, this is what happens when the `NumberTextBox` is clicked:



So we see that the `promptMessage` property gives feedback to what the user should write. If we enter something that is not a number, the following will appear:

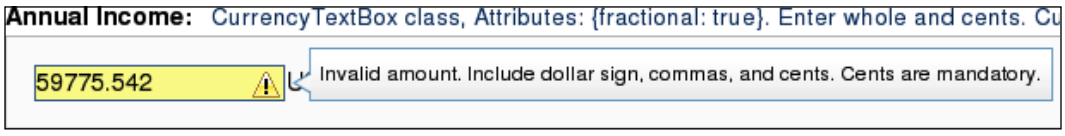


This gives a visible feedback to the user that the value was not valid.

Next up is the `CurrencyTextBox`, which both constrains the values in a similar manner as to those above, and also formats the number entered to the correct currency:

```
<div class="testExample">
   <input id="q08" type="text" name="income1" class="medium"
   value="54775.53"
   dojoType="dijit.form.CurrencyTextBox"
   required="true"
   constraints="{fractional:true}"
   currency="USD"
   onChange="dojo.byId('oc8').value=arguments[0]"
   invalidMessage="Invalid amount.  Include dollar sign, commas, and
   cents.  Cents are mandatory." />USD
    onChange:<input id="oc8" size="15" disabled value="not fired
   yet!" autocomplete="off">
</div>
```

When entering a value with too many decimals, the following is displayed:



Number formatting conforms to the currency defined in the three-letter code for the locale: USD, EUR, JPY, and so on.

For specific information on what formatting rules are available, look in the `dojo/cldr/nls` folder, which has sub-folders for each defined locale, with files for specific rules for currency, gregorian dates, and number formatting. Here is an excerpt from the German number formatting file:

```
({
        'decimal':",",
        'group':".",
        'decimalFormat':"#,##0.###",
        'scientificFormat':"#E0",
        'percentFormat':"#,##0 %",
        'currencyFormat':"#,##0.00 ¤"
})
```

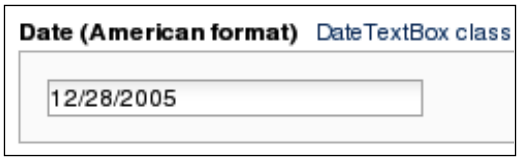
What can be tricky sometimes is that you can define locale for some widgets (for example `CurrencyTextBox`), whereas you define language directly for others, as you will see in the `DateTextBox`.

Dates formatting also differs between locales, and as with the number and currency formatting, you can always choose to hard-code the date format.

The following is an example from the test file for DateTextBox, aptly named `test_DateTextBox.html`:

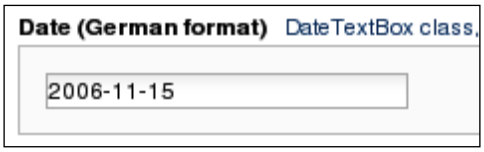
```
<input id="q3" type="text" name="date2" value="2005-12-30"
   dojoType="dijit.form.DateTextBox"
   constraints="{min:'2004-01-01',max:'2006-12-31'}"
   lang="en-us"
   required="true"
   promptMessage="mm/dd/yyyy"
   invalidMessage="Invalid date. Use mm/dd/yyyy format." />
```

Things to note here are that the default value is defined in a standard ISO 8601 format. When displayed it looks like this:



When you click in the textbox, you will get a standard Dojo calendar widget to visually choose the date from, and the date chosen will be converted automatically to the specific format.

In the same test page, there's a German version as well, which sets the `lang` property to `de-de` and displays the following format:



You can also 'manually' specify the date format for the widget in the following way:

```
<input id="q5" name="noDOMvalue" type="text" dojoType="dijit.form.
DateTextBox" constraints="{datePattern:'dd-MM-yyyy', strict:true}">
```

The available locales defined for Dojo are:

- de - German
- de-de - German (Germany)
- en - English
- en-au - English (Australia)
- en-ca - English (Canada)
- en-gb - English (Great Brittain)
- en-us - English (USA)
- es - Spanish
- es-es - Spanish (Spain)
- fr - French
- it - Italian
- it-it - Italian (Italy)
- ja - Japanese
- ja-jp - Japanese (Japan)
- ko - Korean
- ko-kr - Korean (Korea)
- pt - Portuguese
- pt-br - Portuguese (Portugal)
- zh - Chinese
- zh-cn - Chinese (China)
- zh-tw - Chinese (Taiwan)

When working with dates, you often find the need to format and manipulate date-related data outside of widgets. Dojo has a powerful class called `dojo.date` which can help you out.

One particularly nasty thing that raises its head on occasion is when you must show data in a widget for each week, or show a list of dates between two arbitrary other dates that only shows Mondays. And support leap years. And of course, also different locale formatting.

Formatting has been taken care of pretty well already, and if you need to do that inside your code, use the following:

```
dojo.date.locale.format(date, {formatLength:'full',selector:'date',
locale:'en-us'})
```

The `'full'` format returns a string like **Friday, August 11, 2008**, in comparison to the short format which is the more compact form **8/11/08**.

If the locale would have been defined as `de-at` (Austrian German), the function would have returned the string **Freitag, 11. August 2008**.

To work from the other end, you might want to get a `Date` object from a given string describing a date in any language. The function to use then is called `dojo.date.locale.parse`, and is used in the following way:

```
dojo.date.locale.parse("08/11/08", {formatLength:'short',
selector:'date', locale:'en'})
```

Now for the leap-year stuff, `dojo.date` has two functions called `add` and `difference`, which support addition of date values over various date-related boundaries.

What's nice is that you can choose to add not only another date, or a couple of days, but quarters and weeks also. Consider how to add one week to a date:

```
var thedate = new Date(2008, 0, 0);
var newdate = dojo.date.add(thedate, "week", 1));
```

The possible values for the second parameter are:

- `year`
- `quarter`
- `month`
- `week`
- `day`
- `weekday`
- `hour`
- `minute`
- `second`
- `millisecond`

The only odd one here is `weekday`, which (assuming you have a two-day weekend of Saturday and Sunday) counts only days in the week which are not weekends. This can be useful if you want to know the date 20 weekdays from now, skipping over weekends, where people can't (or shouldn't) work.

For a lot of very good examples and inspiration, look at the file `dojo/tests/dates.js`, which has a comprehensive suite of tests of all imaginable combinations of date and time. Here comes a sample, using the test harness function `t.is()` which compares two values:

```
interv = "weekday";
// Sat, Jan 1
dtA = new Date(2000, 0, 1);
// Should be Mon, Jan 3
dtB = new Date(2000, 0, 3);
t.is(dtB, dojo.date.add(dtA, interv, 1));
...
interv = "day";
dtA = new Date(2000, 0, 1);
dtB = new Date(2001, 0, 1);
t.is(dtB, dojo.date.add(dtA, interv, 366));
```

The test harness DOH is defined in Chapter 9, but for the purpose of our current discussion, the `t.is()` function will return **true**, which will mark the test occurred in as passed, so the above code does not produce any actual output.

If you can define your rules for what is considered valid input in a textbox, you can also use the `ValidationTextBox` which takes a `regExp` property, for instance:

```
regExp="[\w]+"
```

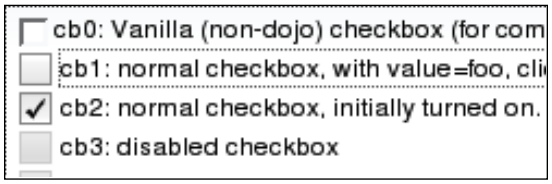
This defines any whitespace character (but nothing else) as valid input.

CheckBoxes, radio buttons, and toggle buttons

No list of form controls would be complete without buttons and checkboxes. Dojo's are styled in the same format as the other Dijits. Here is an example of how to use the checkbox:

```
<input type="checkbox" name="cb1" id="cb1" value="foo"
dojoType="dijit.form.CheckBox" onClick="console.log('clicked cb1')">
```

And here are some examples of how the CheckBox is styled in different configurations (with a leading default HTML textbox for comparison):



The RadioButtons are a bit more stylish, with a nice shading effect on mouseover. They are instantiated the following way, using the standard grouping feature by using the same `name` property:

```
<input type="radio" name="g1" id="g1rb1" value="news" dojoType=
 "dijit.form.RadioButton">
<label for="g1rb1">news</label>
<input type="radio" name="g1" id="g1rb2" value="talk"
 dojoType="dijit.form.RadioButton" checked="checked"/>
<label for="g1rb2">talk</label>
<input type="radio" name="g1" id="g1rb3" value="weather"
 dojoType="dijit.form.RadioButton" disabled="disabled"/>
<label for="g1rb3">weather</label>
```

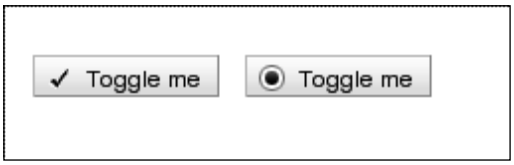
This will look like the following:



ToggleButtons are often what you want when creating a button. Namely, you want to do two things, make the user toggle a state and give feedback on which state is chosen. Here are examples for two types of togglebuttons:

```
<button dojoType="dijit.form.ToggleButton" checked onChange="console.
log('toggled button checked='+arguments[0]);"
iconClass="dijitCheckBoxIcon">
   Toggle me
</button>
<button dojoType="dijit.form.ToggleButton" onChange="console.
 log('toggled button checked='+arguments[0]);" iconClass=
 "dijitRadioIcon">
   Toggle me
</button>
```

With the following looks:



The markup is very forward, using Dojo's own markup classes to allow for mark imagery when state is set.

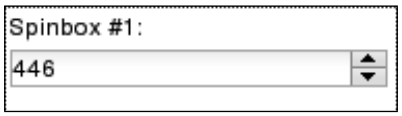
Spinners, sliders, and FilteringSelects

A spinner is a widget that lets the user enter numerical values either by typing or by clicking up or down "spin-wheel" arrows.

As with any form control, the values entered can be constrained to positions, decimal places, and so on. An example can look like this:

```
<input dojoType="dijit.form.NumberSpinner"
   onChange="dojo.byId('oc1').value=arguments[0]"
   value="900"
   class="bigFont"
   constraints="{max:1550,places:0}"
   name="integerspinner1"
   id="integerspinner1">
```

Picture of spinner:



By defining the property `smallDelta`, you can specify how many steps each click on an arrow will affect the value, in either direction. If you define `smallDelta="10"`, each click will increase or reduce the value with 10, correspondingly.

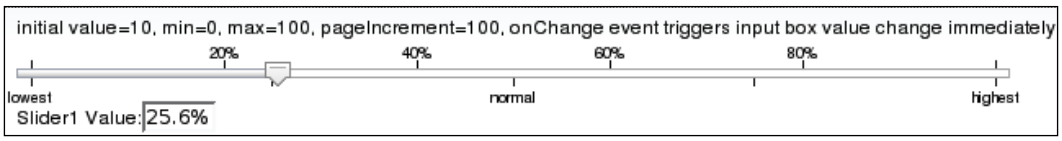
Note also that you can use the spinner for fractional values.

Now for the Slider widget, it is used to select values (or anything, really) by dragging an analog-feeling button along a vertical or horizontal slider. Its instantiation is a bit complex, so it is best to move back and forth between the markup and the visual result a couple of times when trying to understand how it works.

Here is the markup:

```
<div dojoType="dijit.form.HorizontalSlider" name="horizontal1"
   onChange="dojo.byId('slider1input').value=dojo.number.
   format(arguments[0]/100,{places:1,pattern:'#%'});"
   value="10"
   maximum="100"
   minimum="0"
   pageIncrement="100"
   showButtons="false"
   intermediateChanges="true"
   style="width:50%; height: 20px;"
   id="slider1">
<ol dojoType="dijit.form.HorizontalRuleLabels" container="
topDecoration" style="height:1.2em;font-size:75%;" count="6"
numericMargin="1"></ol>
<div dojoType="dijit.form.HorizontalRule" container="topDecoration"
count=6 style="height:5px;"></div>
<div dojoType="dijit.form.HorizontalRule" container="bottomDecoration"
count=5 style="height:5px;"></div>
<ol dojoType="dijit.form.HorizontalRuleLabels" container="bottomDecora
tion" style="height:1em;font-size:75%;">
      <li>lowest</li>
      <li>normal</li>
      <li>highest</li>
      </ol>
</div>
```

And here's how it looks:



You see that the widget is instantiated in a classical way, with properties like value, maximum, and minimum. What is new here is that the widget wraps a number of support widgets which help format the looks of the slide.

The first formatting Dijit you see is the HorizontalRule, which has its Vertical counterpart. It creates marks along the line that the slider defines. The property `count` defines how many marks or grooves are displayed and the property `container` defines if the marks are display above (topDecoration) or below (bottomDecoration).

Also, there are two HorizontalRuleLabels defined, one above and one below, which writes percentages by default, or formats a `<li>` list of string values which will be used instead.

In the above example, six groves are drawn above the slide, and five below. Also, two for decoration.

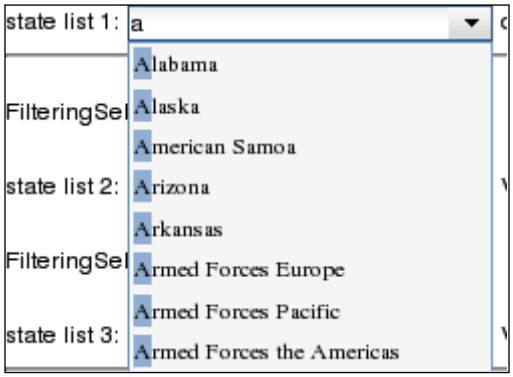
The slider widget also listens for clicks along the slider, and animates movement of the slider button to the point clicked.

`FilteringSelects` look like normal select boxes with drop-down boxes, but accept text input which acts as a filter on the drop-down list.

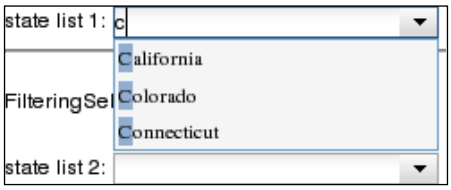
Here is an example markup:

```
<select dojoType="dijit.form.FilteringSelect"
   id="setvaluetest2"
   name="state1"
   autoComplete="false"
   invalidMessage="Invalid state name"
   onChange="dojo.byId('oc1').value=arguments[0]"
   onFocus="console.log('user focus handler')"
   onBlur="console.log('user blur handler')"
   required="false"
   >
   <option value="AL">Alabama</option>
   <option value="AK">Alaska</option>
   <option value="AS">American Samoa</option>
...
...
</select>
```

This means that if the provided list contains a lot of values, and the user clicks the textbox and enters **a**, the available options in the list will be filtered to those only beginning with an **a**:



If the user removes the **a**, and instead enters **c**, the option list will change to this:



The `FilteringSelect` widget is derived from the Dojo `ComboBox`, and as such, can use a `dojo.data store`, which can load the select items dynamically from a server or file.

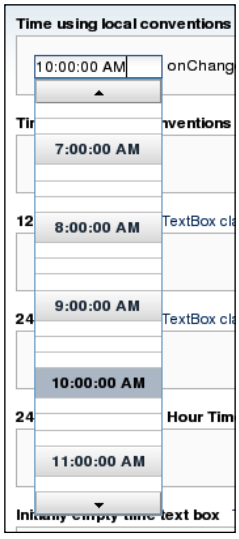
TimeTextBox

Another useful control for browsing over and inputing time values is the TimeTextBox.

The `TimeTextBox` is used in a similar manner to the `DateTextBox`:

```
<input id="q1" type="text" name="time1" value="T17:45:00"
    dojoType="dijit.form.TimeTextBox"
    constraints="{formatLength:'medium'}"
    required="true"
    onChange="dojo.byId('oc1').value=arguments[0]"
    invalidMessage="Invalid time." />
```

The same kind of constraints are used here as well, including patterns. The above example looks like this:



The chooser is implemented as a sliding row of values, which slide by using an animation on hover or mouseover.

Buttons

Classically, the HTML buttons have normal button looks, which are hard to change, and which only display a text caption. Dojo allows for a variety of different buttons, of which many aren't really buttons at all.

The `DropDownButton` lets you simulate a simple drop-down menu, like a ComboBox, but containing any kind of elements, including other widgets.

```
<div dojoType="dijit.form.DropDownButton" iconClass="noteIcon">
   <span>Color</span>
   <div dojoType="dijit.ColorPalette" id="colorPalette"
    style="display: none" palette="3x4"
    onChange="console.log(this.value);"></div>
</div>
```

Using the Dojo icon classes, the button looks like this when normal:



And like this when clicked:

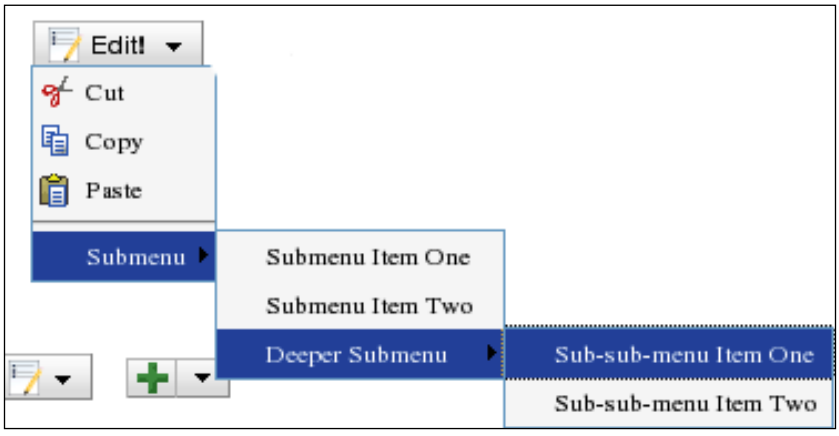


An even more professional looking drop-down button is the following example (as the previous from the unit test in `dijit/test/form/test_Button.html`):

```
<div dojoType="dijit.form.DropDownButton" iconClass="noteIcon">
<span>Edit<b>!</b></span>
<div dojoType="dijit.Menu">
<div dojoType="dijit.MenuItem"  iconClass="dijitEditorIcon
dijitEditorIconCut"  onClick="console.log('not actually cutting
anything, just a test!')">Cut</div>
```

```
<div dojoType="dijit.MenuItem"  iconClass="dijitEditorIcon
dijitEditorIconCopy"  onClick="console.log('not actually copying
anything, just a test!')">Copy</div>
<div dojoType="dijit.MenuItem"  iconClass="dijitEditorIcon
dijitEditorIconPaste"  onClick="console.log('not actually pasting
anything, just a test!')">Paste</div>
<div dojoType="dijit.MenuSeparator"></div>
<div dojoType="dijit.PopupMenuItem">
<span>Submenu</span>
<div dojoType="dijit.Menu" id="submenu2">
<div dojoType="dijit.MenuItem" onClick="console.log('Submenu
1!')">Submenu Item One</div>
<div dojoType="dijit.MenuItem" onClick="console.log('Submenu
2!')">Submenu Item Two</div>
<div dojoType="dijit.PopupMenuItem">
<span>Deeper Submenu</span>
<div dojoType="dijit.Menu" id="submenu4">
<div dojoType="dijit.MenuItem" onClick="console.log('Sub-submenu
1!')">Sub-sub-menu Item One</div>
<div dojoType="dijit.MenuItem" onClick="console.log('Sub-submenu
2!')">Sub-sub-menu Item Two</div>
                    </div>
                </div>
            </div>
        </div>
    </div>
</div>
```

As you can see, we're still using the same `DropDownButton` widget, but enclose a Menu inside it, whose menu items contain Menus themselves. This is what it looks like when expanded:



Ajax integration

One of the most convenient features of Dojo, when it comes to forms, is the possibility of sending the contents of all form controls, be they currency boxes or sliders, with Ajax to the server.

Naturally, you may use Dojo forms the classical way, since all form controls are implemented as classical input elements, with corresponding values. Should you choose (which I very much would recommend, to escape the vicious cycle of visible page reloads) to use Ajax, the format is very compact. Consider the following snippet:

```
dojo.xhrPost(
{
    url: "myservice.php" ,
    form: "theform" ,
    load: function()
    {
          //Things went well
    },
    error: function()
    {
          //Not so good
    }
});
```

Where `theform` is the DOM id of your form. The form will send the values of all its controls as name-value pairs, to the server.

In Chapter 4, you also saw a small PHP server, which can be used for all the examples in this chapter as well, if you want something small to try out your ideas on.

Looking back to the basic function of the form Dijit, you might remember that it had two functions called `setValues()` and `getValues()`.

We've already touched upon using `getValues` to get all the values of a form, regardless of the type of Dijit control we're using.

But even more interesting is that we can request data from the server in a key-value format and populate our form with default or 'latest saved' values. This streamlines the editing process for us when our app consists of many forms to be saved and loaded again and again by the user.

Summary

- Dojo has many widgets that simplify the building of HTML forms.
- All form widgets are derived from a common base class, which gives them similar behavior.
- The form can be submitted in a simple manner, regardless of the internal complexity of the widgets that make up the form.
- Many form widgets support populating themselves with selectable data from the server.
- Many widgets support complex validation rules and can signal to the user when the input deviates from them.
- Many widgets can format the data presented in a locale-dependent way, so that currency and dates are displayed according to the user's current locale.
- Using Dojo's date functions, it is possible to add or subtract months, weeks and years, letting Dojo handle leap-years, different amount of days in a month, and so on.

6
Layout

The layout widgets in Dojo can often be alternatives to complicated CSS rules. Many times it is necessary to create pages with a two- or three-column layout, sometimes with a header and/or footer as well.

The layout manager widgets let you create powerful layouts without using more than a minimum of custom CSS styling. The layout managers have a lot of carefully laid-out styling that you can benefit from without having to interact with it. The BorderContainer makes sure that it has a proper margin to the widget it contains, as well as adding (removable) 'gutter' border lines for default clarity, for example. The TabContainer lets you programmatically choose if you want the tabs on top, bottom, left, or right without using any custom CSS, and so on.

This chapter will list a large number of layout widgets. Some are fully 'official', with accessibility support and internationalization. By the time of the writing this book, some will be experimental, and are found under the `dojox` directory.

Nevertheless, this book's intent is to give you a smorgasboard of widgets, so that you know what you have available. One of the biggest challenges in learning Dojo is the sheer volume of the framework. In previous chapters, there were lists with short descriptions of useful Dijit widgets. The layout managers are too important and too complicated to be reduced to a mere sentence or two, which explains the volume of this chapter.

After the near-exhaustive expose of different layout managers and supporting acts, there will be a couple of longer examples which, with some luck, will give you some good practical points to elaborate upon.

Basic Dojo layout facts

The Layout widgets in Dojo are varied in nature, but their most common use is as 'windows' or areas which organize and present other widgets or information. Several use the same kind of child elements the ContentPane.

The ContentPane is a widget which can contain other widgets and plain HTML, reload content using Ajax and so on. The ContentPane can also be used stand-alone in a page, but is more usable inside a layout container of some sort.

And what is a layout container? Well, it's a widget which contains ContentPanes, of course. A layout container can often contain other widgets as well, but most containers work very well with a different configuration of ContentPanes, which properly insulates the further contents.

Take the TabContainer, for example. It is used to organize two or more ContentPanes, where each gets its own tab. When a user clicks on one of the tabs, the ContentPane inside it is shown and all others are hidden.

Using BorderManager can bring the necessary CSS styling down to a minimum, while giving a simple interface for managing dynamic changes of child widgets and elements.

ContentPane

A ContentPane can look like anything of course, so it doesn't really help putting a screen-dump of one on the page. However, the interface is very good to know.

The following arguments are detected by ContentPane and can be used when creating one either programmatically or by markup:

```
// href: String
//The href of the content that displays now.
//Set this at construction if you want to load data externally
//when the pane is shown.(Set preload=true to load it immediately.)
//Changing href after creation doesn't have any effect;
//see setHref();
href: "",
//extractContent: Boolean
//Extract visible content from inside of <body> .... </body>
  extractContent: false,
//parseOnLoad: Boolean
//parse content and create the widgets, if any
  parseOnLoad:true,
//preventCache: Boolean
```

```
    //Cache content retreived externally
      preventCache:false,
    //preload: Boolean
    //Force load of data even if pane is hidden.
      preload: false,
    //refreshOnShow: Boolean
    //Refresh (re-download) content when pane goes from hidden to shown
      refreshOnShow: false,
    //loadingMessage: String
    //Message that shows while downloading
    loadingMessage: "<span class='dijitContentPaneLoading'>$
    {loadingState}</span>",
    //errorMessage: String
    //Message that shows if an error occurs
    errorMessage: "<span class='dijitContentPaneError'>${errorState}
</span>",
```

You don't need any of those, of course. A simple way to create a ContentPane would be:

```
var pane = new dojo.layout.ContentPane({});
```

And a more common example would be the following:

```
var panediv = dojo.byId('panediv');
var pane = new dojo.layout.ContentPane({ href: "/foo/content.html",
preload: true}, panediv);
```

where we would have an element already in the page with the id `'panediv'`.

As you see, there are also a couple of properties that manage caching and parsing of contents. At times, you want your ContentPane to parse and render any content inside it (if it contains other widgets), whereas other times you might not (if it contains a source code listing, for instance).

You will see additional properties being passed in the creation of a ContentPane which are not part of the ContentPane itself, but are properties that give information specific to the surrounding Container. For example, the TabContainer wants to know which tab this is, and so on.

Container functions

All container widgets arrange other widgets, and so have a lot of common functionality defined in the `dijit._Container` class. The following functions are provided for all Container widgets:

- `addChild`: Adds a child widget to the container.
- `removeChild`: Removes a child widget from the container.
- `destroyDescendants`: Iterates over all children, calling destroy on each.
- `getChildren`: Returns an array containing references to all children.
- `hasChildren`: Returns a boolean.

LayoutContainer

The LayoutContainer is a widget which lays out children widgets according to one of five alignments: right, left, top, bottom, or client. Client means "whatever is left", basically.

The widgets being organized need not be ContentPanes, but this is normally the case. Each widget then gets to set a `layoutAlign` property, like this:

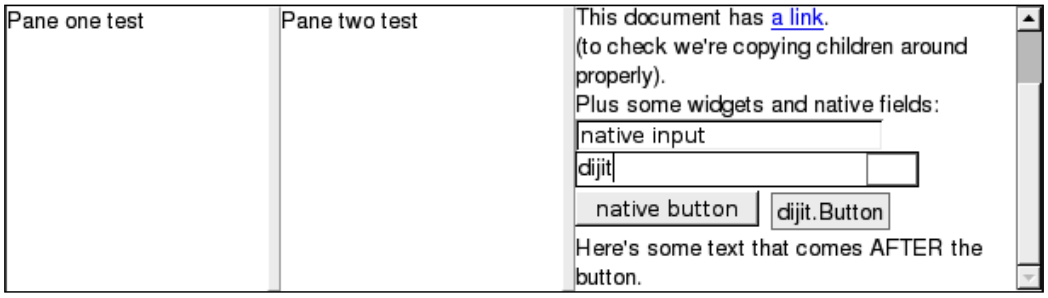
```
layoutAlign = "left".
```

The normal way to use LayoutContainer is to define it using markup in the page, and then define the widgets to be laid out inside it.

LayoutContainer has been superceeded by BorderContainer, and will be removed in Dojo version 2.0.

SplitContainer

The SplitContainer creates a horizontal or vertical split bar between two or more child widgets.



A markup declaration of a SplitContainer can look like this:

```
<div dojoType="dijit.layout.SplitContainer"
     orientation="vertical"
     sizerWidth="7"
     activeSizing="false"
     style="border: 1px solid #bfbfbf; float: left;
     margin-right: 30px;  width: 400px; height: 300px;">
```

The SplitContainer must have a defined height and width. The `orientation` property is self-explanatory, as is `sizerWidth`. The property `activeSizing` means, if set to `true`, that the child widgets will be continually resized when the user changes the position of the sizer.

This can be bad if the child widgets are complex or access remote information to render themselves, in which case the setting can be set to `false`, as in the above example. Then the resize event will only be sent to the child widgets when the user stops.

Each child widget needs to define the `sizeMin` and `sizeShare` attributes. The `sizeMin` attribute defines the minimum size for the widget in pixels, but the `sizeShare` attribute is a relative value for the share of space this widget takes in relation to the other widget's `sizeShare` values.

If we have three widgets inside the SplitPane with `sizeShare` values of 10, 40 and 50, they will have the same ratios in size as if the values had been 1:4:5.

StackContainer

The StackContainer hides all children widgets but only one at any given time, and is one of the base classes for both the Accordion and TabContainers.

