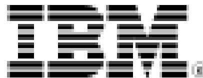


developerWorks.

Developing widgets with Dojo 1.x

Skill Level: Introductory

Marco Lerro (Marco Lerro/Italy/IBM@IBMIT)
Staff Engineer
IBM

Roberto Longobardi (Roberto Longobardi/Italy/IBM@IBMIT)
Software Developer
IBM

Gianluca Perreca (Gianluca Perreca/Italy/IBM@IBMIT)
Staff Engineer
IBM

Alessandro Scotti (Alessandro Scotti/Italy/IBM@IBMIT)
Advisory Software Engineer
IBM

28 Apr 2009

Learn the basics of developing HTML widgets using the Dojo JavaScript toolkit. This article gives you an introduction, and provides several examples to help you in the process—starting with sample widgets and moving up to more complex widgets, while highlighting and solving the common issues you could encounter in the development phase.

Introduction

The goal of this article is to give you the basics for developing HTML widgets using the Dojo JavaScript toolkit, starting from version 1.0. The article also describes several examples, beginning with simple widgets and moving up to more complex ones, while solving common issues you might encounter in widget development.

What is the Dojo toolkit?

© Copyright IBM Corporation 2009

Dojo is an open source, JavaScript-based toolkit for developing dynamic HTML Web applications. It allows you to quickly build widgets that can be more complex than standard HTML widgets. Using the components that Dojo provides makes your Web user interfaces more usable, responsive, and functional. Low-level API and compatibility layers provided by Dojo help you write applications compatible across browsers.

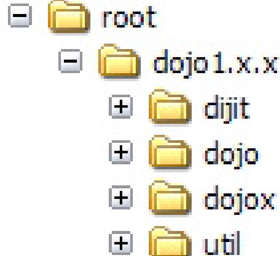
Before you start

Before you start, you'll first need to set up the development environment. To do that:

1. Download the latest released version of the Dojo toolkit (*dojo-release-1.x.x-src.zip* or *dojo-release-1.x.x-src.tar.gz*) from the Dojo project site. (See Resources for a link.)
2. Unpack the contents of the archive into a folder, paying attention to the location of where the dojo.js file is unzipped. The package system manages the loading of all modules after dojo.js has been loaded into the page.

When you're done, the structure of the folders will be like those shown in Figure 1.

Figure 1. Folder structure of expanded dojo file



Dijit is a widget system layered on top of dojo. Through its own theme, *tundra*, it provides a design and color scheme common to all its widgets. *Dojox* is a development package of extensions to the Dojo toolkit. It is intended for someone who is looking for functions that are not in the common collections.

Dojo widgets

When browsing through Web sites, you'll see hundreds of widgets come across your screen. Each button in your Web browser is a widget. Each text entry box is a widget. Standard HTML provides a limited set of widgets: an input box, a button, and a hyperlink.

© Copyright IBM Corporation 2009

Dojo widgets take an item like a text input box and add functions for a more user-friendly object, such as a graphical calendar to choose a date. It does this without breaking the original item on which the new function is built.

A Dojo widget encapsulates visual Web components for easy reuse. It is defined by three files:

- A JavaScript file that contains the widget logic
- An optional HTML file that provides a HTML-like template for the widget
- A CSS file, usually common for all the widgets (the theme), that contains visual styles to be applied to the widget's HTML templates

Importing the Dojo toolkit

Listing 2 shows a basic HTML skeleton that can be used to import a widget in a normal Web page.

Listing 1. HTML code to import a widget into a Web page

```
<html>
   <head>
      <title>Dojo Toolkit Test Page</title>

      <style type="text/css">
         /* CSS style code */
      </style>

 <script type="text/javascript" src="js/dojo1.2/dojo/dojo.js"
 djConfig="parseOnLoad:true, isDebug:true"></script>

      <script type="text/javascript">
         /* Javascript code */
      </script>
   </head>
   <body>
      /* Widgets definition code */
   </body>
</html>
```

The first script tag initializes the Dojo toolkit by simply loading the dojo.js bootstrap file. The `parseOnLoad` and `isDebug` properties of the *djConfig* object are the two most common configuration options that Dojo examines at runtime. `parseOnLoad` toggles the mark-up parsing at load-time, while `isDebug` enables or disables debugging messages. The `djConfig` object can be also set up as a global variable before the dojo.js file is loaded:

Listing 2. Code to set up global variables with djConfig

```
<script type="text/javascript">
   var djConfig = {
      isDebug:true, parseOnLoad:true
```


© Copyright IBM Corporation 2009

```
   };
</script>
<script type="text/javascript" src="js/dojo1.2/dojo/dojo.js"></script>
```

The Dojo package system

Dojo has a package system for structuring the application classes in files and loading them through the `dojo.require` function. This function allows loading parts of the Dojo toolkit not already provided in the base `dojo.js`.

