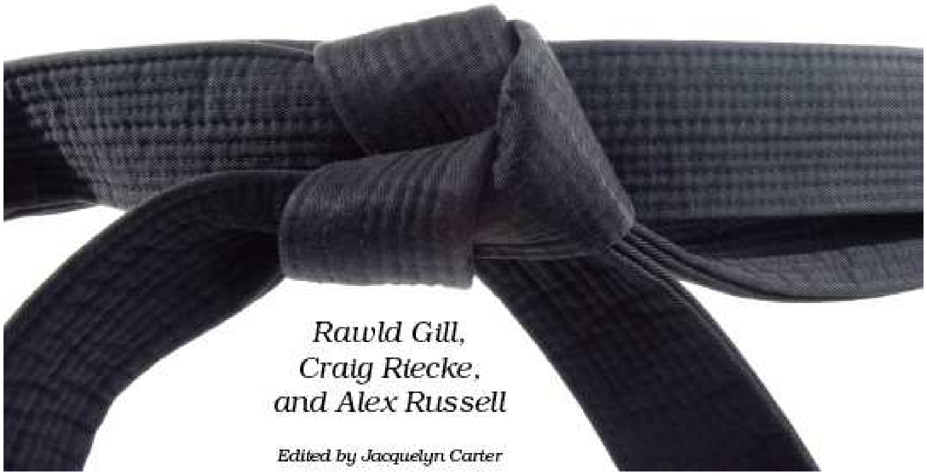
Mastering Dojo

JavaScript and Ajax Tools for Great Web Experiences

The Pragmatic Programmers

Mastering Dojo

JavaScript and Ajax Tools for Great Web Experiences



Rawld Gill,
Craig Riecke,
and Alex Russell

Edited by Jacquelyn Carter

ISBN-10: 1-934356-11-5

ISBN-13: 978-1-934356-11-1

Printed on acid-free paper with 50% recycled, 15% post-consumer content.

P2.0 printing, January 2010

Version: 2010-2-3

```
var listNode = document.getElementById("listOfUrls");
dojo.forEach( urls, function(oneResult) {
    var listItem = document.createElement("li");
    listItem.innerHTML =
       dojo.string.substitute("<a href='${url}'>${title}</a>", oneResult);
    listNode.appendChild(listItem);
});
```

We'll see plenty more examples of function literals, but here's what we need to know for Dojo XHR: XmlHttpRequest calls are asynchronous, meaning they return control to the JavaScript program immediately while they work in the background. We need to tell XHR, "This is what you do when the result comes back." That's perfect for a function literal, because most often the function is used only for that XHR call. Defining a named function to call it only once feels like too much overhead. A function we pass to an asynchronous request is called a *handler*, or equivalently a *callback*. We want the process to "call back" our function when it's ready.

The functions-as-data concept of JavaScript turns out to be extremely useful. Dojo uses it in XHR and also in surprising places such as animations and declaring subclasses. We'll discuss those later, but for now let's dig into the first project.

3.3 A Wish List with dojo.data and dojox.grid.Grid

The Justa Cigar Corporation is overhauling its web site, and we've just won the contract to write it. Customers on the Justa site are gung ho about cigars, and Justa wants to help them connect to each other, share information, and (Justa hopes) purchase cigars.[2]

Each Justa customer has a wish list of cigars with brand names, sizes, country of origin, and other information. The execs want to list this in a scrollable table and give the customer the ability to add, delete, and edit cigars in place without leaving the page. In Figure 3.2, on the next page, you can see the "cocktail napkin" view of what Justa wants. Already this is impossible with the old web technology because of the don't-leave-the-page requirement. We're going to need XHR to send add, delete, and edit requests to the server.

2. Just for the record, we don't condone such carcinogenic activities. The point is you can use the same techniques for finding ice cream, cross-country skis, or flat-screen televisions. Whatever floats your boat. Dojo will not judge you.