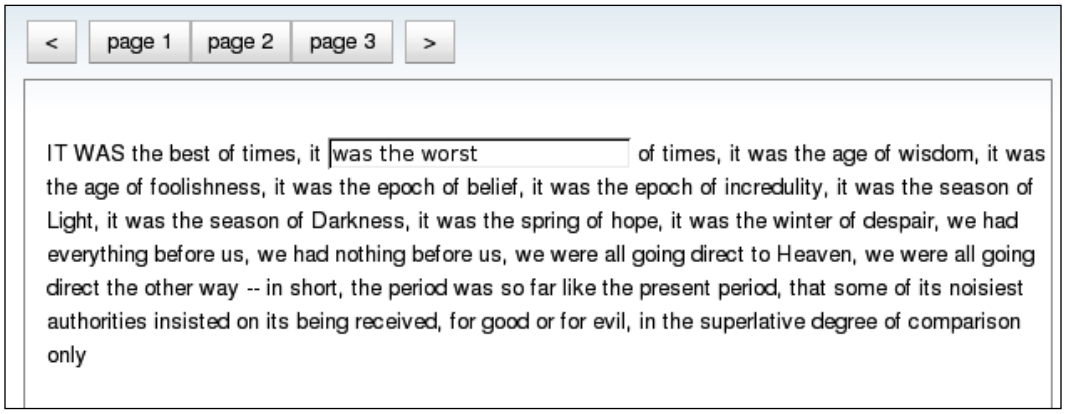
StackContainer exists as a separate widget to allow you to define how and when the child widgets are shown. Maybe you would like to define a special kind of control for changing between child widget views, or maybe you want other events in your application to make the Container show specific widgets.

Either way, the StackContainer is one of the most versatile Containers, along with the BorderContainer.

The following functions are provided for interacting with the StackContainer:

- `back` - Selects and shows the previous child widget.
- `forward` - Selects and shows the next child widget.

- `getNextSibling` - Returns a reference to the next child widget.
- `getPreviousSibling` - Returns a reference to the previous child widget.
- `selectChild` - Takes a reference to the child widget to select and show.
- `closeChild` - If the widget is defined as `closable`, it will present a small **x** icon, which will destroy the widget and remove it. This function does the same programmatically.



Here is the slightly abbreviated markup for the test shown above (`test_StackContainer.html`):

```
<div id="myStackContainer" dojoType="dijit.layout.StackContainer"
      style="width: 90%; border: 1px solid #9b9b9b; height: 20em;
            margin: 0.5em 0 0.5em 0; padding: 0.5em;">
      <p id="page1" dojoType="dijit.layout.ContentPane" title=
       "page 1">IT WAS the best of times, ....</p>
      <p id="page2" dojoType="dijit.layout.ContentPane" title=
       "page 2">There were a king with a large jaw ...</p>
      <p id="page3" dojoType="dijit.layout.ContentPane" title=
       "page 3">It was the year of Our Lord one thousand seven
       hundred and seventy- five. .../p>
</div>
```

The StackContainer also publishes topics on certain events which can be caught using the messaging system. The topics are:

- `[widgetId]-addChild`
- `[widgetId]-removeChild`
- `[widgetId]-selectChild`
- `[widgetId]-containerKeyPress`

Where [widgetId] is the id of this widget. So if you had a StackContainer defined in the following manner:

```
<div id="myStackContainer" dojoType="dijit.layout.StackContainer">
...
</div>
```

You can use the following code to listen to events from your StackContainer:

```
dojo.subscribe("myStackContainer-addChild",  this, function(arg)
{
   var child = arg[0];
   var index = arg[1];
});
```

Compare with the following code from the StackContainer class itself:

```
addChild: function(/*Widget*/ child, /*Integer?*/ insertIndex)
   {
      // summary: Adds a widget to the stack
      this.inherited(arguments);
      if(this._started)
      {
      // in case the tab titles have overflowed from one line
      // to two lines
         this.layout();
         dojo.publish(this.id+"-addChild", [child, insertIndex]);
      // if this is the first child, then select it
         if(!this.selectedChildWidget)
         {
            this.selectChild(child);
         }
      }
   },
```

Also declared in the class file for the StackContainer is the `dijit.layout.StackController`. This is a sample implementation of a separate widget which presents user controls for stepping forward, backward, and so on in the widget stack.

What differentiates this widget from the Tabs in the TabContainer, for example, is that the widget is completely separate and uses the message bus to listen to events from the StackContainer. You can use it as-is, or subclass it as a base for you own controllers.

But naturally, you can build whatever you want and connect the events to the `forward()` and `back()` function on the StackContainer.

It's interesting to note that at the end of the files that define StackContainer, the `_Widget` base class for all widgets is extended in the following way:

```
//These arguments can be specified for the children of a
//StackContainer.
//Since any widget can be specified as a StackContainer child,
//mix them into the base widget class.  (This is a hack, but it's
//effective.)
dojo.extend(dijit._Widget, {
//title: String
//Title of this widget.Used by TabContainer to the name the tab, etc.
title: "",
//selected: Boolean
//Is this child currently selected?
selected: false,
//closable: Boolean
//True if user can close (destroy) this child, such as
//(for example) clicking the X on the tab.
closable: false,   //true if user can close this tab pane
onClose: function(){
//summary: Callback if someone tries to close the child, child
//will be closed if func returns true
   return true;
   }
});
```

This means that all child widgets inside a StackContainer (or Tab or AccordionContainer) can define the above properties, which will be respected and used accordingly. However, since the properties are applied to the `_Widget` superclass they are of course now generic to all widgets, even those not used inside any containers at all.

The most commonly used property is the `closable` property, which adds a close icon to the widget and title, which defines a title for the tab.

A lot of Dijits respond to keypress events, according to WAI rules. Let's look at the code that is responsible for managing key events in StackContainer and all its descendants:

```
onkeypress: function(/*Event*/ e){
   //summary:
   //Handle keystrokes on the page list, for advancing to
     next/previous button
   //and closing the current page if the page is closable.
   if(this.disabled || e.altKey ){ return; }
```

```
var forward = null;
if(e.ctrlKey || !e._djpage){
   var k = dojo.keys;
   switch(e.charOrCode){
      case k.LEFT_ARROW:
      case k.UP_ARROW:
         if(!e._djpage){ forward = false; }
         break;
      case k.PAGE_UP:
         if(e.ctrlKey){ forward = false; }
         break;
      case k.RIGHT_ARROW:
      case k.DOWN_ARROW:
         if(!e._djpage){ forward = true; }
         break;
      case k.PAGE_DOWN:
         if(e.ctrlKey){ forward = true; }
         break;
      case k.DELETE:
         if(this._currentChild.closable)
         {
           this.onCloseButtonClick(this._currentChild);
         }
         dojo.stopEvent(e);
         break;
      default:
         if(e.ctrlKey){
         if(e.charOrCode == k.TAB)
         {
           this.adjacent(!.shiftKey).onClick();
           dojo.stopEvent(e);
         }
         else if(e.charOrCode == "w")
         {
           if(this._currentChild.closable)
           {
             this.onCloseButtonClick(this._currentChild);
           }
         dojo.stopEvent(e); // avoid browser tab closing.
         }
      }
   }
   // handle page navigation
   if(forward !== null)
   {
```

```
            this.adjacent(forward).onClick();
            dojo.stopEvent(e);
        }
    }
},
```

The code is a very good example on how to handle key press events in Dojo in its own right, but for our purposes we can summarize in the following way:

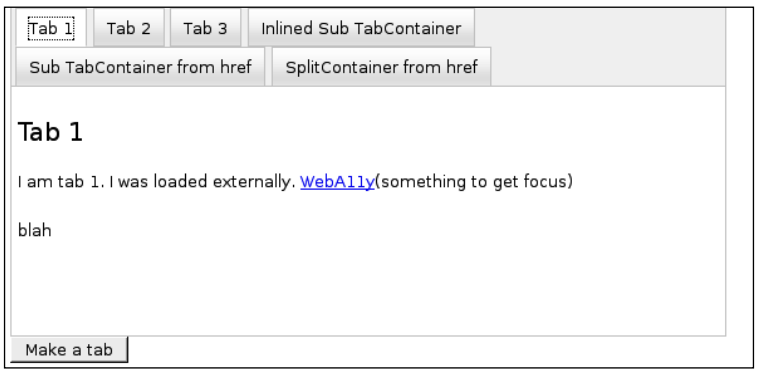
- If *UP*, *LEFT*, or *SHIFT+TAB* is pressed, `forward` is set to `false`, and the last block of code will use that as an argument to the adjacent function which returns the prior child widget if `false` and the next child widget if `true`. In this case, the former.
- If *DOWN* , *RIGHT*, or *TAB* is pressed, `forward` will be set to `true`, which will declare the next child widget to be activated and shown.
- If *DELETE* or *w* is pressed and the current child widget is `closable`, it will be destroyed.

TabContainer

The TabContainer, which derives from StackContainer, organizes all its children into tabs, which are shown one at a time.

As you can see in the picture below, the TabContainer can also manage hierarchical versions of itself.

The TabContainer takes an argument property called `tabPosition`, which controls where the tab icons are displayed for each tab. Possible values are `"top"`, `"bottom"`, `"left-h"`, `"right-h"`, with `"top"` as default.

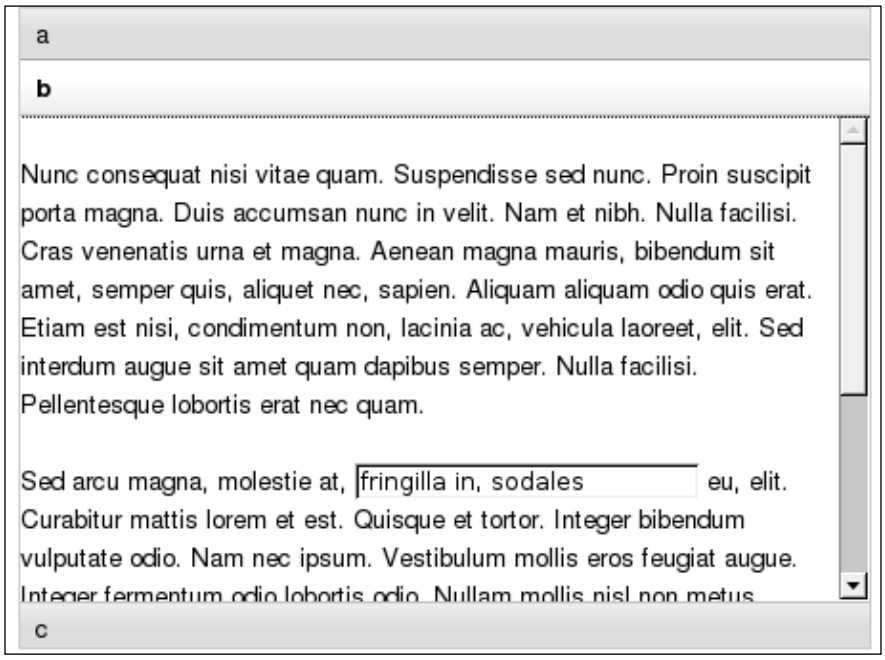


There are no special functions provided for TabContainer, which adds very little logic to that provided from the StackContainer superclass.

AccordionContainer

The AccorionContainer shows a horizontal bar for each added child widget, which represents its collapsed state. The bar acts as a tab and also holds the title defined for the child widget.

When the bar is clicked, an animation hides the current widget, and also animates in the widget whose bar was clicked.



The abbreviated code for the test case above (`test_accordionContainer.html`) is here:

```
<div dojoType="dijit.layout.AccordionContainer"
      style="width: 400px; height: 300px; overflow: hidden">
      <div dojoType="dijit.layout.AccordionPane" title="a">
         Hello World
      </div>
      <div dojoType="dijit.layout.AccordionPane" title="b">
         <p>
            Nunc consequat nisi ...
         </p>
         <p>
            Sed arcu magna...
         </p>
```

```
        </div>
        <div dojoType="dijit.layout.AccordionPane" title="c">
           <p>The quick brown fox jumps over the lazy dog.  The quick
              brown fox jumps over the lazy dog.  The quick brown fox
              jumps over the lazy dog.</p>
        </div>
     </div>
```

A funny thing about the AccordionContainer is that it requires not any old widget as a child node, but its own AccordionPane, as you see in the code above.

However, the AccordionPane has ContentPane as superclass, and defines itself slightly differently due to the special looks of an accordion.

Also, the AccordionPane does not currently support nested Layout widgets, even though single-level widgets are supported.

BorderContainer

The BorderContainer has replaced the functionality of both LayoutContainer and SplitContainer.

Note that the outermost BorderContainer widget does not carry any layout information. This is instead delegated to each individual widget.

As each child gets added to the BorderContainer, the layout is recalculated. Using the BorderContainer is a very good alternative to using CSS-based "tableless tables". For using the BorderContainer, you don't need any other rules, and the Container recalculates positioning automatically, without the need for additional CSS rules (except for the height/width case below) each time you add an element or widget to the area.

Since BorderContainer widget replaces both SplitContainer and LayoutContainer, it both lets its child widgets declare where they are in relation to each other. Optionally, add resizing splitter between children.

Also, instead of optionally declaring one child as "client", one child must now always be declared as "center". For some reason, the child widget now use `region`, instead of `layoutAlign`, so a child widget which would have been defined like this in LayoutContainer:

```
<div dojoType="dijit.layout.ContentPane" layoutAlign="top">...</div>
```

is now defined like this instead:

```
<div dojoType="dijit.layout.ContentPane" region="top">...</div>
```

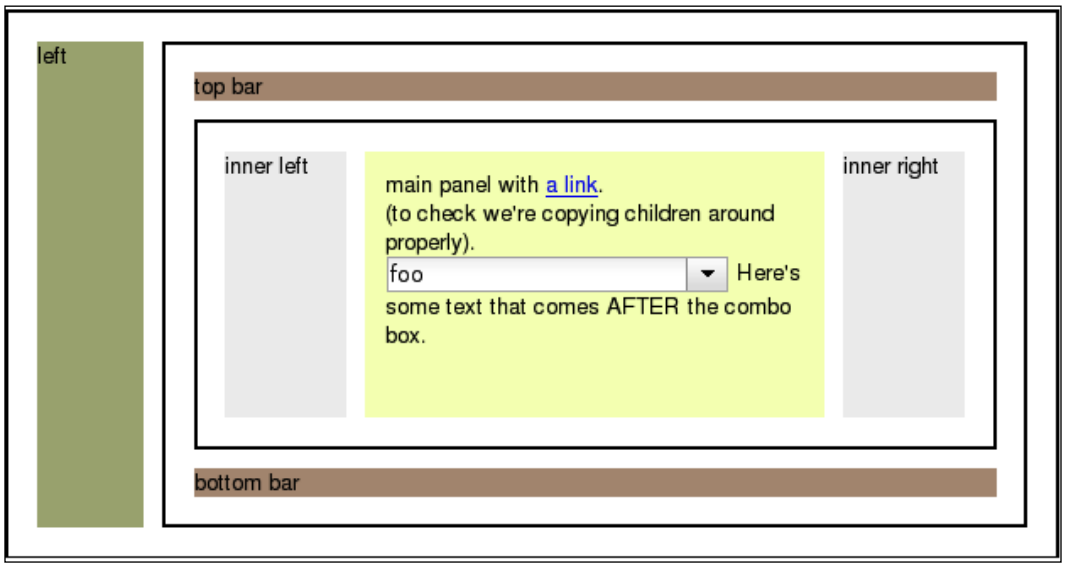
All "side" widgets must define a width, in style, by CSS class or otherwise, and the same applies for top/bottom widgets, but with height. Center widgets must not declare either height or width, since they use whatever is left over from the other widgets.

You can also use `leading` and `trailing` instead of `right` and `left`. The only difference is that when you change locale to a region that has text going from right to left (like Arabic and many others), this will arrange the widgets appropriate to the locale.

The BorderContainer also takes an optional `design` property, which defines if the BorderContainer is a `headline` or `sidebar`. The `headline` is the default and looks like the picture below. `headline` means that the top and bottom widgets extend the full length of the container, whereas `sidebar` means the the right and left (or leading and trailing) widgets extend top to bottom.

The `sizeShare` attribute for the ContentPanes used in the SplitContainer is deprecated in BorderContainer. All ContentPanes sizes are defined using regular techniques (direct stylin, classes, and so on).

From the BorderContainer test located in `dijit/tests/layout/test_BorderContainer_nested.html`, we find the following layout:



The (abbreviated) source code for the example is here:

```
<div dojoType="dijit.layout.BorderContainer"
      style="border: 2px solid black; width: 90%; height: 500px;
      padding: 10px;">
      <div dojoType="dijit.layout.ContentPane" region="left"
       style="background-color: #acb386; width: 100px;">
         left
      </div>
      <div dojoType="dijit.layout.ContentPane" region="right"
       style="background-color: #acb386; width: 100px;">
         right
      </div>
      <div dojoType="dijit.layout.ContentPane" region="top"
       style="background-color: #b39b86; height: 100px;">
         top bar
      </div>
      <div dojoType="dijit.layout.ContentPane" region="bottom"
       style="background-color: #b39b86; height: 100px;">
         bottom bar
      </div>
      <div dojoType="dijit.layout.ContentPane" region="center"
       style="background-color: #f5ffbf; padding: 0px;">
         <div dojoType="dijit.layout.BorderContainer" design="sidebar"
            style="border: 2px solid black; height: 300px;">
            <div dojoType="dijit.layout.ContentPane" region="left"
             style="background-color: #acb386; width: 100px;">
               left
               </div>
            <div dojoType="dijit.layout.ContentPane" region="right"
             style="background-color: #acb386; width: 100px;">
               right
            </div>
            <div dojoType="dijit.layout.ContentPane" region="top"
             style="background-color: #b39b86; height: 100px;">
               top bar
            </div>
            <div dojoType="dijit.layout.ContentPane" region="bottom"
             style="background-color: #b39b86; height: 100px;">
               bottom bar
            </div>
            <div dojoType="dijit.layout.ContentPane" region="center"
             style="background-color: #f5ffbf; padding: 10px;">
               main panel with <a href="http://www.dojotoolkit.org/">
               a link</a>.<br />
               (to check we're copying children around properly).
               <br />
```

```
                <select dojoType="dijit.form.FilteringSelect">
                   <option value="1">foo</option>
                   <option value="2">bar</option>
                   <option value="3">baz</option>
                </select>
                Here's some text that comes AFTER the combo box.
             </div>
          </div>
       </div>
    </div>
```

You see here the recurring theme of using ContentPanes inside Containers. Also, the innermost `"center"` ContentPane wraps a new BorderContainer which has its own internal top/left layout widgets.

Depending on what kind of application you are building, the BorderContainer might be a good starting point. Since you already know that you can change and reload the contents of individual ContentPanes, you are left with a layout in which each element can function as a lightweight Iframe with none of the negative side effects.

DragPane

The DragPane is a very simple idea. You have a very large area of elements to display, and want to let the user 'drag' the underlying pane around using the mouse.

The DragPane can be used in instances where you have a lot of pictures to present. It can also be used to present text or other widgets that are too numerous to fit in your current designated area of screen real estate.

The only property of DragPane is `invert`, which if set to `true`, inverts the axis of the drag of the mouse.

Example:

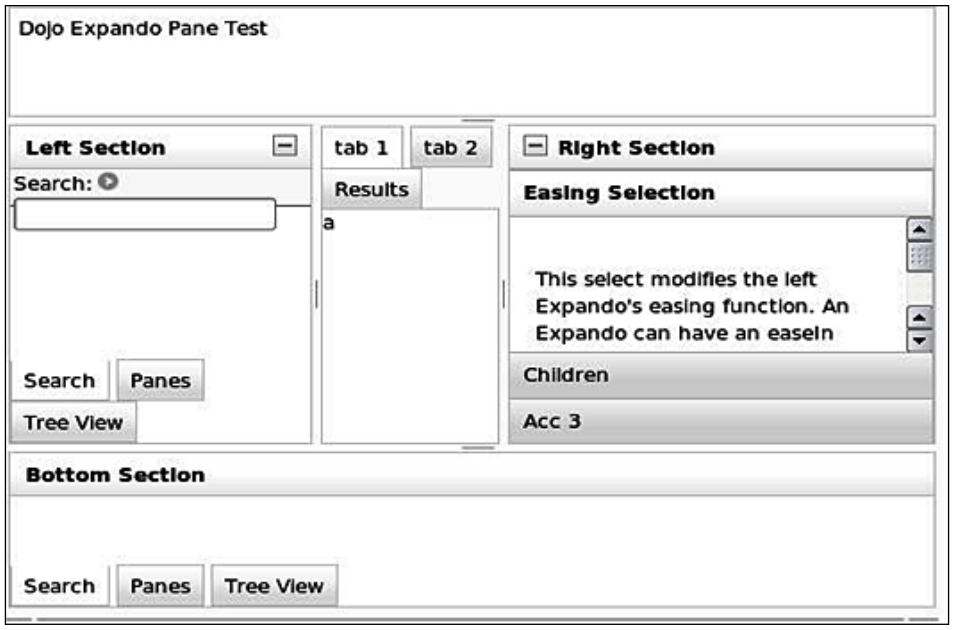
```
<div class="hugetext" id="container" invert="false" dojoType="dojox.
layout.DragPane">
      <p style="color:#666; padding:8px; margin:0;">
      Lorem ipsum dolor sit amet, consectetuer adipiscing elit.
      In porta. Etiam mattis libero nec ante. Nam
      porta lacus eu ligula. Cras mauris. Suspendisse vel augue.
      Vivamus aliquam orci ut eros. Nunc eleifend
      sagittis turpis. purus purus in nibh. Phasellus in nunc.
      </p>
</div>
```

ExpandoPane

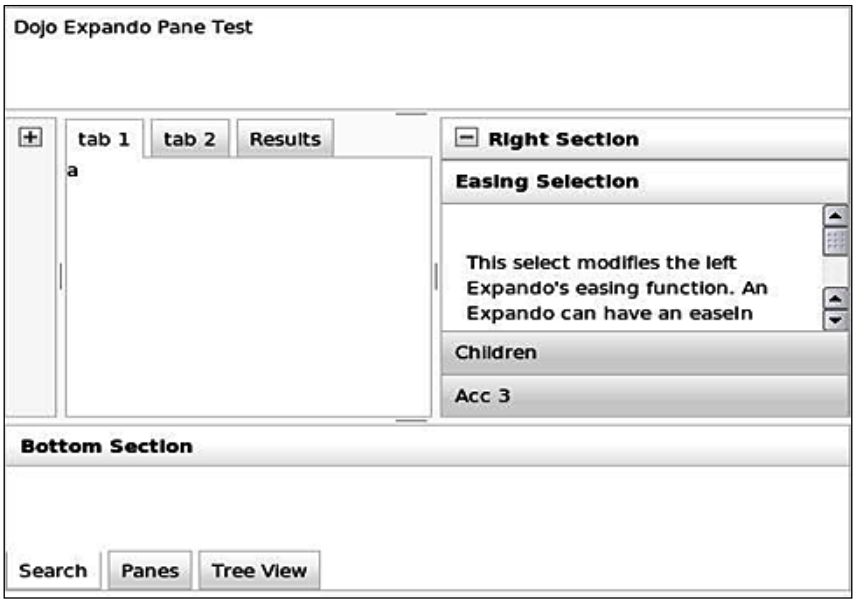
The ExpandoPane is an experimental (in 1.2) dynamic version of the ContentPane for use in BorderContainers.

It can expand or collapse on command, like an AccordionPane, but supports LayoutManagers as children widgets, and can be put anywhere a ContentPane would be inside the BorderContainer. It supports the resize bars that come with the BorderContainer and remembers the position of each after a minimize/maximize animation.

In the example below, the top-level BorderContainer has been populated by a number of normal ContentPanes, and a number of ExpandoPanes. To the right and left are panes which have the tell tale  icons of the ExpandoPanes, which inform the user that they can be minimized.



On pressing the icon of the right pane, the ExpandoPane will animate a minimize action, and come to rest like this:



The ExpandoPane can be declared in the following way:

```
<div dojoType="dojox.layout.ExpandoPane"
     splitter="true"
     duration="125"
     region="left"
     title="Left Section"
     id="leftPane"
     maxWidth="275"
     style="width: 275px;">
</div>
```

The properties you can pass to an ExpandoPane shows some interesting stuff:

```
  maxHeight: "",
 maxWidth: "",
  splitter: "",
  _showing: true,
  //easeOut: String|Function
  //easing function used to hide pane
  easeOut: "dojo._DefaultEasing",
  //easeIn: String|Function
  //easing function use to show pane
  easeIn: "dojo._DefaultEasing",
  //duration: Integer
  //duration to run show/hide animations
  duration: 420,
  startExpanded: true,
```

The `maxHeight`, `maxWidth`, and `splitter` (`true`/`false`) are fairly self-explanatory. The `duration` variable means the duration of the max/min animations.

The fun part here is that Dojo has a very rich animation system that can be used for a variety of purposes. There's a default 'easing' animation which provides a smooth extrapolation between start and stop coordinates.

If you want, you could change the `easeIn` or `easeOut` animation to something quicker or quirkier.

Further down in the code for `ExpandoPane.js`, we see the following:

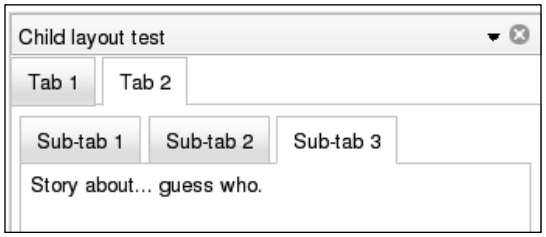
```
if(dojo.isString(this.easeOut))
   {
      this.easeOut = dojo.getObject(this.easeOut);
   }
   if(dojo.isString(this.easeIn))
   {
      this.easeIn = dojo.getObject(this.easeIn);
   }
```

This means that the animations will be left alone if they are not strings. If they are, the objects they represent will be loaded instead.

So what kind of easing animations exist, and how do we roll our own? That will be covered in later chapters, but if you're curious, take a sneak peek at the file `dojo/fx/easing.js`, which contains functions such as `linear`, `quartInOut`, `SineIn`, `elasticOut`, and others.

FloatingPane

The FloatingPane is the obligatory 'window' widget, which can be dragged around like a desktop window and expanded or minimized.



As you can see, it supports complex child widgets, which makes it great for hiding complex layouts in an efficient manner. Not all applications fit into the 'desktop' template, but if you can or want to use that metaphor, the FloatingPane is what you want.

The basic example above shows only a 'close' icon, but more are supported. You can set the following properties when defining a FloatingPane:

```
//closable: Boolean
//Allow closure of this Node
closable: true,
//dockable: Boolean
//Allow minimizing of pane if true
dockable: true,
//resizable: Boolean
//Allow resizing of pane true if true
resizable: false,
//maxable: Boolean
//Horrible param name for "Can you maximize this floating pane?"
maxable: false,
//resizeAxis: String
//One of: x | xy | y to limit pane's sizing direction
resizeAxis: "xy",
//title: String
//Title to use in the header
title: "",
//dockTo: DomNode?
//if empty, will create private layout.Dock that scrolls with
//viewport
//on bottom span of viewport.
dockTo: "",
//duration: Integer
//Time is MS to spend toggling in/out node
duration: 400,
```

Setting `Boolean` to the `closable` property gives rise to the close icon, as `dockable` and `maxable` creates minimize and maximize icons respectively. You also see a `dockTo` property which makes it possible for you to define an element of your own where the pane minimizes and iconifies when the minimize icon is clicked.

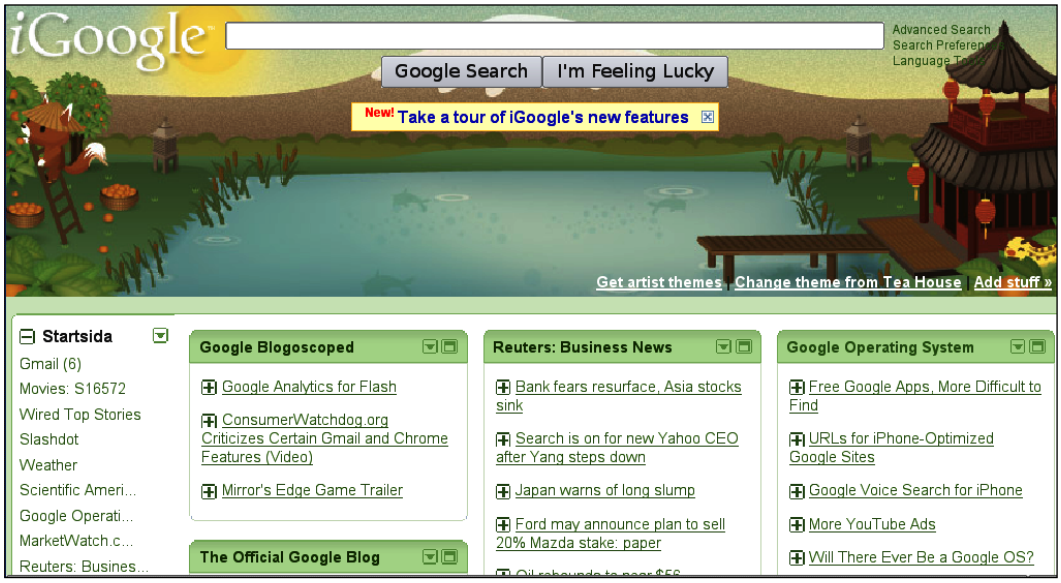
The `resizable` property adds, if `true`, a ResizeHandle to the pane, which is more or less expected of windows in this day and age. And the `resizeAxis`, as the name implies, can restrict resizing to one or none of the two axes.

The `duration` property is, as you can see in the comments, the time spent in in/out animations for docking operations.

GridContainer

There are a lot of sites available that let you add a lot of rss feeds and assorted widgets to a personal page, and which then also let you arrange them by dragging the widgets themselves around the page.