To create a widget, you must import the widget declaration by adding the lines from Listing 3.

Listing 3. Code to import a widget declaration

```
<script type="text/javascript">
   dojo.require("widgets.Button");
</script>
```

Now, you can insert the following code into the body section:

```
<body>
   <div dojoType="widgets.Button">My Button</div>
</body>
```

The `dojoType` attribute causes the Dojo toolkit to manage the tag in a special way. At page-loading-time, the Dojo parser looks for the widget declaration specified in the `dojoType` attribute, instantiates the widget, and replaces the tag with the obtained widget DOM node.

Declaring a widget

Now, let's look at a `TextBox` widget example, defining a JavaScript file, a template file, and a CSS style file.

First, you must create the JavaScript file ,TextBox.js, containing the definition and the logic of the widget. (See Listing 4.)

Listing 4. Contests of the JavaScript file TextBox.js

```
dojo.provide("widgets.TextBox");
dojo.require("dijit._Widget");
dojo.require("dijit._Templated");

dojo.declare(
    "widgets.TextBox",
    [dijit._Widget, dijit._Templated],
    {
```


© Copyright IBM Corporation 2009

```
        /** the template path */
        templatePath: dojo.moduleUrl("widgets", "templates/TextBox.html"),
        /** the input DOM node */
        inputNode: null,
        /** the label */
        label: "",

        /** onkeyup handler */
        onKeyUp: function() {
            // give a chance to the browser to update the DOM
            setTimeout(dojo.hitch(this, this.validate), 0);
        },

        /** validate function */
        validate: function() {
            if ( this.inputNode.value === "Ok" ) {
                // the text is correct
                dojo.addClass(this.inputNode, "inputOk");
                dojo.removeClass(this.inputNode, "inputError");
            } else {
                 // the text is incorrect
                 dojo.removeClass(this.inputNode, "inputOk");
                 dojo.addClass(this.inputNode, "inputError");
              }
           }
      }
);
```

`dojo.provide()` defines the name of the new widget and registers the class declaration. Note that:

- `dijit._Widget` and `dijit._Templated` are superclasses for the TextBox widget
- `templatePath`, `inputNode`, and `label` are attributes of the widget
- `onKeyUp()` and `validate()` are the functions that define the logic of the widget

Now, you can define the template file TextBox.html, as shown in Listing 5.

Listing 5. Contents of TextBox.html

```
<span class="textBox">
   ${label}:
   <input
       type="text"
       class="inputOk"
       dojoAttachPoint="inputNode"
       dojoAttachEvent="onkeyup: onKeyUp">
   </input>
</span>
```

The `${label}` will be substituted with the label property of the widget instance.

The `dojoAttachPoint` declaration will cause the inputNode widget's property to

© Copyright IBM Corporation 2009

be set to the DOM node corresponding to the input tag.

The `dojoAttachEvent` declaration will cause the `onkeyup` events (coming from the input node) to trigger the `onKeyUp` widget's method calls.

The class `textBox` and `inputOk` refers to the CSS class names defined in the TextBox.css file. See Listing 6.

Listing 6. CSS class names in TextBox.css

```
.ibm .textBox {
   margin: 5px;
   padding: 5px;
   background-color: #eee;
}

.ibm .inputOk {
   border: 1px solid green;
}

.ibm .inputError {
   border: 1px solid red;
}
```

Because the class names have to be unique across the project, usually they have a CSS selector (*ibm* in the sample).

Finally, the widget can be created on the HTML page, as shown in Listing 7.