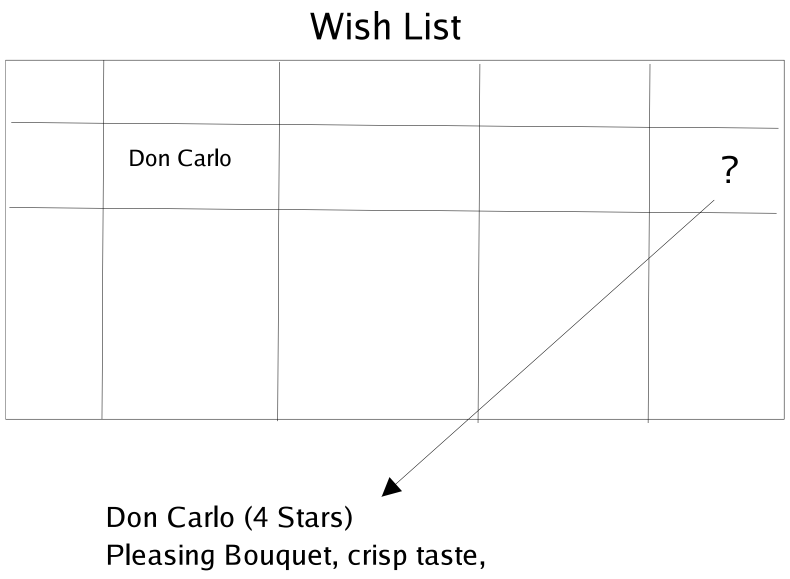


Figure 3.2: Wish-list user interface design (coffee stain omitted)

dojox.grid.Grid is a good fit for the user interface part. Grid acts like a mini-spreadsheet where you can view, sort, filter, and edit tabular information. dojo.data provides plumbing between Grid and server-based data. Together they'll form the backbone of the wish list.

Because Justa controls the web site and database, its IT department can write web services in any format we want. These web services will read from the wish-list database, translate the data to the right format, and send. Writing these server-side scripts is beyond the scope of this book, but Dojo can generally work with any server-side language and any database because they communicate over HTTP. If this notion is foreign to you, the sidebar on the following page gives you some help on where to start.

Fortunately, we can stub out the web services while writing the client. After all, XHR simply sends a request to a URL and gets data back in some standard format. A plain old text file will suffice—it has a URL, and we can write it in some format. That makes the job small and easier to manage. But we still need to pick a data format. dojo.data has drivers for commonly used data formats such as XML, comma-

Server-Side Options

Dojo enables a significant architectural shift. When scripting fill-and-submit pages in a middle-tier language such as PHP, ASP, or JSP, both the navigation and HTML generation rest mostly in that language. With Dojo, more code executes on the client and less on the middle tier. But you still need some middle-tier code for the following:

- Connecting to a database and passing the results in a format Dojo understands
- Proxying calls to external web services

PHP is a particularly adept language for these calls because proxying is a one-line call and JSON support is built in (at least to 5.2 and newer). But any language that can talk to HTTP servers and use databases will work.

Alternatively, you can use an enterprise service bus (ESB). This software is made for proxying and translation, and many require declarative configuration in lieu of programming. Many ESB products are large and expensive, but Apache Synapse is an open source, lightweight ESB that's a good match for Dojo.

separated variables (CSV), JSON, and HTML tables. Seeing this list, you might think XML is the way to go. But hold up a minute!

JSON, the Language

JavaScript Object Notation is a better choice for sending the wish-list data in our example. If you haven't seen or used it before, you might wonder why we'd pick JSON over XML. After all, practically every programming language and every browser on Earth speaks XML. It's self-descriptive, standardized, and mature.

But in browsers, XML suffers from two problems. First, browsers don't implement XML standards uniformly. Second, and this is the deal-breaker, browser-based XML implementations are slow. That's a problem because the more interactive you want your interface, the chattier you have to be with the server, and the faster your data interchange format must be.