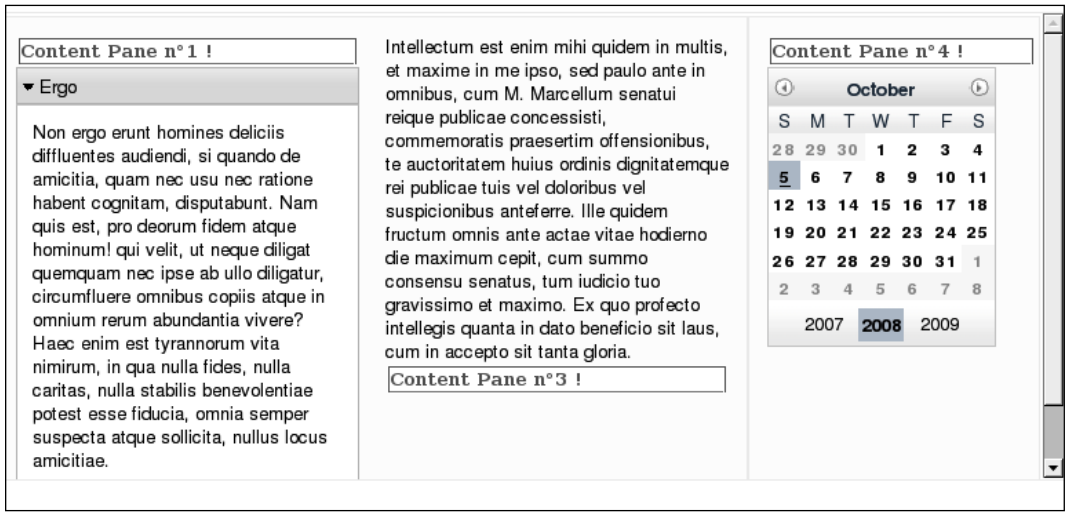
One of the most known examples is iGoogle, Google's personal homepage for users with a staggering amount of widgets that are easy to move around.



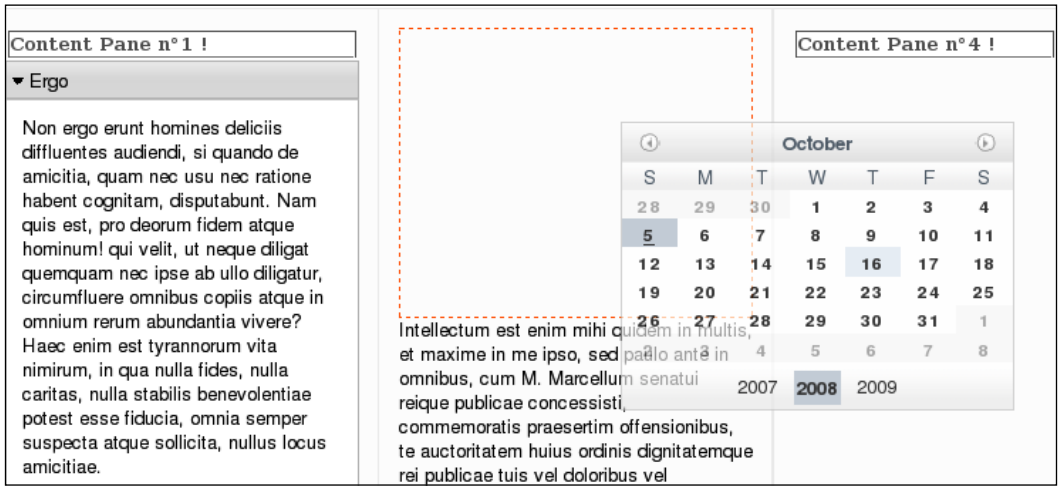
This functionality is called a GridContainer in Dojo. If you're not familiar with the concept and have never used a service which lets you rearrange widgets, it works like this:

1. The GridContainer defines a number of different columns, called zones.
2. Each column can contain any number of child widgets, including other containers (like AccordionContainer or BorderContainer).
3. Each child widget becomes draggable and can be dragged into a new position within its own column, or dragged to a new position in another column.
4. As the widget gets dragged, it uses a semi-transparent 'avatar'.
5. As the widget gets dragged, possible target drop zones open up and close themselves dynamically under the cursor, until the widget is dropped on one of them.
6. When a widget is dropped, the target column automatically rearranges itself to make the new widget fit.

Here is an example from `test_GridContainer.html in /dojox/layout/tests/`. This is what the GridContainer looks like from the beginning:



It has three columns (zones) defined which contain a number of child widgets. One of them is a Calendar widget, which is then dragged to the second column from its original position in the third:



Note the new target area being offered by the second column. This will be closed again if we continue to move the cursor over to the first column. Also, in the example above, transparency of 1.0 (none) is added to the avatar, which looks normal.

Finally, the widget is dropped onto the second column, both the source and target column arrange their widgets according to whether one has been added or removed.



The implications of this is that it becomes very simple to create highly dynamical interfaces. Some examples might be:

1. An internal "dashboard" for management or other groups in the company which needs rearrangeable views on different data sources. Portlets done right.
2. Using `dojox.charting` to create different diagrammatic views on data sources read from the server, letting the user create new diagrams and rearranging them in patterns or groups meaningful to the current viewer.
3. A simple front-end for a CMS-system, where the editor widget is used to enter text, and the user can add, delete or change paragraphs as well as dragging them around and rearranging their order.

An example of how to create a GridContainer using markup (abbreviated) is as follows:

```
<div id="GC1" dojoType="dojox.layout.GridContainer"
            nbZones="3"
            opacity="0.7"
            allowAutoScroll="true"
            hasResizableColumns="false"
            withHandles="true"
            acceptTypes="dijit.layout.ContentPane, dijit.TitlePane,
            dijit.ColorPalette, dijit._Calendar">
<div dojoType="dijit.layout.ContentPane" class="cpane" label="Content
Pane">Content Pane n?1 !</div>
```

```
<div dojoType="dijit.TitlePane" title="Ergo">
Non ergo erunt homines deliciis ...
</div>
<div dojoType="dijit.layout.ContentPane" class="cpane" label="Content
Pane">Content Pane n?2 !</div>
<div dojoType="dijit.layout.ContentPane" title="Intellectum">
Intellectum est enim mihi quidem in multis, et maxime in me ipso, sed
paulo ante in omnibus, cum M....
</div>
<div dojoType="dijit.layout.ContentPane" class="cpane" label="Content
Pane">Content Pane n?3 !</div>
<div dojoType="dijit.layout.ContentPane" class="cpane" label="Content
Pane">Content Pane n?4 !</div>
<div dojoType="dijit._Calendar"></div>
</div>
```

The GridContainer wraps all of its contents.These are not added is not added in a hierarchical manner, but instead all widgets are declared inside the GridContainer element. When the first column's height is filled, the next widget in the list gets added to the next column, and so on.

This is a quite unusual method of layout, and we might see some changes to this mode of layout since the GridContainer is very much beta [2008].

The properties for the GridContainer are the following:

```
//i18n: Object
//Contain i18n ressources.
i18n: null,

//isAutoOrganized: Boolean:
//Define auto organisation of children into the grid container.
isAutoOrganized : true,

//isRightFixed: Boolean
//Define if the right border has a fixed size.
isRightFixed:false,

//isLeftFixed: Boolean
//Define if the left border has a fixed size.
isLeftFixed:false,

//hasResizableColumns: Boolean
//Allow or not resizing of columns by a grip handle.
hasResizableColumns:true,

//nbZones: Integer
```

```
//The number of dropped zones.
nbZones:1,
//opacity: Integer
//Define the opacity of the DnD Avatar.
opacity:1,
//minColWidth: Integer
//Minimum column width in percentage.
minColWidth: 20,
//minChildWidth: Integer
//Minimun children with in pixel (only used for IE6 that doesn't
//handle min-width css property
minChildWidth : 150,
//acceptTypes: Array
//The gridcontainer will only accept the children that fit to
//the types.
//In order to do that, the child must have a widgetType or a
//dndType attribute corresponding to the accepted type.
acceptTypes: [],
//mode: String
//location to add columns, must be set to left or right(default)
mode: "right",
//allowAutoScroll: Boolean
//auto-scrolling enable inside the GridContainer
allowAutoScroll: false,
//timeDisplayPopup: Integer
//display time of popup in miliseconds
timeDisplayPopup: 1500,
//isOffset: Boolean
 //if true : Let the mouse to its original location when moving
 //(allow to specify it proper offset)
 //if false : Current behavior, mouse in the upper left corner of
 //the widget
isOffset: false,
//offsetDrag: Object
//Allow to specify its own offset (x and y) onl when Parameter
//isOffset is true
offsetDrag : {}, //
//withHandles: Boolean
//Specify if there is a specific drag handle on widgets
withHandles: false,
//handleClasses: Array
//Array of classes of nodes that will act as drag handles
handleClasses : [],
```

The property `isAutoOrganized`, which is set to `true` by default, can be set to `false`, which will leave holes in your source columns, and require you to manage the space in the target columns yourself.

The `opacity` variable is the opacity for the 'avatar' of the dragged widget, where `1` is completely solid, and `0` is completely transparent.

The `hasResizableColumns` variable also adds SplitContainer/BorderContainer splitters between columns, so that the user can change the size ratio between columns.

The `minColWidt/minChildWidth` variables manage the minimum widths of columns and child widgets in relation to resizing events.

The `AcceptTypes` variable is an important property, which lets you define which classes you allow to be dropped on a column. In the above example code, that string is set to `dijit.layout.ContentPane, dijit.TitlePane, dijit.ColorPalette, dijit._Calendar`. This makes it impossible to drop an AccordionContainer on a column.

The reason for this is that certain things would want to be fixed, like status bars or menus, but still inside one of the columns.

The `withHandles` variable can be set to `true` if you want each widget to get a visible 'drag handle' appended to it.

RadioGroup

The source code of `dojox.layout.RadioGroup` admits that it probably is poorly named, because it has little to do with radio buttons or groups of them, per se, even if this was probably the case when it was conceived.

The RadioGroup extends the StackContainer, doing something you probably had ideas about the first time you saw it – adding flashy animations when changing which child container is shown.

One example of how to use StackContainer and its derivatives is an information box for a list of friends. Each information box is created as a ContentPane which loads its content from a URL. As the user clicks on or hovers over the next friend on a nearby list, an event is triggered to show the next item (ContentPane) in the stack.

Enter the RadioGroup, which defines its own set of buttons that mirror the ContentPanes which it wraps.

The unit test `dojox/layout/tests/test_RadioGroup.html` defines a small RadioGroup in the following way:

```
<div dojoType="dojox.layout.RadioGroup" style="width:300px;
height:300px; float:left;" hasButtons="true">
<div dojoType="dijit.layout.ContentPane" title="Dojo" class="dojoPane"
style="width:300px; height:300px; "></div>
<div dojoType="dijit.layout.ContentPane" title="Dijit"
class="dijitPane" style="width:300px; height:300px; "></div>
<div dojoType="dijit.layout.ContentPane" title="Dojox"
class="dojoxPane" style="width:300px; height:300px; "></div>
</div>
```

As you can see, it does not take much space. In the test, the ContentPanes are filled with only the logos for the different parts of Dojo, defined as background images by CSS classes.

The RadioGroup iterates over each child ContentPane, and creates a "hover button" for it, which is connected to an event handler which manages the transition, so if you don' t have any specific styling for your page and just want to get a quick mock-up done, the RadioGroup is very easy to work with.

The default RadioGroup works very much like its parent class, StackContainer, mostly providing a simple wrapper that generates mouseover buttons.

In the same file that defines the basic RadioGroup, there are two more widgets: RadioGroupFade and RadioGroupSlide. These have exactly the same kind of markup as their parent class, RadioGroup.

RadioGroupFade looks like this in its entirety:

```
dojo.declare("dojox.layout.RadioGroupFade",
   dojox.layout.RadioGroup,
   {
   // summary: An extension on a stock RadioGroup, that fades the
   //panes.
   _hideChild: function(page){
      // summary: hide the specified child widget
      dojo.fadeOut({
         node:page.domNode,
         duration:this.duration,
         onEnd: dojo.hitch(this,"inherited", arguments)
      }).play();
   },
   _showChild: function(page){
```

```
        // summary: show the specified child widget
         this.inherited(arguments);
         dojo.style(page.domNode,"opacity",0);
         dojo.fadeIn({
            node:page.domNode,
            duration:this.duration
         }).play();
      }
   });
```

As you can see, all it does is override two functions from RadioGroup which manage how to show and hide child nodes upon transitions.

The basic idea is to use the integral Dojo animations `fadeIn` and `fadeOut` for the effects.

The other class, `RadioGroupSlide`, is a little bit longer, but not by much. It goes beyond basic animations and uses a specific easing function. In the beginning of its definition is this variable:

```
// easing: Function
// A hook to override the default easing of the pane slides.
easing: "dojo.fx.easing.backOut",
```

Later on, in the `overridden _hide` and `_showChild` functions, this variable is used when creating a standalone animation:

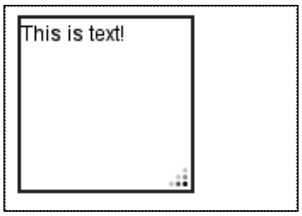
```
...
this._anim = dojo.animateProperty({
   node:page.domNode,
   properties: {
      left: 0,
      top: 0
   },
   duration: this.duration,
   easing: this.easing,
   onEnd: dojo.hitch(page,function(){
      if(this.onShow){ this.onShow(); }
      if(this._loadCheck){ this._loadCheck(); }
   })
});
this._anim.play();
```

What this means is that it is very simple to change (once again) what kind of animation is used when hiding the current child and showing next, which can be very usable.

Also, you can see that it is very simple to create your own subclass widget out of RadioGroup which can use custom actions when child nodes are changed.

ResizeHandle

The ResizeHandle tucks a resize handle, as the name implies, into the corner of an existing element or widget.



The element which defines the resize handle itself need not be a child element or even adjacent to the element which is to receive the handle. Instead the id of the target element is defined as an argument to the ResizeHandle as shown here:

```
<div dojoType="dijit.layout.ContentPane" title="Test window"
 style="width: 300px; height: 200px; padding:10px; border: 1px solid
 #dedede; position: relative; background: white;" id="testWindow">

      ...

<div id="hand1" dojoType="dojox.layout.ResizeHandle"
targetId="testWindow"></div>
   </div>
```

In this example, a simple ContentPane is defined first, with some custom styling to make it stand out a little bit. Further on in the same pages comes a ResizeHandle definition which sets the `targetId` property of the newly created ResizeHandle to that of the ContentPane ('testWindow').

The definition of the ResizeHandle class shows some predictable goodies along with one or two surprises:

```
//targetContainer: DomNode
//over-ride targetId and attch this handle directly to a
//reference of a DomNode
targetContainer: null,
//resizeAxis: String
```

```
//one of: x|y|xy limit resizing to a single axis, default to xy ...
resizeAxis: "xy",
//activeResize: Boolean
//if true, node will size realtime with mouse movement,
//if false, node will create virtual node, and only resize target
//on mouseUp.
activeResize: false,
//activeResizeClass: String
//css class applied to virtual resize node.
activeResizeClass: 'dojoxResizeHandleClone',
//animateSizing: Boolean
//only applicable if activeResize = false. onMouseup, animate the
//node to the new size.
animateSizing: true,
//animateMethod: String
//one of "chain" or "combine" ... visual effect only.
  combine will "scale"
//node to size, "chain" will alter width, then height
animateMethod: 'chain',
//animateDuration: Integer
//time in MS to run sizing animation. if animateMethod="chain",
//total animation playtime is 2*animateDuration.
animateDuration: 225,
//minHeight: Integer
//smallest height in px resized node can be
minHeight: 100,
//minWidth: Integer
//smallest width in px resize node can be
minWidth: 100,
```

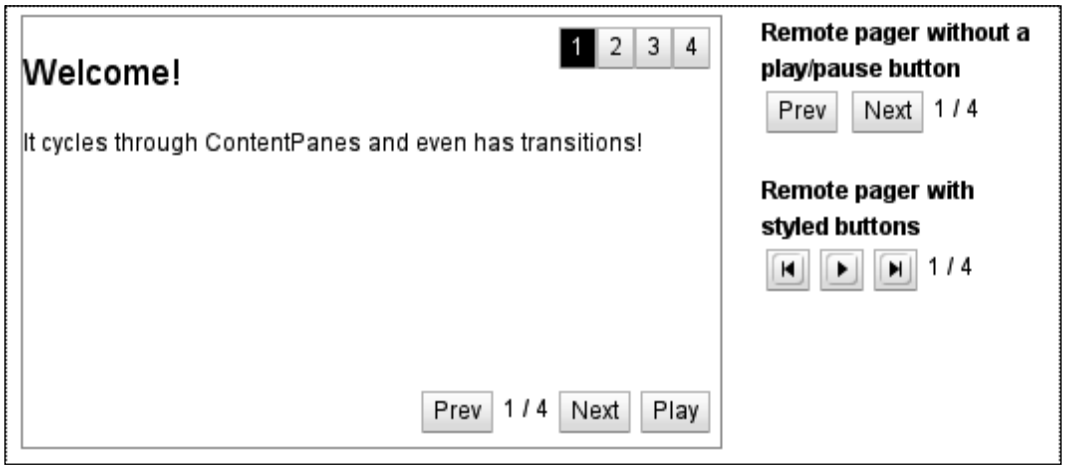
As could be expected, it is simple to change if the resizing is animated during mouse move or afterwards (`activeResize: true/false`). If afterwards, the `animateDuration` declares in milliseconds the length of the animation.

A very useful property is the ability to lock the resizing action to just one of the two axes. The `resizeAxis` property defaults to `xy`, but can be set to only `x` or only `y` as well. Both restricts resizing to only one axis and also changes the resize cursor to show correct feedback to which axis is 'active' at the moment.

If you at any point want to remove the handle, calling `destroy()` on it will remove it from the target node without any repercussions.

RotatorContainer

The RotatorContainer is yet another useful derivative of StackContainer, which implements a slideshow-like functionality. Each child (typically, ContentPanes) can set a custom `transitionDelay` property, which changes the time it is shown. Otherwise, the centrally configured `transitionDelay` on the RotatorContainer itself controls how long each child is shown.



The RotatorContainer is created in the following way:

```
<div dojoType="dojox.layout.RotatorContainer" id="myRotator"
showTabs="true" autoStart="true" transitionDelay="5000">
      <div id="pane1" dojoType="dijit.layout.ContentPane" title="1">
         Pane 1!
      </div>
      <div id="pane2" dojoType="dijit.layout.ContentPane" title="2">
         Pane 2!
      </div>
         <div id="pane3" dojoType="dijit.layout.ContentPane"
              title="3" transitionDelay="10000">
         Pane 3 with overrided transitionDelay!
      </div>
   </div>
```

The file the defines the RotatorContainer also defines a widget called RotatorPager, which can act as a kind of remote-control for a given RotatorContainer.

In the picture above, there are two RotatorPagers, each with a different look. The undermost is defined like this:

```
<div dojoType="dojox.layout.RotatorPager" class="rotatorIcons"
 rotatorId="myRotator">
  <button dojoType="dijit.form.Button" iconClass="previous"
   dojoAttachPoint="previous">Prev</button>
  <button dojoType="dijit.form.ToggleButton"
   iconClass="playPause" dojoAttachPoint="playPause"></button>
  <button dojoType="dijit.form.Button" iconClass="next"
   dojoAttachPoint="next">Next</button>
  <span dojoAttachPoint="current"></span> /
  <span dojoAttachPoint="total"></span>
</div>
```

The important thing here is the `rotatorId` property, which attaches the RotatorPager to a specific RotatorContainer.

Also, very good use has been made of `dojoAttachPoint`. Instead of using dojo/connect script tags or more convoluted schemes of attaching behavior to controls, the widget simply uses the same kind of association used inside a template. But turning it inside out, allowing the user of the widget an elegant way to make almost any element a controller for a specific activity.

The possible attachPoints are (from source):

```
//The pager can contain the following components:
//* Previous button
//  - Must be a dijit.form.Button
//  - dojoAttachPoint must be named "previous"
//* Next button
//  - Must be a dijit.form.Button
//  - dojoAttachPoint must be named "next"
//* Play/Pause toggle button
//  - Must be a dijit.form.ToggleButton
//  - dojoAttachPoint must be named "playPause"
//  - Use iconClass to specify toggled state
//* Current child #
//  - dojoAttachPoint must be named "current"
//* Total # of children
//  - dojoAttachPoint must be named "total"
```

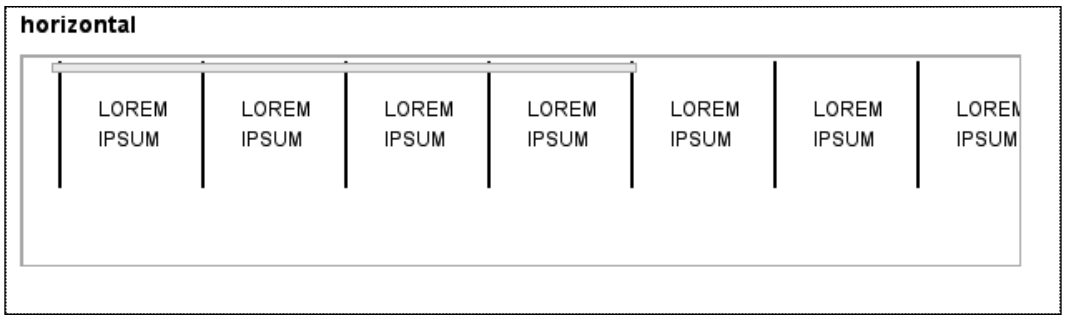
The last two attachPoints define spots where the RotatorContainer will display the current child being viewed, and the total number of children, respectively.

ScrollPane

The ScrollPane is again a variant of using mouseover to change widget state in a very intuitive manner.

Many times you have a list of things that isn't broad or multifaceted enough to warrant a table-like Grid or a Tree, but you still want to present it somewhat orderly. Also, the list might be quite long and you would like to make it easy for people to browse around in it.

One simple solution is to use a regular div with overflow visible and a fixed height. The ScrollPane does away with a lot of details and just lets you present things and browse them by just moving the mouse over either of the active ends. If there are more items to show, the ScrollPane will scroll the remaining items in a natural speed into the viewing area.



As you see in the picture above, the ScrollPane has a small helper line, which helps the user see which part of the list of items they are currently browsing.

The ScrollPane is very simple to use and can be created vertically as well.

In the future, we might see ScrollPanes that have separators based on the first letter of the title of items, and also creates a helper bar which is ordered in the same way.

The ScrollPane is created like this:

```
<div dojoType="dojox.layout.ScrollPane" style="width:100px;
 height:300px; border:1px solid #b7b7b7">
 <ol class="list" id="vertList">
 <li id="cloneMe"><a href="#"><span>testItem</span></a></li>
 </ol>
</div>
```

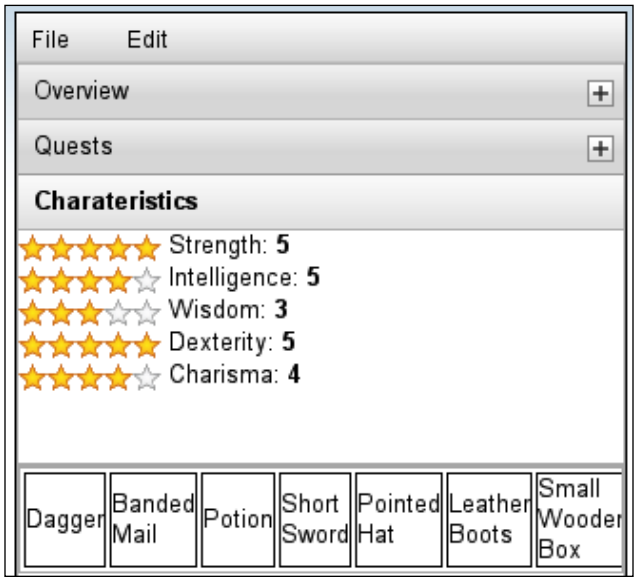
To get a horizontal ScrollPane, use the property orientation set to "horizontal".

Compound example using layout

The following is a more complicated example which is a small information box for a fictitious adventure game. A lot of information is to be displayed in a small area:

- Adventurer status
- Characteristics
- Inventory
- Quests

Using the BorderLayout container for overall layout, it consists of three parts: a toolbar, an accordioncontainer, and a horizontal ScrollPane top display the inventory.



The toolbar is not particularly active in the example, but shows how to use a simple toolbar menu in conjunction with a BorderLayout container.

The source code for the example is here:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/
  TR/html4/strict.dtd">
<html>
<head>
   <title>Chapter 6 Layout Example</title>
   <style>
```

```
        @import "dojo-1.1.0/dojo/resources/dojo.css";
        @import "dojo-1.1.0/dijit/tests/css/dijitTests.css";
        @import "dojo-1.1.0/dijit/themes/tundra/tundra.css";
        @import "dojo-1.1.0/dojox/widget/Rating/Rating.css";
    </style>
    <script type="text/javascript" src="dojo-1.1.0/dojo/dojo.js"
     djConfig="isDebug: true, parseOnLoad: true"></script>
    <script type="text/javascript">
        dojo.require("dijit.Menu");
        dojo.require("dijit.form.CheckBox");
        dojo.require("dijit.form.TextBox");
        dojo.require("dijit.layout.BorderContainer");
        dojo.require("dijit.layout.ContentPane");
        dojo.require("dijit.form.Button");
        dojo.require("dijit.Toolbar");
        dojo.require("dojox.widget.Rating");
        dojo.require("dojox.layout.ScrollPane");
        dojo.require("dijit.layout.AccordionContainer");
        dojo.require("dojo.parser");
        // scan page for widgets and instantiate them
    </script>
    <script>
        function handleMenu(m)
        {
            console.log("handleMenu called for menu '"+m+"'");
        }
    </script>
</head>
<body class="tundra">
<div dojoType="dijit.layout.BorderContainer" style="border:
 1px solid black; height: 300px; width:300px;padding: 10px;">
<div dojoType="dijit.layout.ContentPane" region="top">
<div id="menubar" dojoType="dijit.Toolbar" class="menuBar">
<div dojoType="dijit.form.DropDownButton">
<span>File</span>
<div dojoType="dijit.Menu">
<div dojoType="dijit.MenuItem" onClick="handleMenu('New profile')">
 New Profile</div>
<div dojoType="dijit.MenuItem" onClick="handleMenu('Open
 Profile')">Open Profile</div>
<div dojoType="dijit.MenuSeparator"></div>
<div dojoType="dijit.MenuItem" iconClass="dijitEditorIcon
 dijitEditorIconSave" onClick="handleMenu('Save Profile')">Save
 Profile</div>
```

```
<div dojoType="dijit.MenuItem" onClick="handleMenu('Save Profile
As..')">Save Profile As...</div>
</div>
</div>
<div dojoType="dijit.form.DropDownButton">
<span>Edit</span>
<div dojoType="dijit.Menu">
<div dojoType="dijit.MenuItem"  iconClass="dijitEditorIcon
dijitEditorIconCut" onClick="handleMenu('Cut')">Cut</div>
<div dojoType="dijit.MenuItem"  iconClass="dijitEditorIcon
dijitEditorIconCopy" onClick="handleMenu('Copy')">Copy</div>
<div dojoType="dijit.MenuItem"  iconClass="dijitEditorIcon
dijitEditorIconPaste" onClick="handleMenu('Paste')">Paste</div>
</div>
</div>
</div>
</div>
<div dojoType="dijit.layout.ContentPane" region="center">
<div dojoType="dijit.layout.AccordionContainer"   style="height:
210px;  overflow: hidden">
<div dojoType="dijit.layout.AccordionPane" title="Overview">
  Profile: <b>Foo Barsson</b><br/>
  Status: <b>Cool</b>
</div>
<div dojoType="dijit.layout.AccordionPane" title="Quests">
<bl>
<li>Defeat Giant Mouse in the Cellar of Hierloom House</li>
<li>Find the Apple of Perfume for Fiona the Enchantress</li>
<li>Collect six Emerald Centipedes</li>
</bl>
</div>
<div dojoType="dijit.layout.AccordionPane" title="Charateristics">
<span id="rating0" dojoType="dojox.widget.Rating" numStars="5" value
 = "5" onChange="dojo.query('#rating0Value')[0].innerHTML =
 this.value"></span>
Strength: <b><span id="rating0Value">5</span></b>
<br/>
<span id="rating1" dojoType="dojox.widget.Rating" numStars="5" value
 = "4" onChange="dojo.query('#rating1Value')[0].innerHTML =
 this.value"></span>
Intelligence: <b><span id="rating1Value">5</span></b>
<br/>
```