Enter JSON. In a nutshell, JSON data looks like the right side of a JavaScript assignment. For comparison, here is a snippet of XML data from our Justa wish lists:

```
<wishListItem>
   <wishId>4455</wishId>
   <description>Don Pepin Garcia Delicias</description>
   <size>7-50</size>
   ...
</wishListItem>
```

This is equivalent to the following JSON:

```
"wishListItem": {
   "wishId": 4455,
   "description": "Don Pepin Garcia Delicias",
   "size": "7-50",
...
}
```

This looks suspiciously like a hash literal. And it is! However, JSON has more restrictions placed on it:

- All strings, including the names on the left side of the colon (:), must be quoted. (Hash literals are not as strict.)
- Nested hashes and arrays are allowed on the right side of the colon. But the only primitive data allowed are single- or double-quoted strings, numbers, the boolean constants **true** and **false**, and the constant null. No expressions or variable names are allowed.

It's really that simple. Dojo feeds the data to a JavaScript eval, gets back a hash, and gives it to you. But it also enforces the JSON quoting rules behind the scenes, and that's an important security feature. Otherwise, someone could type in a wish-list item named while(1); and lock up someone's browser.

If you're given the choice of web service output to consume by Dojo, JSON is usually preferable to XML. It's expressive, flexible, and easy to manipulate in JavaScript. Adapters for popular server-side languages are plentiful, as you can see at http://www.json.org. And it's fast, fast, fast! Some studies have clocked it at 100 times faster than XML in a browser. This makes sense because JSON is "closer to the metal" of JavaScript and requires less translation. When you need cigar data, those extra clock cycles count!

A Stub Data Source

dojo.data has its own terminology, which we will cover completely in Chapter 10, *dojo.data*, on page 262. But here's enough to build our test wish-list data. A *data source* is the URL from which the data comes. In our test case, the URL will be very simple: services/cigar_wish_list.json. When we fill out the stubs, we'll probably send parameters along with it, as in services/cigar_wish_list.php?userid=99555. A *data store* is the corresponding dojo.data object that holds the data. Finally, an *item* is one data object. An item is like a database record but can have a more complex structure.

So, here's a snippet from our data source, in services/cigar_wish_list.json:

Download xhr_techniques/services/cigar_wish_list_abbrv.json

```
{
   "identifier": "wishId",
   "label": "description",
   "items":
   [
      {
        "wishId": 4455, "description": "Don Pepin Garcia Delicias",
        "size": "7-50", "origin": "Nicaragua", "wrapper": "Corojo",
        "shape": "Straight"
      },
      {
        "wishId": 4456, "description": "601 Habano Robusto",
        "size": "5-50", "origin": "Nicaragua", "wrapper": "Natural",
        "shape": "Straight"
      },
      {
        "wishId": 4457, "description": "Black Pearl Rojo Robusto",
        "size": "4 3/4-52", "origin": "Nicaragua", "wrapper": "Natural",
        "shape": "Straight"
      },
      /* ... */
   ]
}
```

The structure may look complex on first glance. Judicious use of whitespace makes things a bit easier—here we use a style similar to a JavaScript program, lining up brackets and indenting common levels. Working from the inside out, a wish-list item...

Download xhr_techniques/services/cigar_wish_list_abbrv.json

```
{
  "wishId": 4455, "description": "Don Pepin Garcia Delicias",
  "size": "7-50", "origin": "Nicaragua", "wrapper": "Corojo",
  "shape": "Straight"
},
```

is a simple hash literal following the JSON rules. The brackets surrounding these hashes create an array of wish-list objects:

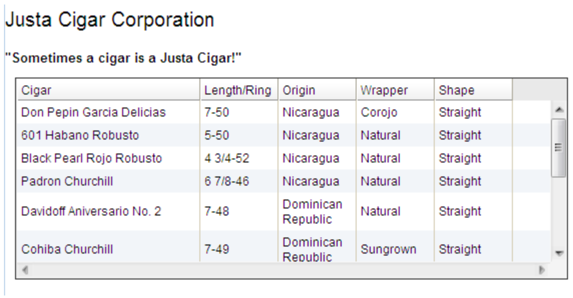
Download xhr_techniques/services/cigar_wish_list_abbrv.json

```
[
   {
     "wishId": 4455, "description": "Don Pepin Garcia Delicias",
     "size": "7-50", "origin": "Nicaragua", "wrapper": "Corojo",
     "shape": "Straight"
   },
   {
     "wishId": 4456, "description": "601 Habano Robusto",
     "size": "5-50", "origin": "Nicaragua", "wrapper": "Natural",
     "shape": "Straight"
   },
   {
     "wishId": 4457, "description": "Black Pearl Rojo Robusto",
     "size": "4 3/4-52", "origin": "Nicaragua", "wrapper": "Natural",
     "shape": "Straight"
   },
   /* ... */
]
```

Finally, this array becomes the items property of the data source, as we saw earlier:

Download xhr_techniques/services/cigar_wish_list_abbrv.json

```
{
   "identifier": "wishId",
   "label": "description",
   "items":
   [
      {
        "wishId": 4455, "description": "Don Pepin Garcia Delicias",
        "size": "7-50", "origin": "Nicaragua", "wrapper": "Corojo",
        "shape": "Straight"
      },
      {
        "wishId": 4456, "description": "601 Habano Robusto",
        "size": "5-50", "origin": "Nicaragua", "wrapper": "Natural",
        "shape": "Straight"
      },
      {
        "wishId": 4457, "description": "Black Pearl Rojo Robusto",
        "size": "4 3/4-52", "origin": "Nicaragua", "wrapper": "Natural",
        "shape": "Straight"
      },
      /* ... */
   ]
}
```



Justa Cigar Corporation

"Sometimes a cigar is a Justa Cigar!"

Cigar	Length/Ring	Origin	Wrapper	Shape
Don Pepin Garcia Delicias	7-50	Nicaragua	Corojo	Straight
601 Habano Robusto	5-50	Nicaragua	Natural	Straight
Black Pearl Rojo Robusto	4 3/4-52	Nicaragua	Natural	Straight
Padron Churchill	6 7/8-46	Nicaragua	Natural	Straight
Davidoff Aniversario No. 2	7-48	Dominican Republic	Natural	Straight
Cohiba Churchill	7-49	Dominican Republic	Sungrown	Straight

Figure 3.3: The wish list: scrollable, sortable, and full of tasty cigars

We are then going to feed this into the dojo.data driver dojo.data.ItemFileReadStore. This driver expects JSON data in a specific format with the following properties: identifier is the field containing an item's ID; label is the field with the human-readable identifier; and items is the data itself, which is an array of hashes. Not all dojo.data drivers are this restrictive: CSV and XML data sources do not require an identifier field, for example.

The IT people at Justa will write a server-side component that reads database records and writes the data into this format. But this fixed data source will do for now.

The Data-Enabled Widget, dojox.grid.Grid

Grid widgets are very familiar to GUI designers. A grid is a spreadsheet-like "supertable" that allows editing, sophisticated display, and a well-structured event system. Grids are unfamiliar to most web programmers, though, because they're difficult to construct from scratch.

The Dojox grid component is the state-of-the-art in web-enabled data grids, and it gives client-server grids a run for their money in features, performance, and stability. You can pipe dojo.data data stores into it with just a few lines of code.

The goal is to get to Figure 3.3. It looks pretty sophisticated, but every journey begins with a small step, so let's begin.

Every grid needs data. A grid's *model* is the set of data that is feeding the grid, named after the *M* in MVC architecture.

We've already built our data source, and making this a model requires just one <*div*> tag:

Download xhr_techniques/wish_list_grid.html

```
<div dojoType="dojo.data.ItemFileReadStore"
    jsId="wishStore" url="services/cigar_wish_list.json">
</div>
```

This looks a lot like a Dijit component, but it's not. It doesn't display anything—which is usually a tip-off that it's not from Dijit. And the dojoType= value does not begin with dijit. (In Section 12.1, *What Exactly Is a Widget?*, on page 321, we'll learn the full story.) Instead, it acts more like an assignment statement. The jsId= attribute declares a JavaScript variable to hold the object. You can use these variables in your own JavaScript code, or, as we do here, feed the contents of one object into another. So, this tag sets up a dojo.data.ItemFileReadStore, a data store using JSON in the special format we used for cigar_wish_list.json.