```
<span id="rating2" dojoType="dojox.widget.Rating" numStars="5" value
= "2" onChange="dojo.query('#rating2Value')[0].innerHTML = this.
value"></span>
Wisdom: <b><span id="rating2Value">2</span></b>
<br/>
<span id="rating3" dojoType="dojox.widget.Rating" numStars="5"
 value = "5" onChange="dojo.query('#rating3Value')[0].innerHTML =
 this.value"></span>
Dexterity: <b><span id="rating3Value">5</span></b>
<br/>
<span id="rating4" dojoType="dojox.widget.Rating" numStars="5"
 value = "4" onChange="dojo.query('#rating4Value')[0].innerHTML =
 this.value"></span>
Charisma: <b><span id="rating4Value">4</span></b>
<br/>
</div>
</div>
<div dojoType="dojox.layout.ScrollPane" region="bottom"
 orientation="horizontal" style=" border:1px solid #b7b7b7">
<table class="list" id="vertList">
<tr>
<td style="border:1px solid black;">Dagger</td>
<td style="border:1px solid black;">Banded Mail</td>
<td style="border:1px solid black;">Potion</td>
<td style="border:1px solid black;">Short Sword</td>
<td style="border:1px solid black;">Pointed Hat</td>
<td style="border:1px solid black;">Leather Boots</td>
<td style="border:1px solid black;">Small Wooden Box</td>
<td style="border:1px solid black;">Steel Helmet</td>
<td style="border:1px solid black;">6 Wooden Arrows</td>
<td style="border:1px solid black;">Orc Teeth</td>
<td style="border:1px solid black;">Wooden Staff</td>
<td style="border:1px solid black;">Zircon Ring</td>
</tr>
</table>
</div>
</div>
</div>
</body>
</html>
```

The Toolbar

The Toolbar is made up of two DropDown buttons, which add simple onclick handlers which call back to a single handler called `handleMenu`. As you can see, it is not particularly complicated, only logging the button which was clicked.

When using the Toolbar, it is very important to remember to use the 'tundra' class on the body element (or any other Dojo theme), since the toolbar effect is heavily dependent on the right classes.

The BorderContainer

When beginning a layout, it's a good idea to begin with the container and the constituent ContentPanes, setting heights and widths as good as you can. It's often hard to get the ratios right the very first time, so it's beneficial using a 'scaffolding' layout with a minimum of other components while trying out which stylings are needed.

Another good idea is to use different glaring background colors for all ContentPanes, possibly also solid borders. This makes it easier to spot bugs in the design as quickly as possible.

The design used here uses only three parts: one "top" for the Toolbar, a "center" for the AccordionContainer, and a "bottom" for the Inverntory ScrollPane.

Ideally, the Inventory would have been a fourth tab on the AccordionPane, but since AccordionPane does not support other Layout Containers as children, it had to have its own ContentPane parallel to the AccordionContainer instead.

The AccordionContainer

Three different AccordionPanes were used, only one showing at a time, with a nice animation between changes. The central pane "Characteristics" uses a lot of Rating widgets. These are very effective at displaying values at a glance, and the ratings provide a simple example showing how to collect the changes to each widget.

The ScrollPane

The ScrollPane is like the AccordionContainer, a very good way to tuck away a lot of information that is still browseable in an intuitive manner. The adventurer has collected quite a few items, all of which do not fit in the allotted width for the BorderContainer. Moving the cursor to the right or left of the ScrollPane will pan hidden items into sight automatically.

Note that when using a "horizonal" layout for the ScrollPane, you must also provide the elements horizontally. If a list element would have been used instead of a table, the information would have been displayed vertically regardless.

One step forward—creating a widget

Now that the BorderContainer display works fine, we suddenly get a change of plans. No longer it is sufficient to display only one adventurer, instead we need to create any number of adventurers, possibly as separate windows.

Clearly, this is possible given the widgets we've already discussed in this chapter. But to create any number of adventurer profiles on screen, we need to reach back a couple of chapters and revisit how to create a Dijit.

Why a Dijit? Because we need to leverage Dojo to create a template out of the design above. This way we can create as many copies of it as we would like, without having to take care of specific ids, colliding global variables, and the like. Fortunately, creating a Dojo Widget out of a single mockup page is fairly simple.

First, we create a subdirectory for our project, which might contain more than one widget when we're done. The best choice is to put this directory parallel to the other Dojo directories so that we can use `dojo.require()` for out new widget.

We will use the following structure:

```
dojo
dijit
dojox
adventure
 profile.js
 profile.html
 profile_test.html
```

So we create a directory called `adventure` and call the widget `profile`, with the definition residing inside the file `profile.js`. That file, in turn, points to the file `profile.html` as containing the html template for the widget.

The test file is a regular html file which contains an example test of the new widget:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/
TR/html4/strict.dtd">
<html>
   <head>
      <title>Adventure Profile Test</title>
      <style>
         @import "../dojo/resources/dojo.css";
         @import "../dijit/themes/dijit.css";
         @import "../dijit/themes/tundra/tundra.css";
         @import "../adventure/profile.css";
         @import "../dojox/widget/Rating/Rating.css";
      </style>
      <script>
         djConfig=
         {
            isDebug: true,
            parseOnLoad: true
         }
      </script>
      <script type="text/javascript" src="../dojo/dojo.js"></script>
      <script type="text/javascript">
         dojo.require("dojo.parser");
         // scan page for widgets and instantiate them
         dojo.require("adventure.profile");
      </script>
   </head>
   <body class="tundra">
      <div dojoType="adventure.profile"
         name = "Foo Barsson"
         status = "OK"
         questinfo = "Defeat Giant Mouse in the Cellar of Hierloom
         House,Find the Apple of Perfume for Fiona the
         Enchantress,Collect six Emerald Centipedes"
         strength = "5"
         intelligence="4"
         wisdom="4"
         dexterity="5"
         charisma="4"
         inventory="Dagger,Banded Mail,Potion,Short Sword,Pointed
         Hat,Leather Boots,Small Wooden Box,Steel Helmet,6 Wooden
         Arrows,Orc Teeth,Wooden Staff,Zircon Ring" ></div>
```

```
        <br/>
        <div  dojoType="adventure.profile"
           name = "Fighter Fightersson"
           status = "OK"
           questinfo = "Find Orcs,Kill Orcs,Find Entrance to
           Dungeon,Find Loot"
           strength = "5"
           intelligence="2"
           wisdom="3"
           dexterity="5"
           charisma="3"
           inventory="Cheese, Nails,Banded Mail,Potion,Long Sword,Brass
           Gauntlets,Leather Boots,Mithril Armor,Steel Helmet,Flask of
           Oil,Torch,Holy Handgrenade,Robe of Invisibility,Ruby Ring">
         </div>
    </body>
</html>
```

The first step in creating a widget of a mockup page is to remove all data and either put it in separate files, to be loaded from data-driven components (more on this in next chapter), or given as arguments as we see above. Two instances of the newly created **adventure.profile** widget are put on the page, each with varying arguments. These will be inserted into the widget's variables with the same names automagically.

Since the mockup used some static tables to lay out the data, I've opted to put the lists of things (like quests) into comma-separated strings. These are then parsed when the widget starts up and used as content of the requisite kind of markup created on the fly and added to parent elements in the template.

The template for the widget is where most of the markup has gone. It looks mostly the same, with some crucial differences:

1. All static ids on elements have either been removed, or augmented with `${id}`. When adding `${id}`, you take advantage of Dojo's common templating mechanisms to insert the current Dom id of the widget's top element, making the id unique in the page, regardless of how many instances of the widget are present.
2. When possible, dojoAttachPoints have been added, where before unique id references were used to refer to elements and/or sub-widgets. When defining a `dojoAttachPoint="xxx"` on an element, that name will then be accessible inside the JavaScript widget definition as `this.xxx`.
3. All parts of the widget that display contents of variables replace the hard-coded values of the mockup with a `${}` variable.

To illustrate the last point, the AccordionPane that shows the name and status of the adventurer now looks like this:

```
<div dojoType="dijit.layout.AccordionPane" title="Overview">
      Profile: <b>${name}</b><br/>
      Status: <b>${status}</b>
</div>
```

where `name` and `status` are arguments given in the markup creation of the widget.

An example of how to use attach points can be seen in the former ScrollPane declaration in the mockup, which now is just an ordinary element:

```
<div dojoAttachPoint="inventorypane">
   <table>
      <tr dojoAttachPoint="inventoryRow">
      </tr>
   </table>
</div>
```

Here is the complete template for the widget:

```
<div>
<div dojoType="dijit.layout.BorderContainer" style="border: 1px solid
 black; height: 300px; width:300px;padding: 10px;">
<div dojoType="dijit.layout.ContentPane" region="top">
<div  dojoType="dijit.Toolbar" class="menuBar">
<div dojoType="dijit.form.DropDownButton">
<span>File</span>
<div dojoType="dijit.Menu">
<div dojoType="dijit.MenuItem" dojoAttachPoint=
 "new_profile">New Profile</div>
<div dojoType="dijit.MenuItem" dojoAttachPoint="open_profile">
 Open Profile</div>
<div dojoType="dijit.MenuSeparator"></div>
<div dojoType="dijit.MenuItem" iconClass="dijitEditorIcon
 dijitEditorIconSave" dojoAttachPoint="save_profile">Save Profile
</div>
<div dojoType="dijit.MenuItem" dojoAttachPoint="save_profile_as">
 Save Profile As...</div>
</div>
</div>
<div dojoType="dijit.form.DropDownButton">
<span>Edit</span>
<div dojoType="dijit.Menu">
<div dojoType="dijit.MenuItem"  iconClass="dijitEditorIcon
 dijitEditorIconCut" dojoAttachPoint="cut">Cut</div>
```

```
<div dojoType="dijit.MenuItem"  iconClass="dijitEditorIcon
 dijitEditorIconCopy" dojoAttachPoint="copy">Copy</div>
<div dojoType="dijit.MenuItem"  iconClass="dijitEditorIcon
dijitEditorIconPaste" dojoAttachPoint="paste">Paste</div>
</div>
</div>
</div>
</div>
<div dojoType="dijit.layout.ContentPane" region="center">
<div dojoType="dijit.layout.AccordionContainer"
 style="height: 210px;  overflow: hidden">
<div dojoType="dijit.layout.AccordionPane" title="Overview">
Profile: <b>${name}</b><br/>
Status: <b>${status}</b>
</div>
<div dojoType="dijit.layout.AccordionPane" title="Quests">
<bl dojoAttachPoint="quests"></bl>
</div>
<div dojoType="dijit.layout.AccordionPane" title="Charateristics">
<span id="${id}rating0" dojoType="dojox.widget.Rating" numStars="5"
 value = "${strength}" onChange="dojo.byId('${id}rating0Value').
 innerHTML = this.value"></span>
Strength: <b><span id="${id}rating0Value">${strength}</span></b>
<br/>
<span id="${id}rating1" dojoType="dojox.widget.Rating" numStars="5"
 value = "${intelligence}" onChange="dojo.byId('${id}rating1Value').
 innerHTML = this.value"></span>
Intelligence: <b><span id="${id}rating1Value">${intelligence}
</span></b>
<br/>
<span id="${id}rating2" dojoType="dojox.widget.Rating" numStars="5"
 value = "${wisdom}" onChange="dojo.byId('${id}rating2Value').
 innerHTML = this.value"></span>
Wisdom: <b><span id="${id}rating2Value">${wisdom}</span></b>
<br/>
<span id="${id}rating3" dojoType="dojox.widget.Rating" numStars="5"
 value = "${dexterity}" onChange="dojo.byId('${id}rating3Value').
 innerHTML = this.value"></span>
Dexterity: <b><span id="${id}rating3Value">${dexterity}</span></b>
<br/>
<span id="${id}rating4" dojoType="dojox.widget.Rating" numStars="5"
 value = "${charisma}" onChange="dojo.byId('${id}rating4Value').
 innerHTML = this.value"></span>
Charisma: <b><span id="${id}rating4Value">${charisma}</span></b>
<br/>
</div>
```

```
</div>
<div dojoAttachPoint="inventorypane">
<table>
<tr dojoAttachPoint="inventoryRow">
</tr>
</table>
</div>
</div>
</div>
</div>
```

The rating widgets are a bit verbose in their definitions. One alternative would have been to create a separate customized subclass of the Rating widget, which managed its own update, to also show a number to the side.

We could also, of course, have managed the changes inside the logic of this widget as well.

In the file that defines the widget, the startup function reads both the quests and the inventory strings, which are split into an array and then used to populate the two parts of the widget that show variable amounts of data.

Here is the JavaScript for the widget definition:

```
dojo.provide("adventure.profile");
dojo.require("dijit._Templated");
dojo.require("dijit._Widget");
dojo.require("dijit.Dialog");
dojo.require("dijit.Menu");
dojo.require("dijit.layout.BorderContainer");
dojo.require("dijit.layout.ContentPane");
dojo.require("dijit.form.Button");
dojo.require("dijit.Toolbar");
dojo.require("dojox.widget.Rating");
dojo.require("dojox.layout.ScrollPane");
dojo.require("dijit.layout.AccordionContainer");
dojo.declare("adventure.profile", [ dijit._Widget, dijit._Templated],
{
   templatePath: dojo.moduleUrl("adventure","profile.html"),
   widgetsInTemplate: true,
   name: "none",
   status: "undefined",
   questinfo: "",
   strength: 0,
   intelligence: 0,
```

```
wisdom: 0,
dexterity: 0,
charisma: 0,
inventory: "",
handleMenu: function(e)
{
   var t = e.currentTarget.textContent;
   console.log("adventure.profile handleMenu called for "+t);
},
startup: function()
{
   console.log("startup for adventure.profile called.");
   dojo.connect(this.new_profile, "onClick",  this.handleMenu);
   dojo.connect(this.open_profile, "onClick",  this.handleMenu);
   dojo.connect(this.save_profile, "onClick",  this.handleMenu);
   dojo.connect(this.save_profile_as, "onClick",  this.handleMenu);
   dojo.connect(this.cut, "onClick",  this.handleMenu);
   dojo.connect(this.copy, "onClick",  this.handleMenu);
   dojo.connect(this.paste, "onClick",  this.handleMenu);
   var q = this.questinfo.split(",");
   for(var quest in q)
   {
     var li = dojo.doc.createElement('li');
      var span = dojo.doc.createElement('span');
      li.appendChild(span)
      span.innerHTML=q[quest];
      this.quests.appendChild(li);
   };
   var i = this.inventory.split(",");
   for(var it in i)
   {
      var td = dojo.doc.createElement('td');
      td.innerHTML=i[it];
      td.style.height = "30px;";
      td.style.border = "1px solid black";
      this.inventoryRow.appendChild(td);
   };
   var pane = new dojox.layout.ScrollPane(
   {
      region: "bottom",
      orientation: "horizontal"
   }, this.inventorypane);
   pane.startup();
},
```

```
    postCreate: function()
    {
       console.log("postCreate for adventure.profile called.");

    }
});
```

Note that we were able to move all the specific require statements from the html file where the widget is created, to the definition of the widget. This makes the use of the widget much simpler.

On the other hand, all specific CSS that the constituent parts of the widget need still needs to be included in the file where the widget is to be created by markup, so true transparency of resources is still a bit away.

Another important thing to reemphasize is that all arguments to be passed as variables from markup to widget need also be defined in the widget. If the widget variable `name` was to be deleted for some reason, that argument would not be carried over from markup to widget and will look "undefined" in the widget upon creation.

Summary

- Dojo has a variety of Layout Containers for arranging other components and markup.
- Most Layout Containers use ContentPanes which demarcate markup into content parcels, which are then arranged and rearranged in different manners.
- The content of a ContentPane can be static, loaded from a URL, or reloaded.
- Using Layout Containers make it simpler to arrange elements on the page.
- The RadioGroup LayoutContainer is a good component for simple site navigation, showing one HTML snippet at a time, using animated transitions.
- The FloatingPane acts as a classical "window" which can be resized, closed, dragged around, and even collapsed to a ‘docking’ element, for later expansion.

7
Data, Trees, and Grids

The main idea behind `dojo.data` is to separate presentation from data in the browser. Too often, complex widgets get created that can only read a certain kind of data, or can only read data from a certain kind of URL and not from a file, for instance.

Dojo Dijits that use dynamic, external data have been created to use `dojo.data` datastores to read (or write) their data from (or to).

This separation makes it possible to use a dynamic widget (a Tree, for instance) with a completely new source of data, just by writing a new `dojo.data` store. The Tree will never know what kind of store it is, just that it confirms to the `dojo.data` specification.

There are a number of different stores, some for specialized file formats (such as CSV or XML), some for special web-based services (like Google Search or Flickr images), but most are general-purpose stores that explore the different possible features in the `dojo.data` API.

However, it's not possible to just throw away some arbitrarily arranged data (even if you know the format). So one important feature of `dojo.data` is that you have to define a query as part of the arguments when creating a store. The query tells the store what property in the data to search for and can also work to filter data.

dojo.data and dojox.data

There are several different kinds of stores, with the well-tested ones under `dojo.data` and the experimental and more interesting ones under `dojox.data`.

As of the writing of this book, the stores under `dojo.data` are:

- ItemFileReadStore - Reads JSON data from a URL.
- ItemFileWriteStore - Reads JSON data from a URL, but allows in-memory changes to data.

The stores under `dojox.data` are:

- `AndOrReadStore` - Based on ItemFileReadStore, but supports complex queries.
- `AndOrReadStore` - Based on ItemFileWrite as above.
- `AtomReadStore` - Read-Only store for Atom XML-based feeds.
- `CouchDbRestStore` - Based on JsonRestStore for r/w `couchDb` operations.
- `CsvStore` - A simple store that reads CSV files.
- `FileStore` - A Read-Only store for filesystem traversal and file read.
- `FlickStore` - A simple store for explicit Flickr feeds without wildcards.
- `FlickRestStore` - Based on FlickStore using Flickr's REST services instead.
- `GoogleSearchStore` - Read-Only store for Google web searches.
- `GoogleFeedStore` - Read-Only store that uses Google's RSS/Atom feed services.
- `HtmlStore` - Read-Only store that reads hierarchical DOM nodes as data.
- `HtmlTableStore` - Based on HtmlStore but specifically made for tables.
- `JsonPathStore` - Takes an arbitrary JS object and uses as store data.
- `JsonRestStore` - Read and Write store using REST access to RESTful services.
- `KeyValueStore` - Read store for key-value JS objects.
- `OpmlStore` - Read store for the OPML XML data format.
- `PersevereStore` - Read and Write store for the server-side JavaScript data repository system Persevere – based on JsonRestStore.
- `PicasaStore` - Read store for Picasa web albums.
- `QueyReadStore` - Read store which can pass different queries to the server for generating data, which can then be queried differently in the client.
- `S3Store` - Read and Write store for Amazon S3 storage service.
- `ServiceStore` - Read and Write store that uses `dojox.rpc` clients to access server, using either URLs or SMD files.
- `SnapLogicStore` - Read only store for the SnapLogic open source data integration framework.
- `WikipediaStore` - Read access to wikipedia searches.
- `XmlStore` - Read and Write store for XML data.

Quick example

Let's say that you need to create a FilteringSelect widget which reads data dynamically from the server backend. This is often used in single-page applications where you have several FilteringSelects that are dependent on each other.

You could have a first FilteringSelect with a static list of countries. When that FilteringSelect fires an onChange event, a handler function builds a custom URL for data request for the second FilteringSelect. Part of the URL will then be the country which was selected.

The second list might then be populated like this (if USA was selected on the first FilteringSelect):



If we look at the code for this example (taken from the unit test for the FilteringSelect widget), it first contains a definition of a data store, which is then used as input for the FilteringSelect:

```
<div dojoType="dojo.data.ItemFileReadStore" jsId="stateStore"
   url="../_data/states.json"></div>
<p>FilteringSelect #2: url, autoComplete=true:</p>
<label for="setvaluetest">state list 2:</label>
<input searchAttr="name"
      id="setvaluetest"
      dojoType="dijit.form.FilteringSelect"
      store="stateStore"
      name="state2"
      autoComplete="true"
      onChange="setValue('value2', arguments[0]);"
      invalidMessage="Invalid state name"
>
<span>Value: <input id="value2" disabled></span>
```

The important part of the data store is the source of the data, the `url` parameter. The next important part is the actual format of the data. Most stores in `dojo.data` use a similar format as the testing file `states.json`:

```
{
   identifier:"abbreviation",
   items: [{name:"Alabama", label:"Alabama",abbreviation:"AL"},
   {name:"Alaska", label:"Alaska",abbreviation:"AK"},
   {name:"American Samoa", label:"American Samoa",abbreviation:"AS"},
   {name:"Arizona", label:"Arizona",abbreviation:"AZ"},
   {name:"Arkansas", label:"Arkansas",abbreviation:"AR"},
   ...]
}
```

The list was abbreviated, so the ellipsis (...) is not part of the format, naturally.

You see that the data format is JSON formatted text, which defines an object literal with two members: `identifier` and `items`. The `identifier` property tells the store which part of the data items are to be used as an unique identifier.

Then follows an array declaration with a number of object literal definitions of objects which must at least contain the `identifier` property.

The third important thing about this code snippet is the `store` property of the FilteringSelect, which tells it to actually use the store.

In the example above, both store and widget are declared in markup, but it works exactly the same way when creating stores and widgets that use them programmatically.

Models

Some widgets can use (and for some configurations require) a man-in-the-middle object, called a model. The model can act as a decorator, which adds information about the data from the store or about how it is to be parsed which is not applicable to the data itself.

For instance, the Tree widget can use a model object which defines the label for the tree, and the Grid widget prior to 1.2 required a model between the Grid and its store for setting queries.

The models are internal to the component using them, with a specific `model.js` file describing the contract used for the specific component or widget.

Models communicate changes in their state both up to the component and down to the data store, so if a Tree adds a new item to a branch, it will inform the model, which in turn updates the data store. Conversely, if the data store gets an item deleted, it will update the model, which in turn updates the Tree.

Data sources, stores, and consumers

Components that use data stores expect (and demand, depending on what kind of component they are) some or all of the following data store APIs: Read, Write, Identity, and/or Notification.

Note that you don't really need to understand the APIs to be able to use data stores, they're presented here to give an overview of what kind of responsibilities a data store can have, and as a guide if and when you need to write your own data store.

The overall idea behind the `dojo.data` APIs is that you access items. However, when you've read an item successfully from a store, you must access the data in the item by calling store functions. This is due to the fact that items are just representations of data, which may not be loaded locally.

Therefore, always use the Read API function `getValue` to access the value of a given attribute, instead of trying to find it by hand inside the item itself.

Again, you will not need to use these functions when trying out some data-driven components the first time, but it makes it easier to understand how stores work.

- Read - Reading items, getting values of attributes of items. Basic queries.
- Write - Manages writing to in-memory storage. Can be extended for customized writebacks to services.
- Identity - For stores whose items have unique identifiers.
- Notification - Defines a number of callbacks for store events.

Here are the API functions for each API (formatted from respective files in the directory dojo/data/api):

dojo.data.Read

- `getValue(item, attribute, defaultValue)`

  If you have already retrieved an item from the datastore, you must use this function to get the actual data. An optional `defaultValue` can be provided, if the item does not have the declared attribute.

- `getValues(item, attribute)`

  Same as above, but returns an array, in case the `attribute` contains multiple values.

- `getAttributes(item)`

  For a given item, returns an array with the attribute names.

- `hasAttribute(item, attribute)`

  Returns **true** if the `item` has an attribute with the given name.

- `containsValue(item, attribute, value)`

  Checks if the `value` is a member of the array of multiple values (`getValues`) for the given attribute.

- `isItem(item)`

  Checks if the given object is an item in the data store.

- `isItemLoaded(item)`

  Checks if the given item is loaded (if the data store reads from a remote server or service).

- `loadItem(keywordArgs)`

  Forces a load of the given item, for remote and partially loaded data stores. The `keywordArgs` object takes four different properties:

  ```
  {
     item: object,
     onItem: Function,
     onError: Function,
     scope: object
  }
  ```

  The `item` is, as always, the actual item in the store to get. `OnItem` and `onError` are functions to be called when the item is successfully retrieved or fails to be retrieved. The `scope` parameter is the object that will be hitched to be the `this` for the functions when they are called.

- `fetch(keywordArgs)`

  Executes a query and returns a `dojo.data.api`. Requests object with the items that resulted from the query. The `keywordArgs` object can contain the following properties:

  ```
  {
     query: query-object or query-string,
     queryOptions: object,
     onBegin: Function,
     onItem: Function,
     onComplete: Function,
     onError: Function,
     scope: object,
     start: int
  ```

```
   count: int
   sort: array
}
```

As with the `loadItem` function, the `onXXX` functions are callback functions being called at various parts of the fetch cycle, and the `scope` object will be the `this` object for the respective functions. Setting the scope object to `this` (unquoted) will retain the current scope for the callback functions as well.

The `start` and `count` properties are especially interesting, since these provide the functionality to page through different parts of a large result set, often being directly applicable to server-side queries that work in the same manner.

The `sort` parameter is itself another JavaScript object literal, with the following structure:

```
{
   attribute: attribute || attribute-name-string,
   descending: true|false;   // Optional.  Default is false.
}
```

The list of properties inside the object literal provides implementation-specific settings for search in a store and as such can be different for different data stores. The `descending` property is a special property, however, which defines if the sorting on the specific attribute(s) will be returned in a descending or ascending list of objects.

The `query` parameter is again generic and what it contains depends on the kind of store used. However, most stores use an object which contains name/value pairs, like {`foo: bar, baz: quatz`}. Another dominant theme between many stores is to take the query object to refer to item attributes and wildcard matches. For example, a query of this type would look like:

```
{name: 'a*'}
```

This query would match any item in the store which contained a `name` attribute which began with an 'a'.

The `queryOptions` parameter is optional and controls how a query is handled. The default is to respect case in string properties and to only load data on demand.

```
{
   ignoreCase: boolean,
   deep: boolean
}
```

In the code above, `ignoreCase` declares if the query should be case-agnostic, and deep. If **true**, it tells the store to pre-load nested items, so that all are locally loaded. The alternative is to load only the items directly affected by the query without loading any nested objects.

dojo.data.Write

- `newItem(keywordArgs, parentInfo)`

  Returns a newly created item, which is initialized with values for properties from the `keywordArgs` object. For example, if the `keywordArgs` object is defined like this: `{name: 'bar', size: 17}`, the new item will have those properties.

  If the store supports hierarchical items, an optional `parentInfo` parameter can be declared, which will have the following format:

  ```
  {
     parent: someItem,
     attribute: 'attrname'
  }
  ```

  The `parent` property refers to the parent item for the newly created item, where a reference to the newly created item will be created with the attribute `'attrname'`. If the attribute already exists and is an array or object, the new object will be appended to the value of the parent's attribute.

- `deleteItem(item)`

  Deletes the item from the store.

- `setValue(item, attribute, value)`

  Sets the attribute of the given item to the value.

- `setValues(item, attribute, values)`

  Sets the multi-valued attribute to the contents of the `values` array.

- `unsetAttribute(item, attribute)`

  Deletes all values, or the value for the given attribute on the given item.

- `save(keywordArgs)`

  Saves the current unsaved changes made to the local datastore, so that any remote server is updated.

  The `keywordArgs` parameter is defined like this:

  ```
  {
     onComplete: function
     onError: function
     scope: object
  }
  ```

Where the callback functions and scope parameters work in similar fashion as in the Read API.

- `revert()`

  Cancels any unsaved changes. Literally reverts the store to last state after it was either first created or saved.

- `isDirty(item)`

  Returns **true** if the given item argument has been changed and not saved yet.

dojo.data.Identity

- `getIdentity(item)`

  Returns the unique identifier for the given item.

- `getIdentityAttributes(item)`

  Returns an array of attribute names that are used as unique identifiers for the given item.

- `fetchItemByIdentity(keywordArgs)`

  `keywordArgs` contains mostly callbacks of the same kind as the `fetch()` function accepts in the Read API. The important property is `identity`**.**

```
{
   identity: string|object,
   onItem: Function,
   onError: Function,
   scope: object
}
```

dojo.data.Notification

- `onSet(item, attribute, oldValue, newValue)`

  This function (as all other function in this API) is a hook, which is called by all other functions in all `dojo.data` APIs that modify an item in some form, even `unsetAttribute`.

- `onNew(item, parentInfo)`

  Called when a new item is created. If a new item was created without providing a `parentInfo` argument, the new item will be placed under the root item, which will lead to a subsequent `onSet` call for the root element of the store.

If the `parentInfo` argument object is provided, no `onSet` will be called for the parent element (which needs to be provided since this then is a hierarchical store), and instead the whole `parentInfo` object will be passed along to this `onNew` hook instead, with some additional information. This is the structure of the second (`parentInfo`) argument, which defines the name of the attribute on the parent that refers down to the new child, the new value (the child) and any old value (previous child, if any):

```
{
   item: someItem,
   //The parent item
   attribute: "attribute-name-string",
   //The attribute the new item was assigned to.
   oldValue: something,
   //Whatever was the previous value for the attribute.
   //If it is a single-value attribute only, then this value will
   //be a single value.
   //If it was a multi-valued attribute, then this will be an
   //array of all the values minues the new one.
   newValue: something
   //The new value of the attribute.  In the case of single value
   //calls, such as setValue, this value will be
   //generally be an atomic value of some sort (string, int, etc,
   //object). In the case of multi-valued attributes,
   //it will be an array.
}
```

- `onDelete(deletedItem)`

  Is called whenever an item is deleted, with the object that just has been removed from the data store as argument.

Common functions for all APIs

- `getFeatures()`

  Returns an object with key-value pairs that declare which APIs the store supports. For instance, the default implementation in the Identity store looks like this:

```
getFeatures: function(){
   //summary:
   //See dojo.data.api.Read.getFeatures()
   return {
       'dojo.data.api.Read': true,
       'dojo.data.api.Identity': true
   };
},
```

This makes it easy for the class or component using the store to see that it supports all features that it requires, or possibly define how the component should present itself or its data.

A simple tree example

Let's say that you want to read a dynamically generated JSON file from the server, and present it as a tree. You don't need any events (yet), you don't need to create new nodes on the fly or change the styling of the tree. Just a simple tree is all you need.

Actually, what you probably want to do first is to create a tree without any server at all, just to quickly play around with the ideas, but we begin by exploring a scenario with a server, or file on a server, and we'll get to the data-in-a-page scenario in just a bit.

Here is a simple page which generates a tree on loading (situated in a directory parallel to `dojo`, `dijit`, or similar):

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
 "http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
   <title>Simple Tree Example</title>
   <style someProperty="text/css">
      @import "../dojo/resources/dojo.css";
      @import "../dijit/css/dijitTests.css";
   </style>
   <!-- required: the default dijit theme: -->
   <link id="themeStyles" rel="stylesheet" href="../dijit/themes/
    tundra/tundra.css">
   <!-- required: dojo.js -->
   <script type="text/javascript" src="../dojo/dojo.js"
    djConfig="isDebug: true, parseOnLoad: true"></script>
   <script language="JavaScript" someProperty="text/javascript">
    dojo.require("dijit.dijit"); // optimize: load dijit layer
    dojo.require("dojo.data.ItemFileReadStore");
    //dojo.require("dojo.data.ItemFileWriteStore");
    dojo.require("dijit.Tree");
    //Declare function that will be called when page is fully loaded
    dojo.addOnLoad(function()
      {
    //Create a simple Store to hold the JSON test data.
    myStore = new dojo.data.ItemFileReadStore({url:'test.json'});
    myModel = new dijit.tree.TreeStoreModel({
```

```
        store: myStore,
        query: {name:'*'},
        rootId: '999'
       });
       myTree = new dijit.Tree({
        id: "myTree",
        model: myModel,
        childrenAttrs: ["children"]
       });
        dojo.byId("container").appendChild(myTree.domNode);
        myTree.startup();
       });
       </script>
   </head>
   <body class="tundra">
      <h1 class="testTitle">Chapter 7 simple Tree test</h1>
      <div id="container"></div>
   </body>
   </html>
```

This quick example might seem daunting at first, but we'll take things one step at a time. First, a ItemFileReadStore is created, which just refers to the URL where the file or server script to generate JSON is situated.

Then a model for the tree is created. From Dojo 2.0, trees will always require model 'filters' between themselves and data stores, so it's just as well to go with the flow, even in a quick example.

The model is of type TreeStoreModel, which defines a single-rooted tree. Also, properties present are `store` (of course, otherwise where do we get the data?) and `query`, which is a regular expression for a certain property inside the data of the store. The reason we use queries is that we might just want parts of a store, or we want to be able to filter out just certain parts dynamically.

This query is a pretty common one, `{name: '*'}`, but it might have been `{category: 'fruit'}` if we had a large data set with items that all had the property categories, but not all were fruit.

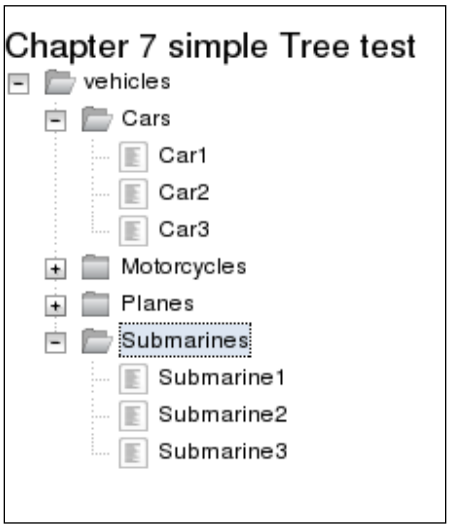
Then comes the property `rootId`, which declares the id of the item in the data store which will hold the rest of the tree. This is handy if we use the data for several things, and want to use only a part of the data store for the present task.

The last thing we do is to create the tree. Naturally, we point out the model in the object literal argument, but also the attribute of the children of an element, since we're dealing with hierarchical data here.

The data read by the store in the example looks like this:

```
{
   identifier: 'id',
   label: 'name',
   items: [
      {id: '999', name: 'vehicles',
      children: [
         { id: '1', name:'Cars' ,
         children: [
            {id: '10', name: 'Car1'},
            {id: '11', name: 'Car2'},
            {id: '12', name: 'Car3'},
         ]
         },
         { id: '2', name:'Motorcycles',
         children: [
            {id: '20', name: 'Motorcycle1'},
            {id: '21', name: 'Motorcycle2'},
            {id: '22', name: 'Motorcycle3'},
         ]
         },
         { id: '3', name:'Planes',
         children: [
            {id: '30', name: 'Plane1'},
            {id: '31', name: 'Plane2'},
            {id: '32', name: 'Plane3'},
         ]
         },
         { id: '4', name:'Submarines',
         children: [
            {id: '40', name: 'Submarine1'},
            {id: '41', name: 'Submarine2'},
            {id: '42', name: 'Submarine3'},
         ]
         }
      ]
      }
   ]
}
```

And here's a screenshot of the tree in action:



But let's say that you just want to do a tree locally in the page, perhaps you don't want to use a store, but want to create your own JSON on the fly.

Then you can actually create a ItemFileReadStore, but give it the data it should contain as a property argument, like this;

```
var data = {
            identifier: 'id',
            label: 'name',
            items: [
               {id: '999', name: 'vehicles',
               children: [
                  { id: '1', name:'Cars' ,
                  children: [
                     {id: '10', name: 'Car1'},
                     {id: '11', name: 'Car2'},
                     {id: '12', name: 'Car3'},
                  ]
                  },
                  { id: '2', name:'Motorcycles',
                  children: [
                     {id: '20', name: 'Motorcycle1'},
                     {id: '21', name: 'Motorcycle2'},
                     {id: '22', name: 'Motorcycle3'},
                  ]
```

```
                },
                { id: '3', name:'Planes',
                children: [
                   {id: '30', name: 'Plane1'},
                   {id: '31', name: 'Plane2'},
                   {id: '32', name: 'Plane3'},
                ]
                },
                { id: '4', name:'Submarines',
                children: [
                   {id: '40', name: 'Submarine1'},
                   {id: '41', name: 'Submarine2'},
                   {id: '42', name: 'Submarine3'},
                ]
                }
             ]
          }
          ]
       }
       myStore = new dojo.data.ItemFileReadStore({data: data})
```

Forests and trees

In the simple tree example, we had only one single tree, but sometimes you want to display a 'forest' of trees and create an artificial root for all of them. This case occurs when the data you want to display as a tree begins with an array of multiple items, which then also branches out.

You need then instead to use the `dijit.tree.ForestStoreModel` instead. First, we have to tweak our data:

```
{
   identifier: 'id',
   label: 'name',
   items: [
      { id: '1', name:'Cars' ,
      children: [
         {id: '10', name: 'Car1'},
         {id: '11', name: 'Car2'},
         {id: '12', name: 'Car3'},
      ]
      },
      { id: '2', name:'Motorcycles',
      children: [
```

```
            {id: '20', name: 'Motorcycle1'},
            {id: '21', name: 'Motorcycle2'},
            {id: '22', name: 'Motorcycle3'},
         ]
         },
         { id: '3', name:'Planes',
         children: [
            {id: '30', name: 'Plane1'},
            {id: '31', name: 'Plane2'},
            {id: '32', name: 'Plane3'},
         ]
         },
         { id: '4', name:'Submarines',
         children: [
            {id: '40', name: 'Submarine1'},
            {id: '41', name: 'Submarine2'},
            {id: '42', name: 'Submarine3'},
         ]
         }
      ]
   }
```

What we did here is to just remove the first (and single) object inside the uppermost `items` array, and instead fill it with a number of parallel children.

This changes the Tree Model line to:

```
myModel = new dijit.tree.ForestStoreModel({
          store: myStore,
         query: {name:'*'},
          rootLabel: 'vehicles'
        });
```

Note that since we don't have an enclosing object in our data called with the name `vehicles`, the `ForestStoreModel` will create a node with the relatively uncharming name `undefined` unless we add the property argument `rootLabel` when creating the ForestStore.

Tree events

The Tree has a number of function hooks including the following:

- `onClick`
- `onOpen`
- `onClose`

These can be hooked into directly, but it is generally recommended to never do immediate overrides in this manner, since you then overwrite (or risk being overwritten) any old handler function.

Using `dojo.connect`, you make sure that Dojo manages all handler functions for a given event or function. Remember also, that if you don't add the current context (this) as the third argument, you will have the context (or 'this') of the event handler, leaving you incapable of accessing any variables or properties outside the handler function.

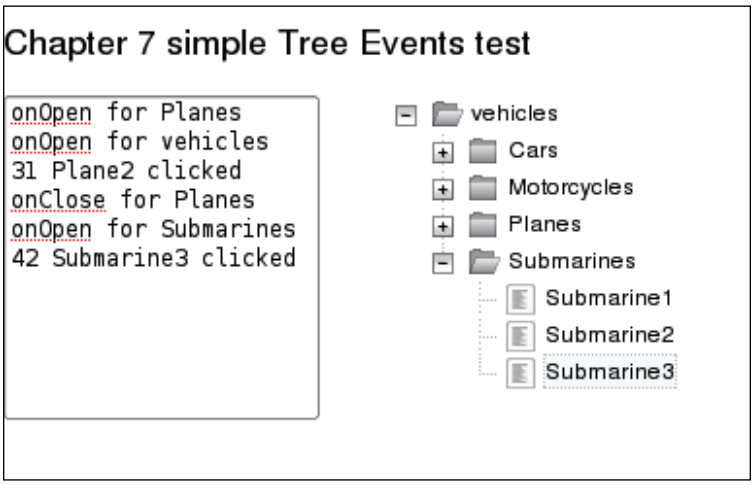
Here is a modified example, where a textarea displays the item information for any clicked Tree item:

```
myStore = new dojo.data.ItemFileReadStore({url:'test.json'});
myModel = new dijit.tree.TreeStoreModel({store: myStore,
query: {name:'*'}});
myTree = new dijit.Tree({id: "myTree", model: myModel,
childrenAttrs: ["children"]});
dojo.byId("container").appendChild(myTree.domNode);
myTree.startup();
dojo.connect(myTree, "onClick", this, function(e)
{
   // e is the actual item from the store
   console.log("--------------------onClick");
   console.dir(e);
   dojo.byId("onclickmessages").innerHTML += e.id +" "+e.name
   + " clicked\n";
});
dojo.connect(myTree, "onOpen", this, function(e)
{
   // e is the actual item from the store
  console.log("--------------------onOpen");
   console.dir(e);
   dojo.byId("onclickmessages").innerHTML += "onOpen for
   "+e.name+"\n";
});
dojo.connect(myTree, "onClose", this, function(e)
{
   // e is the actual item from the store
   console.log("--------------------onClose");
   console.dir(e);
   dojo.byId("onclickmessages").innerHTML += "onClose for
   "+e.name+"\n";
});
```

The HTML for the modified example now looks like this:

```
<body class="tundra">
        <h1 class="testTitle">Chapter 7 simple Tree Events test</h1>
   <br/>
   <div style="width:400px">
        <div id="container" style="width:200px;float:right;
         margin-left:10px;"></div>
      <textarea id="onclickmessages"></textarea>
   </div>
</body>
```

Now each click and open or close of the tree, will be recorded accordingly in the textarea.



Tree styling

For convenient styling, Tree has three hooks called `getIconClass`, `getLabelClass`, and `getLabel`:

- `getIconClass`(/* data store item */ item, /* boolean */ opened)
- `getLabelClass`(/* data store item */ item, /* boolean */ opened)
- `getLabel`(/* data store item */ item)

The first two return the name of the CSS class to use for the given item, and the last one returns a string to use as custom label for the given item in the tree.

The `getLabelClass` is the most simple to use, mostly used to change text color or boldness for certain items or branches in the tree.

The `getIconClass` function is a little bit trickier to implement. The standard implementation of the function in the tree itself look like this:

```
getIconClass: function(/*dojo.data.Item*/ item, /*Boolean*/ opened)
{
// summary: user overridable function to return CSS class name to
//display icon
      return (!item || this.model.mayHaveChildren(item)) ? (opened ?
      "dijitFolderOpened" : "dijitFolderClosed") : "dijitLeaf"
   },
```

It begins by checking that the item is an object at all, and then uses the `mayHaveChildren` function in its data model to see if it is a leaf node or not. Lastly, it chooses between two different classes to represent an open or closed folder, if the item isn't a leaf node.

Since this is very much the default behavior of the Tree, you might wind up presenting all the nodes the same way, opened or not, if you don't mimic the original behavior.

We add a couple of new classes to our example to illustrate basic styling:

```
<style>
   .foo
   {
      color: red;
  }
  .bar
   {
      color: blue;
   }
</style>
```

Then we override the three styling hook functions for our tree, like this:

```
myTree.getIconClass = function(item, opened)
      {
         console.log("--------------getIconClass called for
         '"+item.name+"'");
         return (!item || myTree.model.mayHaveChildren(item)) ?
         (opened ? "dijitFolderOpened" : "dijitFolderClosed") :
         "dijitLeaf"
      }
```

```
myTree.getLabelClass = function(item, opened)
{
   var rv = 'foo';
    if(item.name == 'Motorcycles')
    {
       rv = 'bar';
    }
    console.log("--------------getLabelClass called for
    '"+item.name+"', returns '"+rv+"'");
    return rv;
 }
 myTree.getLabel = function(item)
 {
    return "_"+item.name
 }
```

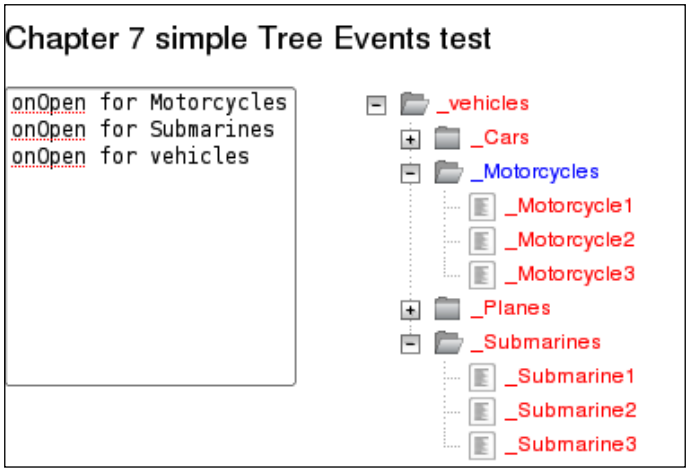
As always, if you try out the examples yourself, you will get a lot of extra information from firebug when testing.

What we've done above is to just mimic the standard icon behavior, with some changes to references to make things work properly, but we've really changed nothing other than writing a new function.

Then we decide to add an underscore to the names of each label, and to make sure that the `Motorcycles` label will be displayed in a blue color.

In a more realistic scenario, we might have set other properties on each data item, and used that to switch for certain label or classes. Still, the purpose of this example is to be as simple as possible.

The look of the tree has now been changed to this:



Tree drag and drop

At times, you want to be able to populate a tree with new nodes. The Tree widget utilizes Dojo's Drag and Drop support to do this. You can actually use anything as a drag and drop source, to make it into something that you can drop on (in this case) a tree.

An element can be made into a DnD (Drag and Drop) target or source. The Tree widget needs to be aware of what is dropped on it (or removed from it), so it can become 'DnD-alive' by just adding a property called `dndController`.

Probably you want to be able to control which items get dropped in which parts of a tree.

The Tree declares the following parameters for DnD operations (taken from source):

```
//parameters to pull off of the tree and pass on to the
//dndController as its params
dndParams: ["onDndDrop","itemCreator","onDndCancel",
"checkAcceptance", "checkItemAcceptance"],
```

- `onDndDrop` - This is a callback hook to manage dropped items on the tree.
- `ItemCreator` - This is a function which returns new items in the tree, when a draggable item is dropped on the tree. This separates the draggable items from the actual data item that will exist in the tree.
- `onDndCancel` - This gets called if you begin to drag an item onto the tree, but change your mind.
- `checkAcceptane` - This returns a boolean which controls if a draggable is allowed to be dropped on the tree at all.
- `checkItemAcceptance` - This allows for item/node level control of where things can be dropped. You might want a certain level only to allow draggables to be dropped.

Adding basic DnD support to our tree makes the following changes and additions to the code and stylesheets:

```
@import "../dojo/resources/dnd.css";
@import "../dojo/tests/dnd/dndDefault.css";

   myStore = new dojo.data.ItemFileWriteStore({url:'test.
   json'});
   myModel = new dijit.tree.ForestStoreModel({
      store: myStore,
      query: {name:'*'},
      rootLabel: 'vehicles'
```

```
});
function itemTreeCheckItemAcceptance(node,source)
{
   console.log("itemTreeCheckItemAcceptance called with
   "+node+" and "+source);
   var item = dijit.getEnclosingWidget(node).item;
   if (item && (item.root || myModel.mayHaveChildren(item)))
   {
      return true;
   }
   return false;
}
function dndAccept(source,nodes)
{
   return true;
}
myTree = new dijit.Tree({
   id: "myTree",
   model: myModel,
   childrenAttrs: ["children"],
   dndController: dijit._tree.dndSource,
   checkAcceptance: dndAccept,
   checkItemAcceptance: itemTreeCheckItemAcceptance
});
dojo.byId("container").appendChild(myTree.domNode);
myTree.startup();
```

In this example, we only implement the `checkItemAcceptance` and `checkAcceptance` with token code in them.

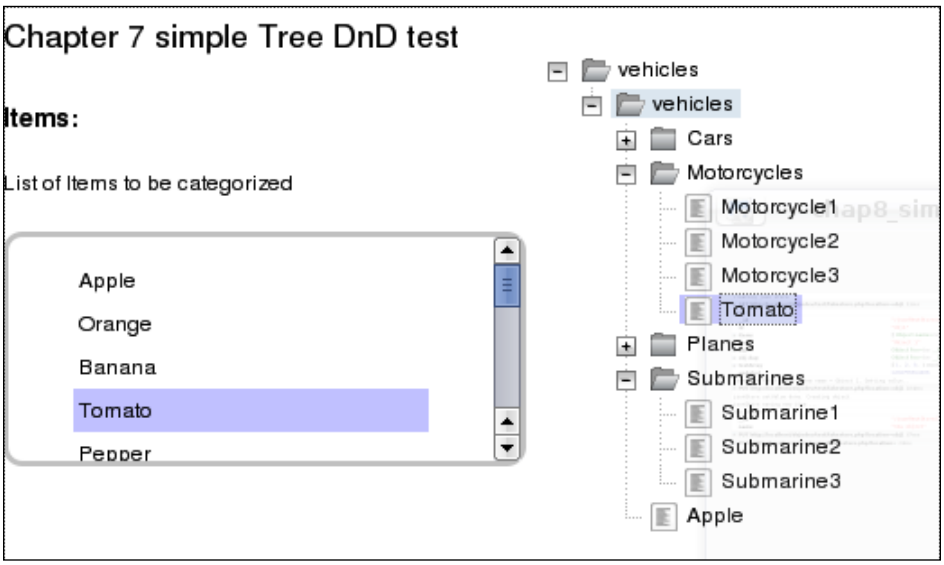
From the unit test for the tree DnD, we borrow a list of things that we can drag about:

```
<div>
   <h2>Items: </h2>
   <p>List of Items to be categorized<p>
   <div dojoType="dojo.dnd.Source" jsId="c2" class="container"
    style="height: 100px; overflow: auto">
   <div class="dojoDndItem" id="1001">Apple</div>
   <div class="dojoDndItem" id="1002">Orange</div>
   <div class="dojoDndItem" id="1003">Banana</div>
   <div class="dojoDndItem" id="1004">Tomato</div>
   <div class="dojoDndItem" id="1005">Pepper</div>
   <div class="dojoDndItem" id="1006">Wheat</div>
   <div class="dojoDndItem" id="1007">Corn</div>
   <div class="dojoDndItem" id="1008">Spinach</div>
```

```
        <div class="dojoDndItem" id="1009">Cucumber</div>
        <div class="dojoDndItem" id="1010">Carrot</div>
        <div class="dojoDndItem" id="1011">Potato</div>
        <div class="dojoDndItem" id="1012">Grape</div>
         <div class="dojoDndItem" id="1013">Lemon</div>
        <div class="dojoDndItem" id="1014">Lettuce</div>
        <div class="dojoDndItem" id="1015">Peanut</div>
    </div>
```

Note that one enclosing element is marked with the dojoType `dojo.dnd.Source`, which then contains any number of elements that have the class `dojoDndItem`, which uses the additional css files imported.

The resulting page looks like this:



What this picture does not show is the default icons that Dojo DnD brings to the table. As soon as you start dragging an item from the list, it gets an 'avatar' which by default is a copy of the element in the list with a green arrow icon on top of it. As the avatar is dragged onto a place where it cannot be dropped (judged by the check* functions), the icon changes to a red arrow.

Reading and writing with JsonRestStore

Most of the times a store is used just to slurp in data from a server or a prepared file in one go, whether it is JSON (as in ItemFileReadStore) or XML (as in XmlStore).

There are occasions where the dataset you're accessing is way too large to read in one go, and you need to send a filtering query to the server, or pagination and max count information to the server, or both.

Dojo's data stores can handle those scenarios as well. The QueryReadStore has ample support for sending specific queries to the server side and requesting specific pages of the result set, which can then be used together with the Grid, for instance.

Eventually, you'll want to use stores to write back data automatically as the widgets that the stores power change local data. Maybe you've added nodes dynamically to a tree, or used a standard Cell editor on a row in the Grid.

Either way, you'd want this to be as transparent as possible and 'just work', leaving you to see which bits plop in at the server side and write your server accordingly.

There's an excellent store called JsonRestStore which has been written for this very purpose. As the name implies, it uses REST (Representational State Transfer), which can be defined as 'Web Services done right'.

If you're not familiar with REST, it means that you use HTTP methods for different kinds of data manipulation at the server. `GET` retrieves an object, `PUT` changes the properties of an existing object, `DELETE` deletes it, and `POST` creates a new object.

Also, REST implies that the objects at the server can be accessed in a hierarchical manner, like a file tree. This means that if you have an object `foo`, which has a reference to an object `bar`, they can be accessed as `/foo` and `/foo/bar` repectively.

To better understand how the JsonRestStore works (and stores in general), I've set up a small example based on the existing unit tests for `dojox.data`.

What the example does is create a JsonRestStore out of the definition of an SMD file (See chapter 4), and then does some basic operations on the store, like creating new objects, modifying old, and saving.

The store itself is based on the ever-present `fakestore.php`, which is used for Dojo unit testing.

So the example will:

1. Create an SMD file which defines a method that can be found at a URL, its parameters.
2. Use that SMD file to create a `dojox.rpc.Service`.
3. Use that service in the creation of a JsonRestStore.
4. Fiddle around with some basic functions in the Read and Write `dojo.data` APIs.

Here's the SMD file used in the example:

```
{
   services: {
      jsonRestStore: {
         transport: "REST",
         envelope: "URL",
         target: "http://localhost/dojodev/test/fakestore.php",
         contentType:"application/json",
         parameters: [{name: "location", type: "string",
         optional: true}]
      }
   }
}
```

It defines just one service called `jsonRestStore` with the given transport types and the given URL.

The example client lives in just one script tag, and uses only the firebug console to display information.

```
<script language="JavaScript" someProperty="text/javascript">
      dojo.require("dojo.data.api.Read");
      dojo.require("dojox.rpc.Service");
      dojo.require("dojox.rpc.Rest");
      dojo.require("dojox.data.JsonRestStore");
      dojo.addOnLoad(function()
      {
         //console.log("creating SMD service");
         var testServices = new dojox.rpc.Service("test2.smd");
         console.log("creating json store");
         //var jsonStore = new dojox.data.PersevereStore
         ({target: "fakestore.php"});
         var jsonStore = new dojox.data.JsonRestStore
         ({service:testServices.jsonRestStore});
```

```
            jsonStore.fetch({query:"obj1",
               onComplete: function(item, request)
               {
                  console.log("item:");
                  console.dir(item);
                  console.log("jsonStore fetch query; item name =
                  "+ item.name+", Setting value..");
                  var now = new Date().getTime();
                  jsonStore.setValues(item, "items", []);
                  jsonStore.save();
                  console.log("jsonStore setValue done.
                  Creating object");
                  var newObject = jsonStore.newItem({"name":"new
                  object"}, {parent: item, attribute: 'items'});
                  console.log("jsonStore saving new item ...");
                  console.log("new object:");
                  console.dir(newObject);
                  jsonStore.save();
            }});
         });
      </script>
```

When the script is run, the firebug console shows the following:

```
creating json store
+ GET http://localhost/dojodev/test/fakestore.php?location=obj1 11ms
item:
  __id                                  "/jsonRestStore/obj1"
  id                                    "obj1"
+ items                                 [ Object name=new object ]
  name                                  "Object 1"
+ obj                                   Object foo=bar __id=/jsonRestStore/obj1["obj"]
+ obj dup                               Object foo=bar __id=/jsonRestStore/obj1["obj dup"]
+ testArray                             [ 1, 2, 3, 1 more... ]
  updated                               1202755814406
jsonStore fetch query; item name = Object 1, Setting value..
+ PUT http://localhost/dojodev/test/fakestore.php?location=obj1 104ms
jsonStore setValue done. Creating object
jsonStore saving new item ...
new object:
  __id                                  "/jsonRestStore/14507521243a8c08fd7e40f0"
  name                                  "new object"
+ PUT http://localhost/dojodev/test/fakestore.php?location=obj1 37ms
+ POST http://localhost/dojodev/test/fakestore.php?location= 13ms
```

The server we're using, fakestore, works by extracting the object location from the `GET` or `POST` parameters, and load files with the same names to server as objects. The results of operations are then stored in the PHP session for quick access.

Here's the server:

```
<?php
   $log = "";
   $logstart = getcwd();
   function logit($it)
   {
      //---------------------------------------------------------------
      global $log;
      global $logstart;
      $log = $log.$it;
      $fp2 = fopen($logstart."/meh.log", "a");
      $d = date("Y-m-d H:i:s");
      fwrite($fp2, $d.">".$it."\n");
      fclose($fp2);
      //---------------------------------------------------------------
   }
   session_start();
   header("Content-Type: " . ($_SERVER["CONTENT_TYPE"] ==
'application/json' ? 'application/json' : 'text/plain'));
   $fn = $_REQUEST['location'];
   if ($fn != "")
   {
   $m = $_SERVER["REQUEST_METHOD"];
   logit("fn = $fn. m = $m");
   switch ($m) {
      case "GET" :
         if (isset($_SESSION[$fn]))
         {
            print($_SESSION[$fn]);
         }
         else
         {
            $fh = fopen($fn, 'r');
            print(fread($fh, filesize($fn)));
            fclose($fh);
         }
         break;
      case "PUT" :
         $contents = file_get_contents('php://input');
         print($contents);
```

```
            $_SESSION[$fn]=$contents;
            break;
         case "POST" :
            if (isset($_SESSION[$fn]))
            {
               $old = $_SESSION[$fn];
            }
            else
            {
               $fh = fopen($fn, 'r');
               $old = fread($fh, filesize($fn));
               fclose($fh);
            }
            $contents = file_get_contents('php://input');
            $_SESSION[$fn]=$old . $contents;
            break;
         case "DELETE" :
            $_SESSION[$fn]="deleted";
            break;
      }
      }
   ?>
```

The server first sets the returning content-type to something reasonable, mimicking the client if it uses json. It then reads the system.variable `$_SERVER["REQUEST_METHOD"]`, which gives us the HTTP method used by the client, on which it then switches and opens files or writes from session variables as the case may be.

The file `obj1`, which is used in the `fetch()` call in the example, contains the following sample JSON:

```
{"id":"obj1","name":"Object 1","updated":1202755814406,"obj":{"foo":
 "bar"},"obj dup":{"$ref":"obj1.obj"},"testArray":[1,2,3,4]}
```

Overview of the Grid Dijit

The Grid is a recent addition to Dojo. In earlier versions of Dojo, there was a widget called SortableTable which created a sortable table out of an ordinary HTML table. It was very simple to use and sorely missed when the Grid first came out, due to all the additional knowledge needed to configure `dojo.data` correctly for the Grid.

The Grid supports lazy rendering of data, which means that it plucks only what is needed from the datastore, making good use of paging queries.

Visually, it supports custom rendering for every column (and cell), making it easy to support images and custom HTML.

Also, it is possible to define a certain type of editor for every column. This is so that the cells of a column might pop up a calendar widget when clicked, where others are inline editable textfields.

Also, all editors support writeback to the datastore, making the Grid widget an excellent basepiece for an administrative view of a service.

In recent iteration of Dojo 1.x, a HTMLStore has been created, so that you can create a data-store 'on the fly' using an existing table in the Markup, if you wish.

This is quite a simple first step for using the grid, and will be our first example.

A simple Grid

In the Grid unit test for multiple stores, one of the stores in such a HTMLStore. The store is very easy to use and needs only one property when created: the id of the table DOM element that it creates a data store out of.

The markup which the data store and the grid component use looks like this:

```
<span dojoType="dojox.data.HtmlStore"
   jsId="htmlStore" dataId="tableExample">
</span>
<span dojoType="dojox.grid.data.DojoData"
   jsId="dataModel6"
   rowsPerPage="5"
   store="htmlStore"
   query="{}">
</span>
<div id="grid6" dojoType="dojox.Grid" elasticView="2"
   model="dataModel6" structure="layoutHtmlTable">
</div>
```

Note that, as with the Tree widget, the Grid also has several different data models used to define additional meta information and filtering for the stores it's using. The query for the Grid is also empty, meaning that it will take everything in the store and render it.

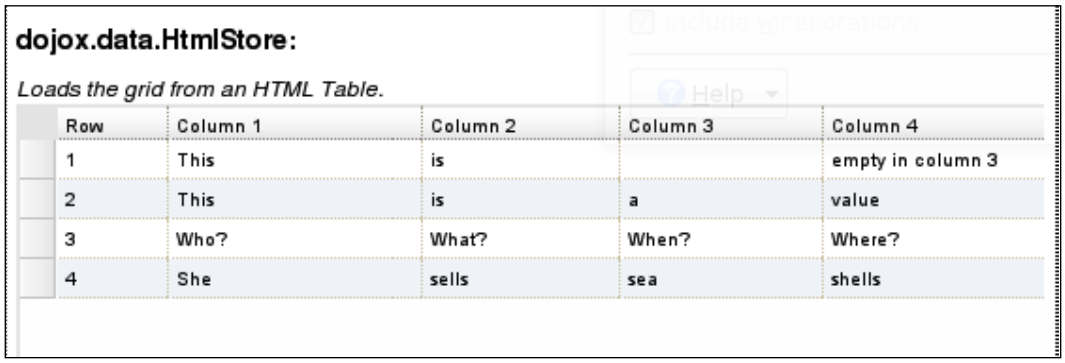
The different components making up a 'data chain' use a `jsId` property to find each other when defined in markup. This is not needed when creating a grid and related components programatically, since the actual JavaScript variable names will be used instead.

The table referred to in the store is defined like this:

```
<table id="tableExample" style="display: none;">
   <thead>
      <tr>
         <th>Column 1</th>
         <th>Column 2</th>
         <th>Column 3</th>
         <th>Column 4</th>
      </tr>
   </thead>
   <tbody>
      <tr>
         <td>This</td>
         <td>is</td>
         <td></td>
         <td>empty in column 3</td>
      </tr>
      <tr>
         <td>This</td>
         <td>is</td>
         <td>a</td>
         <td>value</td>
      </tr>
      <tr>
         <td>Who?</td>
         <td>What?</td>
         <td>When?</td>
         <td>Where?</td>
      </tr>
      <tr>
         <td>She</td>
         <td>sells</td>
         <td>sea</td>
         <td>shells</td>
      </tr>
   </tbody>
</table>
```

The important thing to remember here is that you need to define the table header (`<thead>`) which gives the data store information about which columns are defined.

Running the unit test gives the following Grid (for the above part of the unit test):



By default, the grid has client sorting supported, so clicking in a column will sort the tree after that column, descending or ascending, alternating with every click.

In the simple example above, the table defines both the structure of the table and its contents. More advanced uses of the Grid separates the two, for several reasons:

- You might want to change the presentation of data in a column. If the data contains a string **Yes**, you might want to render an image of a green ball in that cell instead.
- You might want to have multiple views of the same data.

This makes the Grid a little more complex. Normally, the Grid component uses one of several grid-specific data models to filter the data from the store, in very much the same way the Tree component is doing.

With Dojo 1.2, there's a new Grid version called DataGrid, which can talk directly to data stores and use Write and Notification APIs (if available). Actually, the old Grid component will be by the time this book is released, deprecated, so if you see references to `dojox.grid.Grid` in a page, it means it is done prior to Dojo 1.2 and uses internal data models between the store and the Grid.

Grid views

Layout views format the data of the grid. For tabular data which uses many columns, it is sometimes practical to 'fold' the column row into several rows. This can be particularly useful if you only have a certain allotment of screen real estate for your Grid.

So views are programmatically defined as arrays within an array, where an array defines one sub-row of the Grid's header. This sounds a bit complicated, but it isn't.

If we look at the unit test `test_grid_layout.html` in the `dojox/grid/tests/` source folder, a view is defined like this for one of the Grids:

```
var layout = [[
        {name: 'Column 1', field: 'col1'},
        {name: 'Column 2', field: 'col2'},
        {name: 'Column 3', field: 'col3'},
        {name: 'Column 4', field: 'col4', width: "150px"},
      {name: 'Column 5', field: 'col5'}
    ],[
      {name: 'Column 6', field: 'col6'},
      {name: 'Column 7', field: 'col7'},
       {name: 'Column 8'},
       {name: 'Column 9', field: 'col3', colSpan: 2}
    ]];
```

Looking at the test data, we see that a row is defined like this:

```
{ col1: "normal", col2: false, col3: "new", col4: 'But are not
followed by two hexadecimal', col5: 29.91, col6: 10, col7: false }
```

So you see that the `field` property of the view corresponds to the first named property in the data item to be interpreted as a row.

The Grid in the test is declared using the view like this (in markup):

```
<div id="grid1" dojoType="dojox.grid.DataGrid" store="test_store"
structure="layout" title="Tab 1"></div>
```

The final results looks like this:

Column 1	Column 2	Column 3	Column 4	Column 5
Column 6	Column 7	Column 8	Column 9	
normal	false	new	But are not followed by two hexadecimal	29.91
10	false		new	
important	false	new	Because a % sign always indicates	9.33
-5	false		new	
important	false	read	Signs can be selectively	19.34
0	true		read	

So you see that the layout view's two arrays correspond to the two sub-rows of the header row, and that the last field uses a `colSpan` property set to **2**, to span two rows.

If we only had (as is the usual case, admittedly) one single header row, we could have declared the view like this instead:

```
var layout = [[
         {name: 'Column 1', field: 'col1'},
         {name: 'Column 2', field: 'col2'},
         {name: 'Column 3', field: 'col3'},
         {name: 'Column 4', field: 'col4', width: "150px"},
         {name: 'Column 5', field: 'col5'}
      ]];
```

Here we are still using arrays within arrays, even in singularis.

There's quite a lot of work to be done in the layout of an array, and defining basic headers is only part of that.

It is possible to declare a specific `onStyleRow` custom function which will be called by the Grid for each row to be rendered.

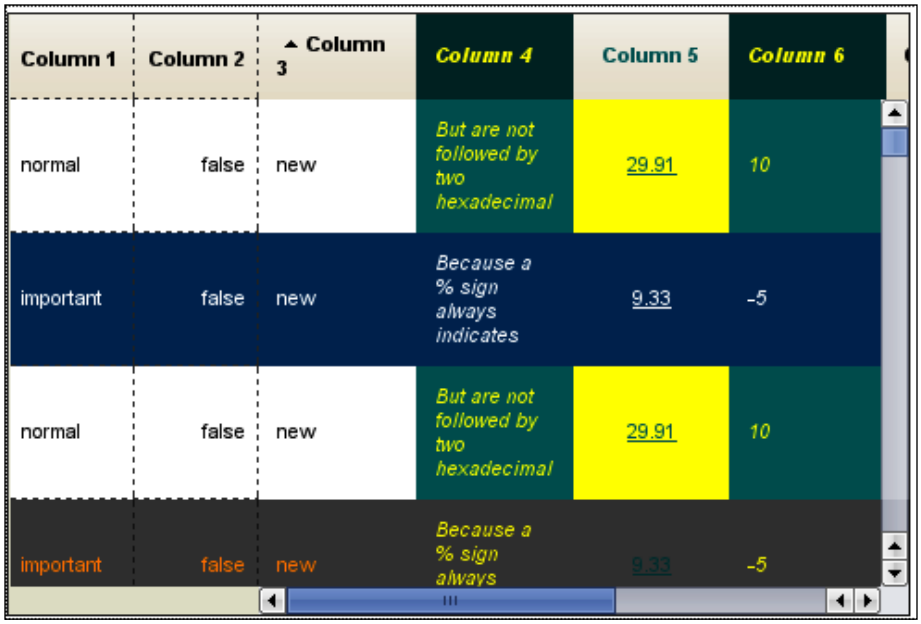
In the unit test `test_styling.html`, there's a good example on how to use this, along some special properties of the row arguments like `odd`, `index`, and `selected`. The example Grid is declared like this:

```
<div id="grid" jsid="grid" dojoType="dojox.grid.DataGrid"
 onStyleRow="onStyleRow" store="test_store" structure="structure">
</div>
```

The `onStyleRow` function itself is declared like this:

```
function onStyleRow(inRow)
      {
         with (inRow)
        {
            var i = index % 10;
            var special = (i > 2 && i < 6);
            if (odd)
               customStyles += ' color: orange;';
            if (selected)
               customClasses += ' selectedRow';
            if (special)
               customClasses += ' specialRow';
            if (over)
               customClasses += ' overRow';
            if (!over && !selected)
               dojox.grid.DataGrid.prototype.onStyleRow.apply(this,
               arguments);
         }
      }
```

The classes `selectedRow`, `specialRow`, and others are defined in the beginning of the example. Together the CSS classes and row styling function create a Grid that looks like this:



Grid custom rendering

Many times you want to control how data is displayed. Maybe you want to add a `%` to the end of a string, or you want to emphasize certain values in boldface. This is possible by defining a custom formatter for a column.

It's also possible to perform operations on the data itself before it is formatted (by your own formatter or by the Grid defaults).

The two function hooks are called `get` (for data manipulations) and `formatter` (for rendering) and are defined on each column in the view definition.

A small example of this might be in order:

```
var test_store = new dojo.data.ItemFileWriteStore({url:'test3.json'});
      function formatSpeed(value)
      {
         console.log("formatSpeed value = "+value);
         var rv = "<strong>low</strong>";
         if (value > 200) rv = "<em>medium</em>";
         if (value > 300) rv = "<u>high</u>";
```

```
            return rv;
         }
         function getSpeed(rowIndex, item)
         {
            if(item)
            {
               console.log("getSpeed rowIndex = "+rowIndex+",
               item = "+item);
               var value = test_store.getValue(item, "speed");
               return parseInt(value);
            }
         }
         var layout = [
            {name: 'Id', field: 'id'},
            {name: 'Name', field: 'name'},
            {name: 'Speed', field: 'speed', get:getSpeed,
            formatter: formatSpeed}
         ];
```

This small grid example reads from a ItemFileWriteStore (though we're not currently writing to it) and populates that with values from the file `test3.json` which looks like this:

```
{
   identifier: 'id',
   label: 'name',
   items: [ { id: '1', name:'Cars' , speed: '200'},
      { id: '2', name:'Motorcycles', speed: '300'},
      { id: '3', name:'Planes', speed: '1200'},
      { id: '4', name:'Submarines', speed: '200'},
      { id: '5', name:'Trains', speed: '300'} ]
}
```

The grid definition is here:

```
<div jsid="grid" id="grid" dojoType="dojox.grid.DataGrid" store="test_
store" query="{id: '*'}" structure="layout"></div>
```

What happens in the code is that for every value in the `speed` column, a call is first made to the `get` function. Since all JSON data are strings, the only thing it does is to convert the value to an integer.

After that, the `formatter` function is called, which exchanges the passed value for a custom HTML string.

The resulting Grid looks like this:

Id	Name	Speed
1	Cars	**low**
2	Motorcycles	*medium*
3	Planes	high
4	Submarines	**low**
5	Trains	*medium*

Grid sorting

Sorting actually takes place in the data store, and a store can have a `comparatorMap`, which is an associative array between column names and functions which takes care of sorting those columns.

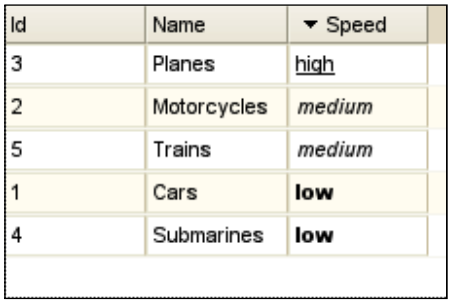
In our example, we do have actual numbers to work on, but custom sorting might be necessary when the information in the data store is in a string, but is to be sorted as a number. Maybe you need to strip certain characters, or sort Weekdays in order.

What you do then is to define a `comparatorMap`, and give it at least one entry for the column that needs custom sorting:

```
test_store.comparatorMap = {};
test_store.comparatorMap.speed = function(a, b)
{
   var ret = 0;
   var x = parseInt(a);
   var y = parseInt(b);
   if (x > y)
   {
      ret = 1;
   }
   if (x < y)
   {
      ret = -1;
   }
   console.log("comparator called for "+a+" and "+b+"
   returning "+ret);
   return ret;
};
```

The comparator function returns **0** if the two arguments are equal, **1** if the first one is 'bigger', and **-1** if the second one is. In this example, we do a very basic number sorting, which still is better than the default string sorting. Remember to always call `parseInt` or `parseFloat` on the arguments if you're going to do number comparison.

Our new custom sortable grid now looks like this:



Id	Name	Speed
3	Planes	high
2	Motorcycles	*medium*
5	Trains	*medium*
1	Cars	**low**
4	Submarines	**low**

The **Speed** column would otherwise had been sorted in alphabetical order, not on the custom strings, but on the strings representing the numbers!

Cell editing and styling

Once you manage to use the grid to present your information and sort it correctly, you want to be able to edit the information and hopefully write it back to the data store.

One of the problems with the Grid is that it has several ways of doing things. Up to this point, we've just defined a layout, but internally the Grid has something called 'structure' of which the layout is just one property. A structure can contain one or more view objects, which means that a grid can support more than one view of the same data store.

The structure can also be defined directly, just as with layout. A good example of this can be found in the unit test `test_edit_dijit.html` which defines two view structures. Let's dive down into the test:

```
// ================================================
// Custom formatter
// ================================================
formatMoney = function(inDatum)
{
   return isNaN(inDatum) ? '...' : '$' + parseFloat(inDatum).
   toFixed(2);
}
```

```
// =================================================
// Grid structure
// =================================================
statusCell = { field: 'col3', name: 'Status', styles: 'text-align:
center;', type: dojox.grid.cells.Select, options: [ "new", "read",
"replied" ] };
gridLayout = [{ width: '200px',
defaultCell: { width: 8, editable: true, styles:
'text-align: right;'  },
   cells: [ { name: 'Id', field: 'id', width: 3 },
   { name: 'Priority', field: 'col1', styles: 'text-align:
    center;', type: dojox.grid.cells.Select, options: ["normal",
    "note", "important"] },
   { name: 'Mark', width: 3, field: 'col2', styles: 'text-align:
   center;', type: dojox.grid.cells.Bool },
   statusCell,
   { name: 'Message', field: 'col4', styles: '', width: '100%' },
   { name: 'Amount', field: 'col5', formatter: formatMoney },
   { name: 'Amount', field: 'col6', formatter: formatMoney } ]},
   {
   defaultCell: { width: 4, editable: true, styles:
   'text-align: right;' },
   rows: [ { name: 'Mark', width: 3, field: 'col2', styles:
   'text-align: center;', type: dojox.grid.cells.Bool},
      statusCell,
      { name: 'Amount', field: 'col5', formatter: formatMoney},
      { name: 'Detail', value: 'Detail'}]
}];
```

This code snippet does the following:

- Defines a custom formatter, called `formatMoney`.
- Defines a structure object, which will later be passed to the grid constructor.
- The view objects contain the normal layout array under the property name of `cells`.
- It also contains a `defaultcell` property which declares the properties of the default cell, width, editability, and so on.
- In the cell layout array, a cell can have a type, for example `dojox.grid.cells.Select`, which defines which editor for data input, will be instantiated when the cell is double-clicked.

The following editor types are defined out of the box:

- `dojox.grid.cells.Cell` is a basic text input box.
- `dojox.grid.cells.Select` is a select options element which requires an additional `options` property.
- `dojox.grid.cells.Bool` defines a checkbox.
- `dojox.grid.cells.DateTextBox` pops up a calendar widget.
- `dojox.grid.cells._Widget` lets you define any form widget you want as a cell editor, using the additional property `editorClass` to define which dijit to use.

What happens when you edit a cell is that the data contained in the cell will be written back to the data store (given that your data store supports the Write API).

If you're using a complete store, like JsonRestStore, it will automatically arrange for the data to be written back to the server, but if you have to use a less integrated store, you can hook into the `onSet` hook of the notification API:

```
test_store.onSet = function (item, attribute, oldValue, newValue)
{
   // Use custom arrangements to write back data to the server.
}
```

Summary

- `dojo.data` is an API that implements a client-side data store which hides the intricacies of the actual service providing the data.
- `dojo.data` serves as an intermediary between widgets that consume data and services which provide them.
- The four APIs of `dojo.data` are Read, Write, Notification, and Identity.
- Some widgets provide their own data model objects for meta-information about the data in the connected store which relates to how they display it.
- Many stores accept a `query` parameter, which provides a way for the widget to sort or narrow down which objects it wants to read from the store, based on wildcards on object properties. Some stores allow for specific server-side queries to be passed to the service as well.
- The Tree widget supports drag-and-drop and communicates changes to its configuration down to the data store and vice versa.
- The Grid widget is highly configurable and is simple to layout, sort, and use special functions for formatting cells.

8
Real-World Dojo

In this chapter, we'll discuss things that are practical and useful in a business or organizational setting. The aim is to give you hints on deployment, building, and solutions to common problems actually being thrown upon you by customers or colleagues.

This chapter will cover the following:

- First, we'll explore the benefits of using resource domains for speeding up page loads and delivering mailable mockups in single files. This has been touched upon throughout the book, so it will be relatively brief, assuming you've caught the most important stuff by now.
- Then comes a section on how to use Dojo's build system to create your own custom builds. You will want to do that eventually in a production site, to lower the number of files being loaded by the browser and to pack the code reasonably well.
- Lastly, there's a couple of full examples that you can use as boilerplate code for your own escapades, or as panic, off-the-shelf, and good-enough code for situations when time is of essence.

Using AOL's and Google's CDN

AOL has been offering a Dojo cross-domain build for public use on its Content Delivery Network (CDN) or edge servers since march 2007 (and version 0.4.2).

Using a cross-domain build of Dojo means the following:

1. You don't need to manage Dojo versions yourself.
2. You don't need to unzip all of Dojo (which could be tens of megabytes) on your server, you just point to AOL's servers in your HTML page instead.

3 Your page will load much faster, because it's using a separate resource domain (AOL's), which lessens the load on your server and parallelizes loading.

4 You can actually do a mockup application with great functionality and mail it to your boss or client as a single HTML page, since no server is needed (on your part) to serve Dojo.

Sounds good? It is! Here's how to do it:

The instructions at the AOL Dojo page are very short and clear, and the page is located at `http://dev.aol.com/dojo`, but amount to the following:

When loading Dojo, use the following URL:

```
<script type="text/javascript" src="http://o.aolcdn.com/dojo/1.1.1/
dojo/dojo.xd.js"></script>
```

Of course, you should also use AOL to get the CSS stylesheets you need. A longer example from an actual page could look like this:

```
<link href="http://o.aolcdn.com/dojo/1.1/dojo/resources/dojo.css"
 rel="stylesheet" type="text/css"/>
<link href="http://o.aolcdn.com/dojo/1.1/dijit/themes/dijit.css"
 rel="stylesheet" type="text/css"/>
<link rel="stylesheet" type="text/css" href="http://o.aolcdn.com/
 dojo/1.1/dijit/themes/tundra/tundra.css"/>
<script type="text/javascript">
djConfig=
    {
     parseOnLoad: true,
     isDebug: true,

    }
</script>
<script src="http://o.aolcdn.com/dojo/1.1.1/dojo/dojo.xd.js"
 type="text/javascript"></script>
```

Google uses a slightly different approach. Since 2007, Google has been using a common loader for all Google Ajax APIs. This is also used when loading Dojo (and other Ajax libraries) from the Google CDN.

The URL to the homepage for the APIs is at `http://code.google.com/apis/ajaxlibs/`.

First, you must load the Google loader itself:

```
<script src="http://www.google.com/jsapi"></script><script>
```

And when loading Dojo, you add the following inside a script tag:

```
google.load("dojo", "1.1.1");
```

One interesting feature of Google's system is that you can give a generic version number, like "1", which will be interpreted as the highest available version that still begins with a "1". So, if you're using Dojo 1.2 at the moment, the page will then get 1.3 when available automatically.

Just as with AOL, you can (and should) load your stylesheets from Google:

```
<style type="text/css">
@import "http://ajax.googleapis.com/ajax/libs/dojo/1.1.1/dojo/
resources/dojo.css";
@import "http://ajax.googleapis.com/ajax/libs/dojo/1.1.1/dijit/themes/
soria/soria.css";
</style>
```

Note that no functionality exists for wildcarding stylesheet loading, in the same manner as for the actual Ajax library.

Dojo's build system and cross-domain loading

Dojo is as Ajax toolkits go quite large. One reason for this is that instead of creating free-standing 'plugins' or wholly separate third-party extensions (as is often the case with other popular Ajax toolkits), the contributors to Dojo have traditionally taken another route; donating the functionality directly to the Dojo foundation.

Partly, this is because Dojo itself was created out of such a process, being the result of the putting together of several of the original contributors' own toolkits, and partly because Dojo has a policy of being "the whole stack" instead of just core Ajax or just effects, etc.

However, it's hard to define what should not be part of "the whole stack", which is another reason Dojo has such a mind-bogglingly large amount of functionality.

This functionality naturally translates into a large number of files, mostly JavaScript, but also stylesheets and image files.

By default, loading the standard version of Dojo results in dozens, sometimes hundreds of files being loaded, one or two at a time, which results in quite a lot of loading time for the page in question.

Therefore, Dojo is endowed with its own build system, conveniently placed in the `util` folder parallel to the `dojo`, `dijit`, and `dojox` folders in the source distribution.

`util` also contain such gems as Dojo's own unit test system D.O.H. (see later), maven integration, documentation scripts, and shrinksafe for packing and stripping JavaScript files (which is also used by the build system).

The build system is pretty complicated and a complete discussion is outside the scope of this book, but it's very good to learn the basics of it, as we shall see.

What the build scripts do, is let you wrap several Dojo components into one big file, including any custom components you have made yourself while you're at it. Each has the following features:

- It lets you define one or more files, which will contain an arbitrary number of components that would otherwise have to be loaded one at a time.
- It can compress and/or strip comments from the JavaScript files.
- It can inline external HTML templates for widgets as internal strings.
- It can prepare the JavaScript files to be loaded cross-domain.
- It copies the resulting files (and mostly the rest of Dojo as well) to a release directory.

Each such resulting file is called a 'layer' and has its own definition in the configuration file for the build script.

The build scripts use Java version 1.4.2 or greater, so make sure that you have it installed.

There's a `build.bat` for people with operating systems they can't see the source code of, and there's `build.sh` for people who can. Building a standard version of Dojo is as easy as:

```
build.sh profile=standard action=release
```

The profile argument must match a file by the same name (+ ".profile.js") in the `profiles` folder under buildscripts. This file defines which things shall be included in the build, and how they are to be processed.

The `standard.js` file contains the following:

```
dependencies = {
   stripConsole: "normal",
   layers: [
      {
         name: "../dijit/dijit.js",
         dependencies: [
            "dijit.dijit"
         ]
```

```
        },
        {
           name: "../dijit/dijit-all.js",
           layerDependencies: ["../dijit/dijit.js"],
           dependencies: ["dijit.dijit-all"]
        },
        {
           name: "../dojox/off/offline.js",
           layerDependencies: [ ],
           dependencies: ["dojox.off.offline"]
        }
     ],
     prefixes: [
        [ "dijit", "../dijit" ],
        [ "dojox", "../dojox" ]
     ]
  }
```

What the profile defines is an object literal called `dependencies`. Under it can be found the properties `layers` and `dependencies`. Each `layer` defines a separate file and the assumption is that it should contain all the Dojo (and/or custom) scripts that a single page of your application requires—if you're writing a classical application that spans over several pages.

Each layer must have a `name` property defined, which will be the name of the script file that is generated by the buildscript. Other than that, a `dependencies` property declares which things to put in the file, and which `dojo.require` statements the layer can answer to.

For example, the first layer above defines all dijits to be put inside a file named `dijit-all.js`. This is probably overkill for any app, but since this file is only about 284KB compressed, it gives you all widgets under the `dijit` folder for the cost of just one network access.

The build system can also take any number of directories and 'normalize' their locations relative to the release directory.

In many examples in this book, I've put the directory containing example widgets parallel to the `dojo`, `dijit`, and `dojox` directories, to get automatic leverage from Dojo's loader. In real life, this might prove cumbersome and/or impossible due to deployment issues and misalignment between development and production environments.

So some of your custom Dojo classes might be several levels higher up in the file system than where Dojo is actually installed, some might be several levels deeper, but under Dojo, etc. You get the picture.

What you can do then is to define which directories are to be copied to the release folder, by including them in the `prefixes` variable. The build system not only copies the directories, but it also makes sure that any references to the 'old' file and object structure are rewritten to fit that of the release folder.

That means that you can have components in wildly different places, which reference each other (with `dojo.require`, etc.) using the same long names, and the build system will change the references to the new structure it creates in the release folder.

This build process always includes the `dojo` folder, which is why it is missing from the default profile.

To make your own `test` directory part of the Dojo build, copy the default profile file to a new one and add it following to the prefixes array.

```
prefixes: [
      [ "dijit", "../dijit" ],
      [ "dojox", "../dojox" ],
      ["test", "../test"]
   ]
```

When building a cross-domain build, the build system will insert logic into Dojo (and your own classes and widgets) that lets them be loaded using dynamically generated script tags. You will then use the following command to build:

```
build.sh profile=foo loader=xdomain action=release
```

Using unit testing with Dojo Objective Harness (DOH)

Dojo uses a unit testing framework called DOH (Dojo Objective Harness) for all its unit tests. Most of the functionalities of DOH is contained in the file `runner.js`, in the `util` directory.

DOH is actually separate from Dojo, and can even be used for unit testing server-side JavaScript, or just headlessly without a web page, using the Rhino JavaScript runtime, which is included in the Dojo source distribution and can be found in the `shrinksafe` folder parallel to the `DOH` folder under `util`.

The normal case is that you use Dojo in a client-side browser environment, and so can utilize the provided runner facilities.

In the `dojo` folder, there's a file called `runTests.html` that is responsible for running all tests in that folder. Similar files exist for other parts of the system, and it is a concise way to build your own test suites:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
    <head>
    <title>Dojo CORE and BASE D.O.H. Unit Test Runner</title>
    <meta http-equiv="REFRESH" content="0;url=../../util/doh/
     runner.html?testModule=dojo.tests.module"></HEAD>
    <BODY>
        Redirecting to D.O.H runner.
    </BODY>
</HTML>
```

The important parts of this file are:

- The redirect to the `runner.html` file
- The argument which declares the path to the module that contains the tests.

The `runner.html` file will build a nice, generic user interface for any test, and the module `dojo.tests.module` is resolved to the file `dojo/tests/module.js`, which looks like this:

```
dojo.provide("dojo.tests.module");
try{
   dojo.require("tests._base");
   dojo.require("tests.i18n");
   dojo.requireIf(dojo.isBrowser, "tests.back-hash");
   dojo.require("tests.cldr");
   dojo.require("tests.data");
   dojo.require("tests.date");
   dojo.require("tests.number");
   dojo.require("tests.currency");
   dojo.require("tests.AdapterRegistry");
   dojo.require("tests.io.script");
   dojo.require("tests.io.iframe");
   dojo.requireIf(dojo.isBrowser, "tests.rpc");
   dojo.require("tests.string");
   dojo.require("tests.behavior");
   dojo.require("tests.parser");
   dojo.require("tests.colors");
```

```
    dojo.requireIf(dojo.isBrowser,"tests.cookie");
    dojo.require("tests.fx");
    dojo.require("tests.DeferredList");
}catch(e){
    doh.debug(e);
}
```

This is also a typical strategy for the unit tests in Dojo; Each major module has a `base.js` file, which loads all the other unit tests under that module. Also, note that special care has been taken to ensure that unit tests can be run outside the browser-environment, server-side or otherwise, by using `dojo.isBrowser`.

OK, fine. But where are the actual tests? If we look inside the `date` test file; `dojo/tests/date.js`, we see a couple of simple tests being registered:

```
dojo.provide("tests.date");
dojo.require("dojo.date");
tests.register("tests.date.util",
    [
/* Informational Functions
 *************************/
function test_date_getDaysInMonth(t){
    // months other than February
    t.is(31, dojo.date.getDaysInMonth(new Date(2006,0,1)));
    t.is(31, dojo.date.getDaysInMonth(new Date(2006,2,1)));
    t.is(30, dojo.date.getDaysInMonth(new Date(2006,3,1)));
    ...
    // Februarys
    t.is(28, dojo.date.getDaysInMonth(new Date(2006,1,1)));
    t.is(29, dojo.date.getDaysInMonth(new Date(2004,1,1)));
    ...
},
function test_date_isLeapYear(t){
    t.f(dojo.date.isLeapYear(new Date(2006,0,1)));
    t.t(dojo.date.isLeapYear(new Date(2004,0,1)));
    ...
},
...
```

Note that the unit tests are just Dojo classes, just like widgets, so that they can be referenced and invoked by scripts or pages in the usual manner.

All tests also must require in the functionality they need to do the test, just like a regular widget.