With the model taken care of, we can define the grid itself:

Download xhr_techniques/wish_list_grid.html

```
<table id="grid" dojoType="dojox.grid.Grid" store="wishStore"
     query="{ wishId: '*' }" clientSort="true">
    <thead>
       <tr>
          <th field="description" width="15em">Cigar</th>
          <th field="size">Length/Ring</th>
          <th field="origin">Origin</th>
          <th field="wrapper">Wrapper</th>
          <th field="shape">Shape</th>
       </tr>
    </thead>
</table>
```

Hmmm, that sure looks like an HTML <*table*>. . . although there's a few extra attributes. But those extra attributes wield tremendous power. First, dojox.grid.Grid takes in the data store wishStore, defined earlier. The grid can apply sorting and filtering to the data store, designated by the attributes clientSort= and query=. The former is straightforward. The latter involves another hash literal defining filter criteria. In this case, { wishId: '*' } means "Match every item that has a wishId property." In our case, that's all the records in the store.

Inside the <*table*> tag, it looks like a table with exactly one row. For our simple two-dimensional table type of grid, all we need are a few <*th*>

tags, each of which define column characteristics. The field property points to a field in our data source. (You'll learn more about that later.) And the body of the tag is used as the column header.

Let's step back, review our work, and fold in some last-minute touches. Here's the full script:

Download xhr_techniques/wish_list_grid.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
         "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Justa Cigar Wish List</title>
<style type="text/css">
    @import "/dojoroot/dijit/themes/tundra/tundra.css";
    @import "/dojoroot/dojo/resources/dojo.css";
    @import "/dojoroot/dojox/grid/_grid/tundraGrid.css";
</style>
<script type="text/javascript" src="/dojoroot/dojo/dojo.js"
    djConfig="parseOnLoad: true"></script>
<script>
    dojo.require("dojo.parser");
    dojo.require("dojo.data.ItemFileReadStore");
    dojo.require("dojox.grid.Grid");

</script>
<style>
#grid {
    border: 1px solid #333;
    width: 550px;
    margin: 10px;
    height: 200px;
    font-size: 0.9em;
    font-family: Geneva, Arial, Helvetica, sans-serif;

}
</style>

</head>
<body class="tundra">

<h1>Justa Cigar Corporation</h1>
<h3>"Sometimes a cigar is a Justa Cigar!"</h3>

    <div dojoType="dojo.data.ItemFileReadStore"
        jsId="wishStore" url="services/cigar_wish_list.json">
    </div>

    <table id="grid" dojoType="dojox.grid.Grid" store="wishStore"
         query="{ wishId: '*' }" clientSort="true">
        <thead>
           <tr>
              <th field="description" width="15em">Cigar</th>
              <th field="size">Length/Ring</th>
              <th field="origin">Origin</th>
              <th field="wrapper">Wrapper</th>
              <th field="shape">Shape</th>
           </tr>
        </thead>
    </table>

</body>
</html>
```

There are two things to note here. First, we must add an extra style sheet to make the Grid match the Tundra theme, which you see at line 10. Second, the two dojo.require calls starting at line 16 load the Grid and ItemFileReadStore components.

Run this script, and the grid pops up, as shown in Figure 3.3, on page 54. And dig the functionality! The grid has the following characteristics:

- *Alternately striped*: Odd/even colors are automatically applied for easy reading.
- *Scrollable*: Scroll bars automatically appear if needed for horizontal or vertical scrolling.
- *Column sizable*: Point between columns on the top, and drag left or right.
- *Row-selectable*: Just click anywhere on a row to select it. The row changes color.
- *Sortable*: Just click a column header to sort by that field. Click again to switch the sort direction.

If you have the Firebug debugger installed in your browser, you can watch the XHR packets go over the network, as shown in Figure 3.4, on the next page.

dojo.data and dojox.grid provide a quick way to get XHR up and running against your domain's web services, but what if you want to use services outside your network? You can write your own proxy in a server-side language that calls the outside service on your behalf. But for some specially written web services, there's an even easier way.