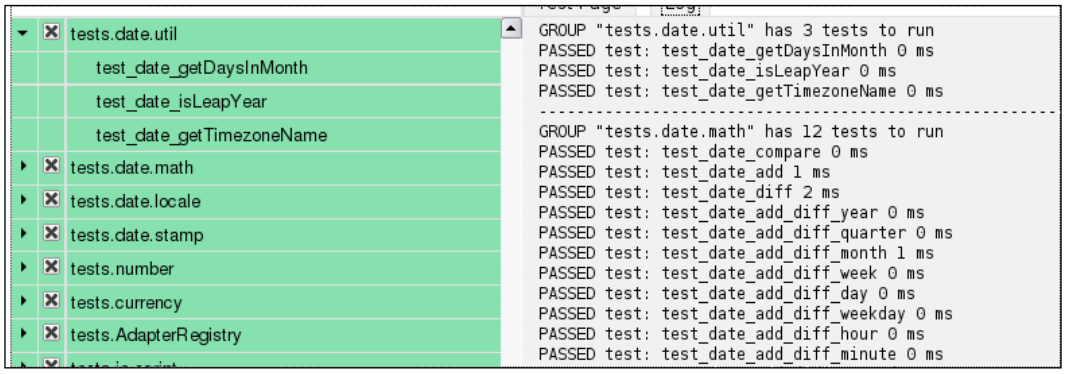
The test object is provided by the runner object, which loads the tests, and the common thing to do is to use the `test.register` function with two arguments:

- The name of the test (`tests.date.util`, in the example)
- An array containing one or more tests

Each test is a function that returns a Boolean. The function takes as argument a reference to the test object, where a couple of handy functions are provided for testing purposes:

- **assertEqual** tests if two values are equal
- **assertTrue** tests if the argument evaluates to true
- **assertFalse** tests if the argument evaluates to false

When the test runner has run, it displays the following output:



Here follow some practical examples of how to use Dojo to create small, specific widgets or applications, including server-side code. The ambition here is to let you copy-paste these examples into your own servers and play around with them as necessary.

Example 1: Site navigation scaffolding

Often when you have an idea or get an assignment, you want to get started with a simple site design that you can fill up with the stuff you need. It has saved me a lot of time to have a general template that can be reused for different purposes.

Dojo has a lot of very versatile widgets that make a scaffolding very simple, but taking it one step further by creating a generic 'site' scaffolding reduces the time to create a quick mockup even further.

The general idea is to create a Dijit that is a 'site in a can', which contains a simple menu which flips between parts of the 'site'. In essence, it is a Dijit that combines a simple navigational menu with a StackContainer.

The content of the 'site' is then created in a number of separate HTML files, which contain snippets of markup that define the content for just one 'page' of the site.

More specifically, this means that you get the scaffolding and navigation for a simple site, where the actual content can be created separately and swapped quite easily.

The basic idea is to be able to pass the widget/Dijit an object full of name-value pairs, where each pair consists of a menu name and a URL for the content. This is the test file that loads the widget:

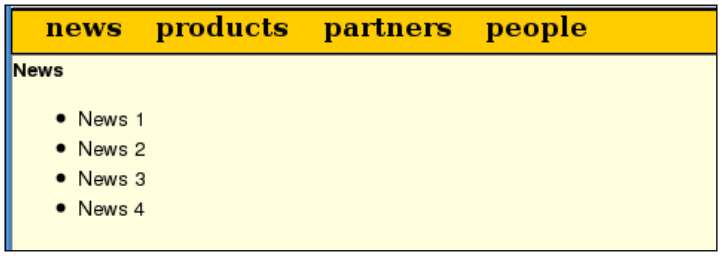
```
<html>
    <head>
        <link href="scaffold/templates/main.css" rel="stylesheet"
         type="text/css"/>
        <link href="../dojo/resources/dojo.css" rel="stylesheet"
         type="text/css"/>
        <link href="../dijit/themes/dijit.css" rel="stylesheet"
         type="text/css"/>
        <link href="../dijit/themes/tundra/tundra.css"
         rel="stylesheet" type="text/css" />
        <title>Test Scaffolding</title>
        <script type="text/javascript">
        djConfig=
        {
        parseOnLoad: true,
        isDebug: true,
        modulePaths: {scaffold: "../test/scaffold"}
        }
        </script>
        <script src="../dojo/dojo.js"  type="text/javascript">
        </script>
        <script type="text/javascript">
            /* all dojo.require statements go here */
            dojo.require("dojo.parser");
            dojo.require("scaffold.main");
        </script>
    </head>
  <body class="tundra">
    <div dojoType="scaffold.main" style="height:110px;"
```

```
        content="{'news': 'http://localhost/dojodev/test/scaffold/
        content/news.html', 'products': 'scaffold/content/products.html',
        'partners': 'scaffold/content/partners.html', 'people' :
        'scaffold/content/people.html'}">
      </div>
    </body>
  </html>
```

There's only one widget required and referenced above, and that's `scaffold.main`. The path to the scaffold module is declared in the `djConfig` variable's `modulePaths` property. The test file is assumed to be put in a `test` directory parallel to `dojo`, `dijit`, etc.

Under the `test` directory, lies the `scaffold` folder, which contains the file `main.js` and a directory named `templates`, which in turn contains the HTML template of the widget and CSS classes.

Here's a screenshot of the test page in action:



The file `news.html` contains only the following:

```
  <div class="scaffolding_content">
    <h1>News!</h1>
    <ul>
      <li>News 1</li>
      <li>News 2</li>
      <li>News 3</li>
      <li>News 4</li>
      <li>News 5</li>
    </ul>
  </div>
```

This is just basic HTML to create the list shown. What the scaffold widget does is to create the menu header out of the key names of the provided argument object, and the create a ContentPane for each of the content URLs, which is then added to an existing StackContainer.

The `file main.js` implements all functionality of the widget. An idea for further development would be to separate out the menu-building code and maybe add a possibility to create vertical menus as well as multi-level menus.

Let's first look at the template that the widget uses:

```
<div>
  <div dojoAttachPoint="mymenu" class="scaffold_menu">
  </div>
  <div dojoType="dijit.layout.StackContainer" dojoAttachPoint=
   "mystack" class="scaffold_main">
  </div>
</div>
```

The attachpoint for `mymenu` will resolve to the DOM node in the template, for use when adding new 'span' elements for menu items, and the attachpoint for `mystack` will instead refer to the StackContainer dijit.

Let's look at the code for `main.js` that creates the menu and ContentPanes as well as a lot of event handlers:

```
dojo.provide("scaffold.main");
dojo.require("dijit._Templated");
dojo.require("dijit._Widget");
dojo.require("dijit.layout.ContentPane");
dojo.require("dijit.layout.StackContainer");
dojo.require("dojox.fx");
dojo.require("dojox.fx.easing");
dojo.declare("scaffold.main", [  dijit._Widget, dijit._Templated ],
{
  templatePath: dojo.moduleUrl("scaffold","templates/main.html"),
  widgetsInTemplate: true,
  content: {}, // name - url pairs for menu building
  lastitem: "",
  constructor: function()
  {
    console.log("constructor for scaffold.main called");
  },
  postCreate: function()
  {
    console.log("postCreate for scaffold.main called");
    for(var k in this.content)
    {
```

```
var v = this.content[k];
console.log("Building menu for item "+k+" -> "+v);
var item = dojo.doc.createElement('span');
item.innerHTML = k; // k is the key name of the current content,
//like 'news', 'partners', etc.
dojo.addClass(item, "scaffold_menu_item"); // Sprinkle on a nice
//CSS class
(dojo.hitch(this, function(item, v)  // Create closure to save
//currently looped item and value (href)
{
  // Create ContentPane for each url
  var div = dojo.doc.createElement('div');
  dojo.addClass(div, "scaffold_cp");
  console.log("Creating content pane title "+k+" href "+v);
  var cp = new dijit.layout.ContentPane({href: v, preload: true,
  title: k}, div);
  this.mystack.addChild(cp); // this.mystack is an automatic
  //reference to the StackContainer dijit because of its
  //dojoAttachPoint
  if(!this.selectedpane) //If this is the first time (which also
  //will correspond to the first loaded and shown contentpane)
  {
    this.selectedpane = cp; //Set the current selected pane
    //as cp
    this.highlight(item); // Add a first highlight for the first
    //item in the menu.
    this.lastitem = item; // Note that this item is to be
    //lowlighted when the next one gets highlighted
  }
  dojo.connect(item, "onclick", this, function(e)
  {
    console.log("menu item "+item+" clicked. Stack "+this.
    mystack+" child "+cp+" selected");
    if (this.lastitem)
    {
      this.lowlight(this.lastitem);
    }
    this.lastitem = item;
    this.selectedpane = cp;
    this.mystack.selectChild(cp);
  });
  dojo.connect(item, "onmouseover", this, function(e)
  {
    this.highlight(item);
  });
```

```
            dojo.connect(item, "onmouseout", this, function(e)
            {
              if(this.selectedpane != cp) // Let higlight stand if this
              //contentpane is selected and currently shown
              {
                this.lowlight(item);
              }
            });
          }))(item, v);
          this.mymenu.appendChild(item, v);
        }
      },
      highlight: function(item)
      {
        dojo.animateProperty(
        {
                node: item,
                delay: 0,
                duration:400,
                easing:dojo.fx.easing.quartIn,  // Check out which other
                //animations exist in the file dojo/fx/easing.js
                properties:{
                backgroundColor:{ end:"#ffffff" }  // See which properties
                //can be animated in the file dojox/fx/style.js
                }
        }).play();
      },
      lowlight: function(item)
      {
        dojo.animateProperty(
        {
                node: item,
                delay: 0,
                duration:400,
                easing:dojo.fx.easing.quartOut,
                properties:{
                backgroundColor:{ end:"#ffcc00" }
                }
        }).play();
      }
    });
```

If you read through the preceding code, the comments will give you references to other files and functions you can use to change the behavior of the widget. The outline of the widget's functionality is as follows:

- In the `postCreate` function, which is called by Dojo after the widget is created, it reads the `content` property, which is passed to it from the HTML property of the same name, in the page where the widget is used. The `content` variable will contain an array.
- Each element of the `content` array has a name and a value. The name will be the string that is presented in the menu item, and the value is a string representing the URL to the page to show when the menu item is selected.
- Inside the loop, a closure is created to save the current key/value variables state (see below for detailed explanation), and a new ContentPane Dijit is dynamically created and inserted under the existing StackContainer. When the menu item is clicked (selected), it will send a message to the StackContainer to switch ContentPane to that which was associated with the menu item.
- Each menu item gets a couple of handlers for animating the highlighting when the mouse cursor hovers over them. By using the `dojo.animateProperty` function, effects like background color change can be 'eased' in, and two functions called `highlight` and `lowlight` manage the effects of smoothly changing background for a menu item.

When a menu item is clicked, it retains its highlight, even on mouseout, until another item is clicked:



The CSS file used in the example looks like this:

```
.scaffold_menu
{
  background:       #ffcc00;
  border:           1px solid black;
  height:           25px;
}
.scaffold_menu_item
{
  color:            black;
  font-family:      verdana;
  font-weight:      bold;
  margin-left:      20px;
  font-size:        16;
}
.scaffold_highlight
{
  background:       white;
}
.scaffold_main
{
  background:       white;
  border:           1px solid grey;
  height:           400px;
}
.scaffold_cp
{
  height:           400px;
  width:            400px;
  background:       lightyellow;
}
.scaffolding_content
{
  margin:           30px;
}
```

An interesting feature I use here is something you might want to use in your own applications, if you're not using it already. Inside the loop that hooks up events to each menu item, an anonymous function is created and called (it doesn't actually have to be redeclared for each loop, but it is just laziness on my part).

The basic structure is this:

```
(dojo.hitch(this, function(item, v)
{
  ....
 dojo.connect(item, "onclick", this, function(e)
  {
    console.log("menu item "+item+" clicked. Stack "+this.
    mystack+" child "+cp+" selected");
    if (this.lastitem)
    {
      this.lowlight(this.lastitem);
    }
    this.lastitem = item;
    this.selectedpane = cp;
    this.mystack.selectChild(cp);

  });    ....
}))(item, v);
```

You could write this function without using `dojo.hitch`, but then you would have another `this` or context than what you'd expect and wouldn't be able to access widget properties or functions, should the need arise.

So `dojo.hitch` just fixes so that the `this` in the function, when called, will be set to that of the first argument.

The interesting part is that we wrap the anonymous function in parentheses for delineation purposes and that we then call the function immediately afterwards, again anonymously.

As you remember from earlier chapters, all function calls in JavaScript generate a closure, in which variables will remain even after they have 'returned'. Also, closures inherit properties from each other, so the item argument in the anonymous function's closure (or the specific value of that item) will be inherited into the event handler function defined as part of the argument to `dojo.connect`.

So even though this function is created and called inside a loop, it retains the specific variable values of the loop count it was declared in, and those values will be saved inside its closure long after the loop has finished, and will be available to the event handler function when the user clicks on the menu item.

You don't actually need to understand what's going on here. Just copy the code to your own things, change as necessary, and eventually you'll grok it.

Example 2: Multi-charting application

Often, when discussing a project with customers, the need to have a customer-rearrangeable page comes up. Since Dojo recently gained the wonderful GridContainer (not to be confused in any way with the Grid or DataGrid), this is relatively simple.

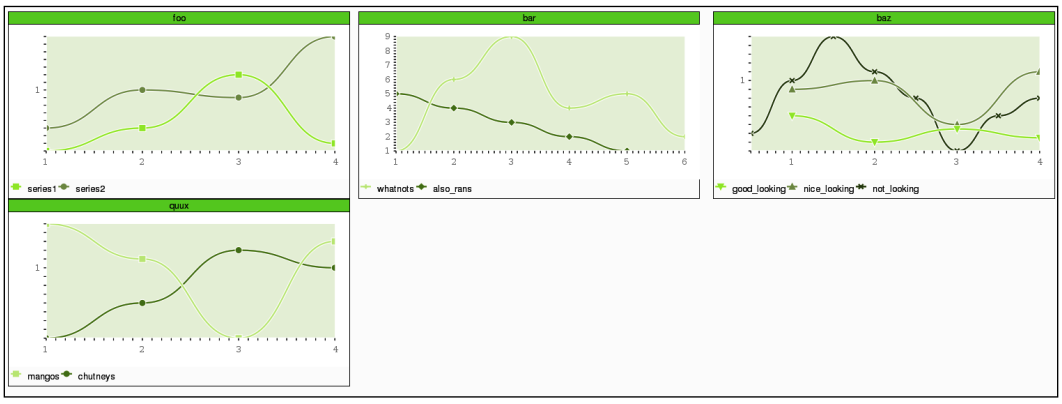
Charting in Dojo is also something that gets asked about a lot, especially how it compares with FusionCharts, etc. A couple of times I've been discussing having charts in some kind of administrative interface that are rearrangeable in an iGoogle / GridContainer fashion.

What this example shows is both how simple it is to create your own custom widgets even when creating your own interactive charts. Charting has its own widget, which can read a data source, but sometimes you want to do everything yourself to make sure you have control of what happens (read: understand what you're doing).

Also, the example uses a very simple, yet powerful technique for separating the work of the client team from that of the server team: creating local mock-server files, which are used when loading data, which in this case means data for the charts.

Later, when the client code works perfectly, the mock-server text files can be sent to the server-side developers, and will become a contractual protocol, which serves the dual purpose of hiding client internals from the server development team and vice versa.

The demo consists code-wise of two Dijits: one main Dijit, which creates the GridContainer, and a chart Dijit, which gets created one at a time and added to the main Dijit, for each provided chart.



Each chart has a simple template, which looks like this:

```
<div class="multichart_chart_div">
  <div dojoAttachPoint="mytitle" class="multichart_title">${name}</
div>
  <div dojoAttachPoint="mychart" class="multichart_chart"></div>
  <div dojoAttachPoint="mylegend" class="multichart_legend"></div>
</div>
```

Note also the use of `${name}`, which automatically inserts the contents of the chart Dijit's `name` property on creation.

The CSS file for the sample demo declares some simple classes, which give each chart a border and a title bar with centered text:

```
.multichart_title
{
  height: 15px;
  border-bottom: 1px solid black;
  text-align: center;
  background: #90de52;
}
.multichart_chart
{
  height:200px;
  position: relative;
}
.multichart_chart_div
{
  position: relative;
  border: 1px solid black;
}
.multichart_legend
{
  height: 30px;
}
```

The code for the chart itself comes here:

```
dojo.provide("multichart.chart");
dojo.require("dijit._Templated");
dojo.require("dijit._Widget");
dojo.require("dojox.charting.Chart2D");
dojo.require("dojox.charting.themes.PlotKit.green");
```

```
dojo.require("dojox.charting.action2d.Highlight");
dojo.require("dojox.charting.action2d.Magnify");
dojo.require("dojox.charting.action2d.Tooltip");

dojo.require("dojox.charting.widget.Legend");

dojo.require("dojo.colors");
dojo.require("dojo.fx");
dojo.require("dojo.fx.easing");

dojo.declare("multichart.chart", [  dijit._Widget, dijit._Templated ],
{
  templatePath: dojo.moduleUrl("multichart","templates/chart.html"),
  widgetsInTemplate: true,
  url: "",// The url to use (on same server) to load json that defines
          //data series to plot for this chart
  name: "",// The name

  constructor: function()
  {
    console.log("constructor for multichart.chart called");
  },

  postCreate: function()
  {
    console.log("postcreate for multichart.chart called. url =
     "+this.url);
    dojo.xhrGet({// If you need to load json cross-domain, you can use
                 //dojo.io.script.get instead
    url: this.url,
    handleAs: 'json',
    load: dojo.hitch(this, function(result)
    {
      console.log("Drawing chart "+this.name+", result is "+result);
      var chart1 = new dojox.charting.Chart2D(this.mychart);
      this.chart = chart1;
      chart1.setTheme(dojox.charting.themes.PlotKit.green);
      chart1.addPlot("default", {type: "Default", lines: true,
      markers: true, tension:2});
      chart1.addAxis("x", {majorTick: {stroke: "black", length: 3},
      minorTick: {stroke: "gray", length: 3}});
      chart1.addAxis("y", {vertical: true, majorTick: {stroke:
      "black", length: 3}, minorTick: {stroke: "gray", length: 3}});
      for(var s in result.series)
      {
        var serie = result.series[s];
        chart1.addSeries(s, serie);
```

```
        }
        var anim1a = new dojox.charting.action2d.Magnify(chart1,
        "default");// Add magnify animation for datapoints
        var anim1b = new dojox.charting.action2d.Tooltip(chart1,
        "default");// Add tooltip action for datapoints
        chart1.render();
        var legend1 = new dojox.charting.widget.Legend({chart: chart1,
        horizontal: true}, this.mylegend);
      })});
    }
});
```

If you're using Firefox 3.x, you might want to create an symbolic link to your local `dojo` directory to your local web server, since it otherwise will complain about security issues loading files using Ajax.

When this Dijit gets created, it does the following (as usual in the `postCreate` function called by Dojo upon creation):

- It uses the provided `url` property and uses that string in an Ajax call, using `dojo.xhrGet` to read the file that contains the data for the chart.
- When the data has loaded, the `load` function is called, which creates the actual chart.
- The specific options and function calls that can be used to create a chart can vary a great deal. The line chart created in this example is very basic and the detailed study of `dojox.charting` is outside the scope of this chapter.
- The data loaded from the Ajax called will be placed in a variable called (in this example) `result`. A loop is created, which takes each data point in the result object and adds it as a 'data series'—an array of data points to be plotted in the chart.
- A couple of chart features, such as a legend that lists the names and symbols used in each serie, and animations for mousovers are also added.

The charting example is taken from one of the standard tests in the `dojox.charting` test directory (`dojox/charting/test`). Please try different themes, and other settings.

The main component, or Dijit, for the project is fairly simple. It accepts the properties being set from the test web page, and creates a `multichart.chart` Dijit for each, placing them in the right place in the GridContainer.

```
dojo.provide("multichart.main");

dojo.require("dijit._Templated");
dojo.require("dijit._Widget");
```

```
dojo.require("dijit.layout.ContentPane");
dojo.require("dojox.layout.GridContainer");
dojo.require("multichart.chart");
dojo.declare("multichart.main", [  dijit._Widget, dijit._Templated ],
{
  templatePath: dojo.moduleUrl("multichart","templates/main.html"),
  widgetsInTemplate: true,
  content: {}, // name - url pairs to make draggable charts out of.
               //must be passed from calling script / page
  columns: 3,  // How many columns we want to have in the
               //GridContainer
  constructor: function()
  {
    console.log("constructor for multichart.main called");
  },
  postCreate: function()
  {
    console.log("postCreate for multichart.main called mycontainer is
    "+this.test);
    this.gc = new dojox.layout.GridContainer(
      {
        nbZones: this.columns,
        opacity: 0.7,           // For avatars of dragged components
        allowAutoScroll: true,
        hasResizableColumns: false,
        isAutoOrganized : true,
        withHandles: true,
        acceptTypes: ["multichart.chart"]  // This property must
        //always be present, and can take any kind of widget,
        //including your own
      }, this.test);
    this.gc.startup(); // When creating some dojox widgets
    //programmtaically, you must manually call the startup() function,
    //to make sure it's properly intialized
    var i = 0; // Count for which column we'll place each chart in
    var p = 0; // Count for which row we'll place a chart in
    for(var name in this.content)
    {
      var url = this.content[name];
      var chart = new multichart.chart({name: name, url:url});
    // Create a custom chart with the given name and url
      console.log("adding chart "+chart+" to zone "+i+", place "+p);
      this.gc.addService(chart, i, p); // Add the chart to the
      //GridContainer (This function will be called addChild in the
      //future, for conformance with similar containers.)
      i++;
```

```
        if (i > this.columns-1)  // Go to next row if we've hit the end
                                 //of the columns
        {
          i = 0;
          p++;
        }
      }
    }
});
```

The number of columns that a GridContainer supports must be specified, but can be changed at run time. If it is not specified by the properties, we set `3` as default.

The widget works in the following way:

1. It creates a new GridContainer programmatically.
2. It then loops over all provided items in the `content` variable, passed from the HTML page in which the widget was used.
3. It then creates a new `multichart.chart` widget (explained above) for each item in `content`, and passes an argument array containing the name and URL to use for chart `name` and chart data `url` respectively.
4. Additional logic (the `i` and `p` variables) keeps track of which row and column each chart is created at, so that the `addService` call to the GridContainer does not specify a column that does not exist.
5. The `multichart.chart` widget is added to the GridContainer.

What is very gratifying in working with Dojo in this manner, is that the template for a component gets inserted at the right place and initialized correctly regardless of whether you have a 'static' component in the template of an enclosing component, or you (as in this example) add it on the fly.

The only caveat is that the DOM node of the component must actually be inserted in the document in the browser. Otherwise, the `dojox.gfx` logic gets no foothold and breaks in strange and mysterious ways.

The calling test page, which uses the outermost component, looks like this:

```
<html>
    <head>
        <link href="multichart/templates/main.css" rel="stylesheet"
         type="text/css"/>
        <link href="../dojo/resources/dojo.css" rel="stylesheet"
         type="text/css"/>
        <link href="../dijit/themes/dijit.css" rel="stylesheet"
         type="text/css"/>
```

```
    <link href="../dijit/themes/tundra/tundra.css" rel="stylesheet"
      type="text/css" />
    <link href="../dojox/layout/resources/GridContainer.css"
     rel="stylesheet">
    <link href="../dojox/layout/resources/DndGridContainer.css"
     rel="stylesheet">
        <title>Test Multichart</title>
        <script type="text/javascript">
        djConfig=
        {
                parseOnLoad: true,
                isDebug: true,
                //debugAtAllCosts: true,
                modulePaths: {multichart: "../test/multichart"}
        }
        </script>
        <script src="../dojo/dojo.js"  type="text/javascript">
        </script>
        <script type="text/javascript">
            /* all dojo.require statements go here */
            dojo.require("dojo.parser");
            dojo.require("multichart.main");
        </script>
    </head>
  <body class="tundra">
    <div dojoType="multichart.main" content="{'foo': 'multichart/
     mock-server/foo.txt',
   'bar': 'multichart/mock-server/bar.txt',
   'baz': 'multichart/mock-server/baz.txt',
   'quux': 'multichart/mock-server/quux.txt' }">
    </div>
  </body>
</html>
```

Note that just one tag adds the complete multi-chart functionality in the page: the line that reads `div dojoType="multichart.main"`. The `content` property will be automatically passed along to the class variable of the `multichart.chart` widget with the same name, where it will be used to create a series of charts using the provided URLs.

Finally, the data files, which are located in a subdirectory, as shown in the HTML file, have a simple JSON text content (`foo.txt` and `baz.txt` shown):

`foo.txt`:

```
{
  series:
  {
    series1: [{x: 1, y: 0.2}, {x: 2, y: 0.5}, {x: 3, y: 1.2},
    {x: 4, y: 0.3}],
    series2: [{x: 1, y: 0.5}, {x: 2, y: 1.0}, {x: 3, y: 0.9},
    {x: 4, y: 1.7}]
  }
}
```

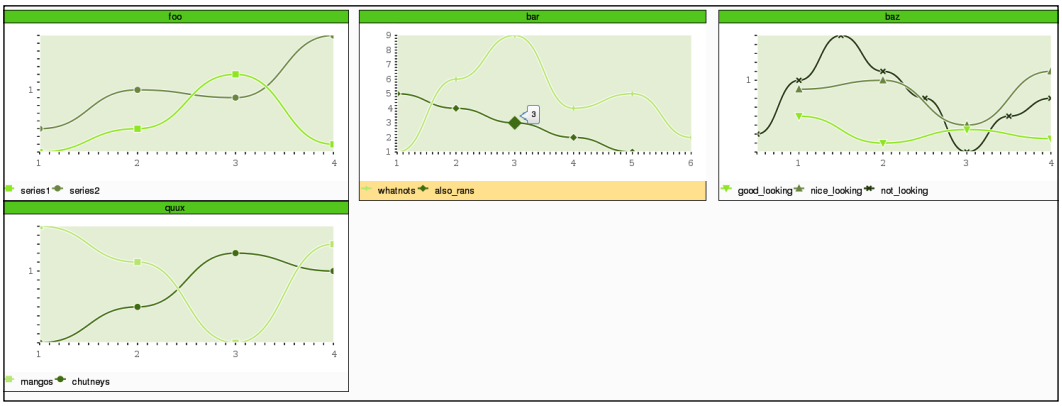
`baz.txt`:

```
{
  series:
  {
    good_looking: [{x: 1, y: 0.6}, {x: 2, y: 0.3}, {x: 3, y: 0.45},
    {x: 4, y: 0.35}],
    nice_looking: [{x: 1, y: 0.9}, {x: 2, y: 1.0}, {x: 3, y: 0.5},
    {x: 4, y: 1.1}],
    not_looking: [{x: 0.5, y: 0.4}, {x: 1, y: 1.0}, {x: 1.5, y: 1.5},
    {x: 2, y: 1.1}, {x:2.5, y:0.8}, {x:3, y:0.2}, {x:3.5, y:0.6},
    {x:4, y:0.8}]
  }
}
```

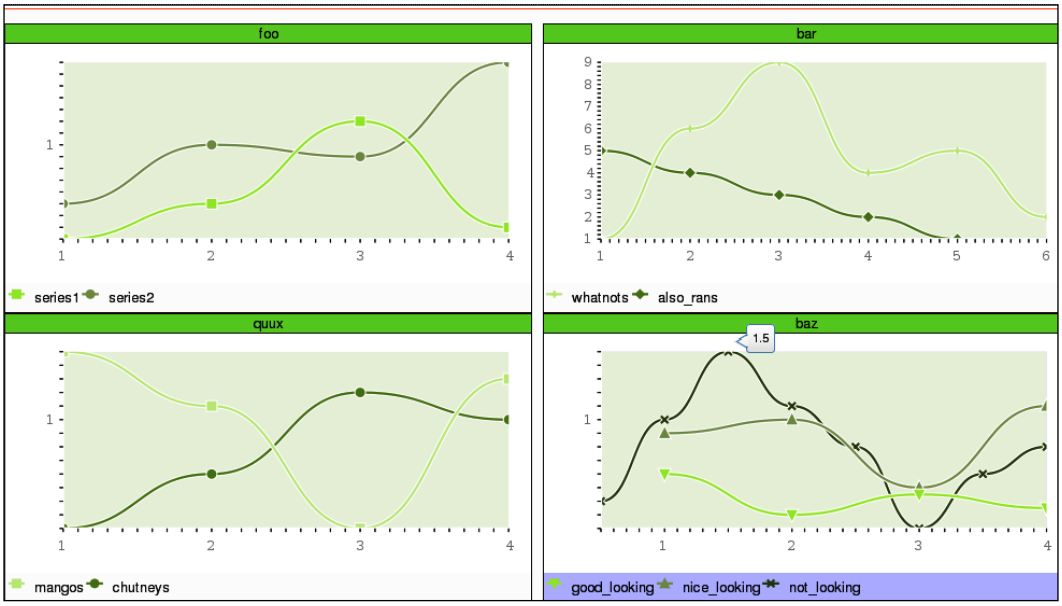
The trick of getting the data out of the files relies in the `postCreate` function in `chart.js`, which instructs Dojo to evaluate the text read from the (mock) server into a JavaScript, called `result`.

```
dojo.xhrGet({
    url: this.url,
    handleAs: 'json',
    load: dojo.hitch(this, function(result)
    {
     ....
      for(var s in result.series)
      {
        var serie = result.series[s];
        chart1.addSeries(s, serie);
      }
      ...
    })});
```

The `result` object contains only one property called `series`, which is then treated as an array, and on each loop, the next property of the object will be put in the variable `s`, which is first "good_looking", then "nice_looking", etc. And the variable `serie` will then contain the array property associated with the name.



Here you can see the the tooltip for one of the middle data points being active in the highlighted chart. The chart in the third column to the far right is then dragged directly under it, producing the square formation that is displayed below.



Example 3: CRUD application

One of the most common things you want to do with a web-based application is to have some sort of updateable Grid. It's commonly referred to as a CRUD (Create, Retrieve, Update, Delete) application.

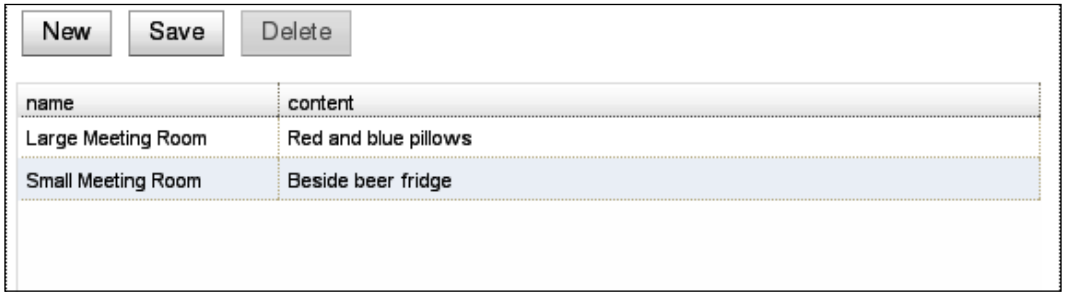
Maybe you want this up front as the whole point of your app, or maybe you just need a quick and dirty admin interface, but at some point you'll be longing for something like this.

The example uses a partially finished backend in PHP, which is used by a DataGrid with some kind of generality. You need to specify the layout of the grid and the name of the entity in the backend, and the rest will work more or less automatically.

Inspired by Kris Zyp's excellent server-side JavaScript tour-de-force, Persevere, I 've added a simple button row to create new items, save the state of the Grid, and to remove selected rows.

Persevere has excellent Dojo support, can connect to most Databases, and sports a dynamic data store interface, which lets you play around with objects and schemas dynamically in a very nice way. Persevere can be found here: `http://code.google.com/p/persevere-framework/`.

The simple crud app, after being loaded and getting initial data from the server:



The main HTML file is similar to the other examples. The difference here is the string argument `layout` to the crud Dijit, which really is a general Grid layout, formatted as a String. The layout must match the fields present in the actual data model coming from the server, naturally.

```
<html>
    <head>
        <link href="../dojo/resources/dojo.css" rel="stylesheet"
         type="text/css"/>
        <link href="../dijit/themes/dijit.css" rel="stylesheet"
         type="text/css"/>
```

```
    <link href="../dijit/themes/tundra/tundra.css" rel="stylesheet"
     type="text/css" />
    <link href="../dojox/grid/resources/Grid.css" rel="stylesheet"
     type="text/css"/>
    <link href="../dojox/grid/resources/tundraGrid.css"
     rel="stylesheet" type="text/css"/>
        <link href="/favicon.ico" rel="shortcut icon"/>
        <title>myGrid Test</title>
        <script type="text/javascript">
        djConfig=
        {
                parseOnLoad: true,
                isDebug: true,
                debugAtAllCosts: true,
                modulePaths: {mygrid: "../test/mygrid"}
         }
        </script>
        <script src="../dojo/dojo.js"  type="text/javascript">
        </script>
        <script type="text/javascript">
          dojo.require("mygrid.crud");
        </script>
     </head>
    <body class="tundra">
     <div dojoType="mygrid.crud" model="rooms" layout='
           [
           { field: "name", width: "120px", name: "name",
            editable: true  },
           { field: "location", width: "auto", name: "content",
            editable: true}
           ];' style="margin: 20px;height:300px; width:500px;"></div>
  </body>
</html>
```

As before, the HTML page looks similar to the other examples, in that only one line is required to pull in all the functionality of the widget. One property, `layout`, is used to provide information to the widget about how to present the crud table. Readers of the chapter on `dojo.data` will remember this as a typical Grid layout definition.

The component / Dijit `mygrid.crud` looks like this:

```
dojo.provide("mygrid.crud");
dojo.require("dijit.form.TextBox");
dojo.require("dijit._Templated");
dojo.require("dijit._Widget");
```

```
dojo.require("dijit.form.Button");
dojo.require("dijit.form.SimpleTextarea");
dojo.require("dojox.grid.DataGrid");
dojo.require("dijit.Editor");
dojo.require("dojox.data.JsonRestStore");
dojo.require("dojo.data.api.Read");
dojo.require("dojox.rpc.Service");
dojo.require("dojox.rpc.Rest");
dojo.declare("mygrid.crud", [  dijit._Widget, dijit._Templated ],
{
  store: "",
  layout: "",
  grid: "",
  model: "",
  baseurl: "http://localhost/dojodev/test/store.php/",
  templatePath: dojo.moduleUrl("test.mygrid","templates/crud.html"),
  widgetsInTemplate: true,
  constructor: function()
  {
    console.log("constructor for mygrid.crud called");
  },
  postCreate: function()
  {
    var self = this;
    console.log("postcreate for mygrid.crud called. model =
    "+this.model+", layout = '"+this.layout+"'");
    if (!this.store)
    {
      this.store =  new dojox.data.JsonRestStore({target: this.
      baseurl+this.model});
      console.log("store created");
    }
    dojo.connect(this.store, "onNew", function(e)
    {
        console.log("onNew event with;");
        console.dir(e);
    });
    dojo.connect(this.store, "onSet", function(e)
    {
        console.log("onNew event with;");
        console.dir(e);
    });
    var makeitems = function(items, rv)
```

```
    {
      for(var i in items)
      {
        var item = items[i];
        if (item)
        {
          console.log("Adding property "+item.field+" to default
          item");
          rv[item.field] = "";
        }
      }
    }
    var items = eval(this.layout);
    var ii = {};
    if (dojo.isArray(items[0])) // inner array def for folded rows
    {
      for(var i in items)
      {
        var inner = items[i];
        makeitems(inner, ii);
      }
    }
    else
    {
      makeitems(items, ii);
    }
    this.defaultitem = ii;
    this.render();
    console.log("postCreate done");
  },
  render: function()
  {
    if (this.grid)
    {
      this.grid.update();
    }
    else
    {
      this.createGrid();
    }
  },
  createGrid: function()
  {
    console.log("creating grid");
```

```
    var self = this;
    this.grid = new dojox.grid.DataGrid(
    {
      id: this.id+"_grid",
      store: this.store,
      clientSort: true,
      structure: eval(this.layout)
    }, document.createElement('div'));
dojo.byId(this.id+"_gridContainer").appendChild(this.grid.domNode);
    this.grid.startup();
    dojo.connect(
      this.grid,
      "onRowClick",
      function(evt)
      {
        var i = evt.rowIndex;
        console.log("row "+i+" clicked");
        self.row = self.grid.getItem(i);
        self.deletebutton.attr('disabled',false);
      }
    );
    dojo.connect(
      this.newbutton,
      "onClick",
      function(evt)
      {
        self.deletebutton.attr('disabled',true);
        console.log("creating new item locally");
        self.store.newItem(self.defaultitem);
        console.log("Saving state to server");
        self.store.save();
      }
    );
    dojo.connect(
      this.deletebutton,
      "onClick",
      function(evt)
      {
        console.log("delete button clicked");
        var item = self.row;
        console.log("item is "+item)
        console.dir(item);
        self.store.deleteItem(item);
        self.store.save();
```

```
            self.deletebutton.attr('disabled',true);
          }
        );
        dojo.connect(
          this.savebutton,
          "onClick",
          function(evt)
          {
            console.log("save button clicked deletebutton =
            "+self.deletebutton+" store = "+self.store);
            try
            {
              self.deletebutton.attr('disabled', true);
              self.store.save();
            }
            catch(e)
            {
              console.log("ERROR "+e);
              console.dir(e);
            }
          }
        );
      }
    });
```

The widget does the following:

- It creates a JsonRestStore, which reads JSON data from a server at a specific URL. The URL is created by concatenating a base URL with the name of the table that we want to show.
- In this example, we assume that the server is capable of mapping its exposed objects using REST methods, so that we can get a list of items of type `foo` by calling the URL `baseurl+/foo` and get the content of the item `1` of type `foo` by calling `baseurl+/foo/1`.
- For those interested, two handler functions have been created, which will be called each time a new item is created (`onNew`) and each time an item is changed (`onSet`) in the store.
- The Grid is then created, in its own function (which is explained in more detail later), and added to the DOM element provided for it in the template below.
- Then we add a handler for the `onRowClick` event, which manages selection of specific rows by clicking on them. This is used by the delete function, which checks which row, if any, is selected.
- Lastly, handlers for the new, save, and delete buttons get defined.

The template for the widget is very simple and looks like this:

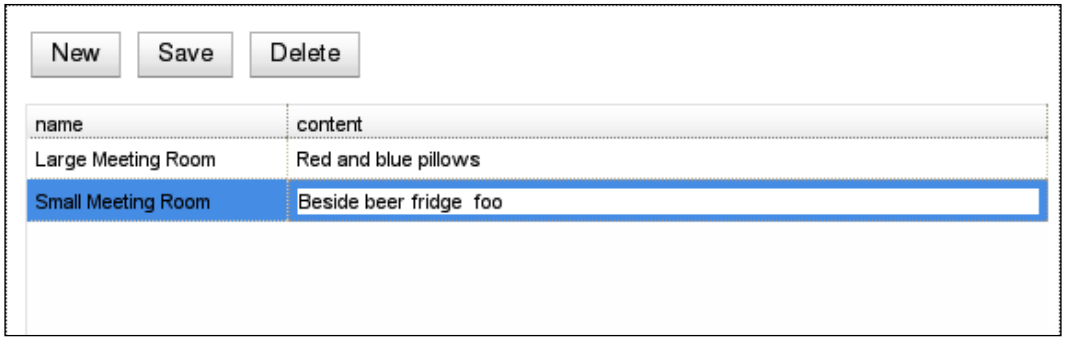
```
<div>
   <button dojoType="dijit.form.Button" dojoAttachPoint="newbutton">
    New</button>
   <button dojoType="dijit.form.Button" dojoAttachPoint="savebutton">
    Save</button>
        <button dojoType="dijit.form.Button" dojoAttachPoint=
         "deletebutton" disabled="true" >Delete</button>
   <div id="${id}_gridContainer" style="margin-top:10px;height:350px;
    width: 100%;"></div>
</div>
```

The reason for a separate `createGrid` functions is because the Grid sometimes has a problem refreshing itself when being hidden under a tab or as part of a StackContainer or similar component.

If you're using a grid (or this component) in such a way, try to detect when it is re-shown, and call the `render` function, which will update the grid in a fitting way.

In the `postCreate` function, the data store is created as part of the base URL (which is inconveniently hard-coded in this example, but which could easily be overwritten by using a parameter from the file from which the component is created.

Since editing is enabled for each column in the Grid, just double-clicking a cell makes it editable as shown below:



After clicking another row to deselect the row you can click **Save**, which will result in the store sending all changes (only one in this case) to the server; which brings us to the server.

At the moment, the server creates a simple array of rooms, if none exists, and respond to `GET` requests for either a list of rooms or a single room, and to `PUT` requests that modify a specific room.

```
<?php
   $log = "";
   $logstart = getcwd();
   function logit($it)
   {
      //----------------------------------------------------------------
----
      global $log;
      global $logstart;
      $log = $log.$it;
      $fp2 = fopen($logstart."/meh.log", "a");
      $d = date("Y-m-d H:i:s");
      fwrite($fp2, $d.">".$it."\n");
      fclose($fp2);
      //----------------------------------------------------------------
---
   }
   session_start();
   // Init "datastore"
   if (!isset($_SESSION['store']))
   {
      $store['rooms'] = Array(
                "1" => "{'id':'1', 'name':'Large Meeting Room',
                'location':'Red and blue pillows' }",
                "2" => "{'id':'2', 'name':'Small Meeting Room',
                'location':'Beside beer fridge' }"
                );
      $_SESSION['store'] = $store;
      $_SESSION['count'] = 0;
      $count = 0;
   }
   else
   {
      $store = $_SESSION['store'];
                $count = $_SESSION['count'];
   }
   header("Content-Type: " . ($_SERVER["CONTENT_TYPE"] ==
   'application/json' ? 'application/json' : 'text/plain'));
   $fn = $_SERVER['REQUEST_URI'];
   $stripval = -1;
```

```
      if( $fn[strlen($fn)-1] == '?') // If the store sends a '?' at
      //the end, it wants all entities
      {
        $stripval = -2;
      }
      $foo = substr($fn, strpos($fn, ".php")+5, $stripval);
      $bar = substr($fn, strrpos($fn, "/")+1, strlen($fn));
      $r = $store['rooms'];
      //logit("entity string is $foo");
if ($foo != "")
{
   $m = $_SERVER["REQUEST_METHOD"];
   logit("entity = '$foo', id = '$bar',  method = '$m'");
   switch ($m) {
      case "GET" :
      $rv = "";
      $s = $store[$foo];
      logit("lenght of s = ".count($s));
      if ($bar == "?")
      {
      $rv = "[";
      while (list($key, $value) = each($s))
        {
          $rv = $rv.$value.",";
        }
      if($i > 0) $rv = substr($rv, 0, -1);
      $rv = $rv."]";
         }
      else
         {
      $rv = $s[$bar];
         }
      logit("Returning '".$rv."'");
        print $rv;
        break;
      case "PUT" :
         $contents = file_get_contents('php://input');
         $x = $store[$foo];
         logit("PUT old content of '$bar' is '".$x[$bar]."'");
         $x[$bar] = $contents;
      logit("PUT new content of '$bar' is '".$x[$bar]."'");
         $store[$foo] = $x;
      print $store[$foo];
         break;
```

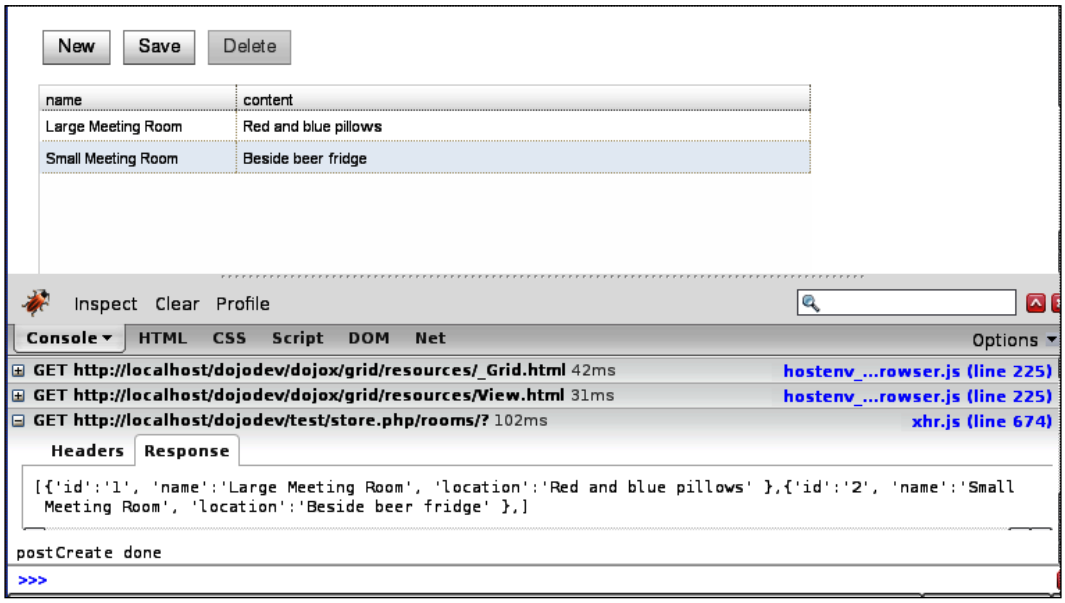
```
            case "POST" :
            //Remember to set the 'Location' header with the entity and
            //id, so that the store gets informed back
            break;
            case "DELETE" :
            $x = $store[$foo];
            unset($x[$bar]);
            $store[$foo] = $x;
            break;
        }
        $_SESSION['store'] = $store;
            $_SESSION['count'] = $count;
    }
?>
```

This simple PHP server uses session variables to hold the data items for the front-end component. It does the following:

- It checks if the `store` variable is already stored in the session. If so, it gets used and if not, it gets created for the first time and saved in the session, to be used on the next request.
- It splits the `$_SERVER['REQUEST_URI']` variable into different parts, so that it can get the name of the object type being referenced (`$foo`) and the current object `id` (`$bar`), if present.
- It checks which type of HTTP method is used, where `GET` means either get a list objects of type `$foo`, or get the contents of object with `id` `$bar` of type `$foo`, `DELETE` deletes an object, `PUT` updates an object, and `POST` should create a new instance of an object, but is left as an exercise for the reader.
- In each case, a JSON object or an array of objects gets returned to the client, by using print.

`DELETE` is partially implemented. `POST` is not implemented, and is a bit more tricky due to the fact that objects are now only stored as strings. When a new object is created, its `id` would have to be set (count) and communicated back to the JsonRestStore using the Location header.

When the example page is loaded, it shows the following:



The **Delete** button is disabled, since no row has been selected. In the bottom of the picture, we also see that the firebug console shows the content of the response that the server has sent to the `GET` method for the URL `http://localhost/dojodev/test/store.php/rooms/?`. It corresponds to the two rows being shown in the DataGrid at the top.

Summary

- Dojo can be loaded cross-domain, removing the need to host your own versions and simplifying the creation of mockup pages of applications.
- Dojo has a Java-based build system, which lets you combine multiple files into one, compress source, and manage JavaScript source files in several different places.
- Dojo has a unit-test system called **DOH (Dojo Objective Harness)**, which is used for other/external scripts as well.

Index

A

Ajax Hello World
about 86-88
content-types 93
content parameters 92
data, posting 89- 92
errors 89
Ajax integration
example 119
All Form Dijit
dojo.date 110
Dojo locales 109
event handlers 105
I18N control Dijits 104
properties 104
t.is()function, using 111
validation-aware Dijits 105
anatomy, Dojo page
djConfig 40
dojo.css 40
dojo.require statements 40
dojoType used 41
example 42, 43
AOL's cross-domain build
using 207
array
about 28
pop() function 29
push() function 28
stack, using as 28

B

BorderLayout, compound example
AccordionContainer 157
BorderContainer 157
ScrollPane 158
source code example 153-157
toolbar 157
buttons
DropDownButton 117
DropDownButton, example 117, 118
DropDownButton, viewing 117

C

CDN 207
CheckBox
using 112
closure
facts, utilizing 35
using 33, 34
using, rules 33
compound example
BorderLayout container, using 153
informations, displaying 153
using layout 153
container, functions
AccordionContainer 131
BorderContainer 132-135
StackContainer, codes 128-130
StackContainer,defining 127, 128
StackContainer, topics 126
TabContainer 130
content-types, Ajax Hello World
javascript 94
json 94
text 94
Content Delivery Network. *See* **CDN**
Create,Retrieve,Update,Delete application. *See* **CRUD application**
CRUD application 233

D

debug
about 46
flag 46
Dijits
about 57
creating 59
creating, ways 57
CSS files, including 58
dijit.byId(), using 60
Dojo 1.1.0 list 61-64
identifying 60
structure 64
Django Templating Language. *See* **DTL**
DnD 187, 188
DOH
about 212
runTests.html 213
unit testing, assertEqual function 215
unit testing, assertFalse function 215
unit testing, assertTrue function 215
unit testing, using 214, 215
Dojo
about 7, 14
Ajax, using 86, 87
Ajax Hello World 86
Ajax integration 119
API listings 22
basic functions 43, 44
buttons 117
CheckBox 111
class, declaring 35
comparing, with Ajax frameworks 21
comparing, with EXT 21
comparing, with jQuery 21
comparing, with Prototype 21
comparing, with YUI 21
DOH, using 212
Dojo Ajax features 83-86
dbugging, features 46
dynamic charting page example 8, 9
event, handling 49
features 17-20
functionality 7
getting started 16
intermediate functions 44, 45
mailing list 22
Object-Oriented class system 35, 36
overview 5
properties 47
RadioButton 112
reference sites 22
remote procedure calls 96
server-side compatibility 21
stack container wizard 13
subdirectories 15
ToggleButton 112
uses 6
uses, examples 6
Dojo, build system
about 209-212
dependencies 211
features 210
function 210
layers 211
standard.js file 210, 211
dojo.data
about 167
feature 167
FilteringSelect widget example 169, 170
stores 167
dojo.data API functions
common function 177
dojo.data.Identity 175
dojo.data.Notification 175, 176
dojo.data.Read 171, 173
dojo.data.Write 174
Identity function 171
Notification function 171
Read function 171
Write function 171
dojo.io.script.get 95
Dojo Ajax features
about 83-86
dojo.xhr 84
handlesAS parameter 85
load parameter 85
url parameter 85
Dojo Ojective Harness. *See* **DOH**
Dojo page
anatomy 38, 39

dojox.data
FilteringSelect widget example 169, 170
stores 168
Drag and Drop. *See* **DnD**
DragPane
about 135
invert property, example 135
uses 135
DTL
about 70, 71
features 71
recommended widget structure 72-76
using 70
DTL, features
argument, passing 72
conditionals 72
loops 72
variables, substituting 71

E

event, DOJO
charcodes, references 50, 51
feature 49
properties 50
ExpandoPane
about 136
declaring 137, 138
example 136

F

FilteringSelects widget
example markup 115, 116
Firebug
using 46
FloatingPane
about 138
boolean property 139
closable property 139
duration property 139
example 139
resizable property 139
forest
about 181, 182
dijit.tree.ForestStoreModel used 181, 182
Form
about 99
features 100
Form, properties
action 100
method 100
Form Dijit
functions, calling 101
using 101-103
functions
using, as arguments 29-31

G

getValue function 171
Google's CDN. *See* **CDN**
using 208
GridContainer
about 140
creating, using markup 142
example 140
properties 143-145
viewing 141
working 140, 142
Grid Dijit
cell, editing 205
code snippet, activities 204
custom, formatter function 201
custom, get function 201
custom, rendering 200
custom sortable grid 203
editor types, defining 205
grid, sorting 202
grid structure 203
grid views 197
overview 194
simple grid 195-197
grid views, Grid Dijit
about 198
defining 197
onStyleRow function 199

I

I18N
about 78
Accessibility Rich Internet Applications(ARIA) 81
djConfig.locale 79
djConfig.locale, configuring 79

programmatic locale formatting 80
widgets, degrading 81

J

JavaScript
closure 33
history 23, 24
object literal notation 25-27
object literal notation, precautions 25
objects 24
objects, creating 24
JavaScript's protoype
using 31
JavaScript Object Notation. *See* **JSON**
JavaScript protoyype
properties, defining 32, 33
JSON
about 54
functions, describing 55
JsonRestStore
about 190
working 190-194

L

logging
levels 46
in Firebug 46

M

models 171

N

Number TextBox, validation-aware Dijits 105, 106

O

Object-Oriented class system, Dojo
class, creating 36, 37
class references array, using 38
dojo.declare function 36
subclassing 37, 38

P

properties, Dojo 47
pop() function, array 29
push() function, array 28

R

RadioButton
using 112
RadioGroup
about 145-148
standalone animation, creating 147
remote procedure calls
about 96
Simple Method Description(SMD) file 97
ResizeHandle
about 148, 149
defining 148, 149
example 148
RotatorContainer
about 150
attatchPoints 151
creating 150, 151

S

ScrollPane
about 152, 153
creating 152
Simple Method Description(SMD) file, remote procedure calls 97
simple tree example
about 177-181
rootID property 178
TreeStoreModel 178
Slider widget
markup 114
markup, viewing 114
using 113
spinner
about 113
example 113
structure, Dijits
Attach Point 68
DTL 70, 71
events 69

extension point 69
lifecycle 64, 65
templates 66, 67
subdirectories, Dojo
dijit 15
dojo 15
dojox 15
util 15

T

themes
a11y 77
about 77
Noir 77
setting, for Dojo widgets 78
Soria 77
Tundra 77
TimeText Box
example 116, 117
using 116
ToggleButton
examples 112
using 112
trees 181
trees, events
dojo.connect, using 183
example 183, 184
function hooks 182
trees, styling
DnD, parameters 187
DnD, unit test 188, 190
DnD 187
getIconClass hook 185
getLabelClass hook 185
getLabel hook 184
hook functions 184
hook functions, overriding 185, 186

U

unit testing, DOH
CRUD app 233
CRUD app, widget activities 238
example 215
example, CRUD app 233-243
example, multi charting application 224-232
example, site navigation scaffolding 215-223
examples 233
widgets functionality outline 221

V

validation-aware Dijits, All Form Dijit
CurrencyTextBox 107, 108
DataTextBox 108
NumberTextBox 105, 106
TextBox 105

W

widget, creating
steps 159-165
structures, using 158
template differences 160



Thank you for buying
Learning Dojo

Packt Open Source Project Royalties

When we sell a book written on an Open Source project, we pay a royalty directly to that project. Therefore by purchasing Learning Dojo, Packt will have given some of the money received to the Dojo Foundation project.

In the long term, we see ourselves and you—customers and readers of our books—as part of the Open Source ecosystem, providing sustainable revenue for the projects we publish on. Our aim at Packt is to establish publishing royalties as an essential part of the service and support a business model that sustains Open Source.

If you're working with an Open Source project that you would like us to publish on, and subsequently pay royalties to, please get in touch with us.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

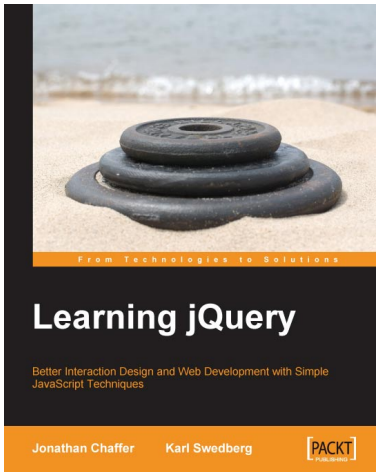
About Packt Publishing

Packt, pronounced 'packed', published its first book "Mastering phpMyAdmin for Effective MySQL Management" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.PacktPub.com`.

PACKT PUBLISHING



Learning jQuery

ISBN: 978-1-847192-50-9 Paperback: 380 pages

jQuery: Better Interaction Design and Web Development with Simple JavaScript Techniques

1. Create better, cross-platform JavaScript code
2. Learn detailed solutions to specific client-side problems
3. For web designers who want to create interactive elements for their designs
4. For developers who want to create the best user interface for their web applications

---



Google Web Toolkit GWT Java AJAX Programming

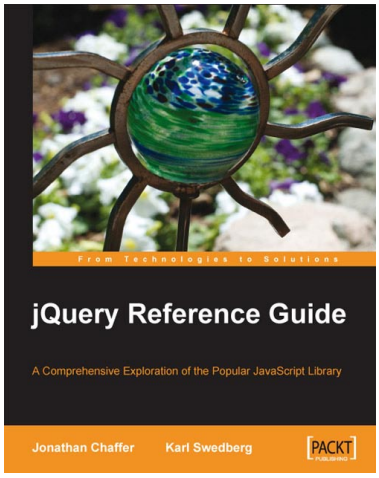
ISBN: 978-1-847191-00-7 Paperback: 240 pages

A practical guide to Google Web Toolkit for creating AJAX applications with Java, fast

1. Create rich Ajax applications in the style of Gmail, Google Maps, and Google Calendar
2. Interface with Web APIs create GWT applications that consume web services
3. Completely practical with hands-on examples and complete tutorials right from the first chapter

Please check **www.PacktPub.com** for information on our titles

PACKT PUBLISHING



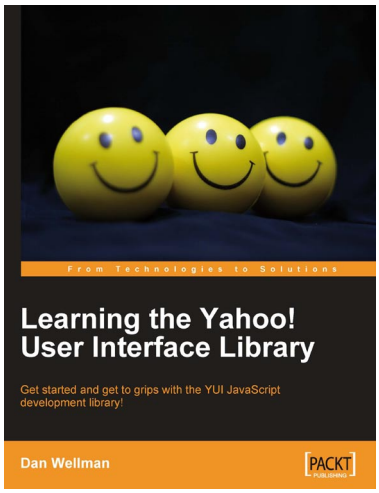
jQuery Reference Guide

ISBN: 978-1-847193-81-0 Paperback: 225 pages

A Comprehensive Exploration of the Popular JavaScript Library

1. Organized menu to every method, function, and selector in the jQuery library
2. Quickly look up features of the jQuery library
3. Understand the anatomy of a jQuery script
4. Extend jQuery's built-in capabilities with plug-ins, and even write your own

---



Learning the Yahoo! User Interface library

ISBN: 978-1-847192-32-5 Paperback: 380 pages

Develop your next generation web applications with the YUI JavaScript development library

1. Improve your coding and productivity with the YUI Library
2. Gain a thorough understanding of the YUI tools
3. Learn from detailed examples for common tasks

Please check **www.PacktPub.com** for information on our titles