Listing 7. Code to create the widget on the HTML page

```
<html>
   <head>
      <!-- page title -->
      <title>TextBox Widget</title>
      <!-- include the DOJO -->
      <script type="text/javascript" src="../dojo-1.0.0/dojo/dojo.js"
                      djConfig="isDebug: true, parseOnLoad: true">
      </script>

      <!-- import DOJO base CSS, DIJIT theme, and widget CSS -->
      <style type="text/css">
         @import "../dojo-1.0.0/dojo/resources/dojo.css";
         @import "../dojo-1.0.0/dijit/themes/tundra/tundra.css";
         @import "templates/TextBox.css";
      </style>

      <!-- import DOJO stuff -->
      <script type="text/javascript">
         dojo.require("dojo.parser");
         <!-- register our module path -->
         dojo.registerModulePath("widgets", "../../widget");
         <!-- import our stuff -->
         dojo.require("widgets.TextBox");
      </script>
   </head>
```


© Copyright IBM Corporation 2009

```
    <body class="tundra ibm" style="padding:5px">
    <br/>
       <!-- declare the DOJO widget -->
       <div
          dojoType="widgets.TextBox"
          label="Name">
       </div>
    </body>
</html>
```

Figure 2 shows the `TextBox` widget.

Figure 2. TextBox Widget

Declarative versus programmatic

Dojo supports two programming models, declarative and programmatic. Both models can be used on the same page if needed. In Listing 7, the widget is created in a declarative model.

```
<div dojoType="widgets.TextBox" label="Name"></div>
```

The same widget could be created in a programmatic model, as shown iin Listing 8.

Listing 8. Programmatic creation of the TextBox widget

```
<html>
   <head>
      <!-- page title -->
      <title>TextBox Widget</title>
      <!-- include the DOJO -->
      <script type="text/javascript" src="../dojo-1.0.0/dojo/dojo.js"
                      djConfig="isDebug: true, parseOnLoad: true">
      </script>

      <!-- import DOJO base CSS, DIJIT theme, and widget CSS -->
      <style type="text/css">
         @import "../dojo-1.0.0/dojo/resources/dojo.css";
         @import "../dojo-1.0.0/dijit/themes/tundra/tundra.css";
         @import "templates/TextBox.css";
      </style>

      <!-- import DOJO stuff -->
      <script type="text/javascript">
         dojo.require("dojo.parser");
         <!-- register our module path -->
         dojo.registerModulePath("widgets", "../../widget");
         <!-- import our stuff -->
         dojo.require("widgets.TextBox");
      </script>

      <script type="text/javascript">
         function createWidget()
         {
```


© Copyright IBM Corporation 2009

```
            var widget = new widgets.TextBox();
            widget.label = "Name";
            widget.startup();
         }
      </script>
   </head>

   <body class="tundra ibm" style="padding:5px">
   <br/>
      <!-- declare the DOJO widget -->
      <script type="text/javascript">
         createWidget();
      </script>
   </body>
</html>
```

More on Dojo widgets

As previously stated, a Dojo widget is a Dojo class declaration that inherits from a special class: the `dijit._Widget` class. This class defines the basic behavior for a widget and provides the common functions shared by all the widget implementations. The most important tasks implemented by the _Widget class are:

- Defining the `create` method, which is called automatically when the widget is instantiated; this method executes all the needed creation steps
- Defining some `template` methods, which provide special hooks for widget implementers, giving them the chance to implement special initialization actions in a specific creation phase
- Defining a bunch of `destroy` methods, which clean up the allocated widget's resources
- Defining `event management` methods, which wire-up the widget methods with the DOM node/method call events

The widget creation phase

The `create` method performs the following basic steps for the widget initialization:

Create a unique ID for the widget, if not provided

All Dojo widgets must be identified by a unique ID. When a widget is instantiated, a widget ID can be provided. If not, Dojo creates an ID having the following form:

packageName_className_number

For example, a button could have the following auto-generated ID: dijit_form_Button_0.

© Copyright IBM Corporation 2009

The *packageName* is the widget package with the dot replaced by underscores (for example, dijit_form), and *number* is a progressive number incremented on a widget basis.

Register the widget in the global widget registry

Widgets can be looked up later using the `dijit.byId` method, which returns the widget object corresponding to the given ID. The registry is an Object that lives in the global space at dijit.registry_hash. Hacking in this registry using debugging tools like FireBug (see Resources) could be useful to track, for example, leaking widgets.

Map the attributes

Widget properties can be remapped to user-defined nodes of the widget template. The attributes to map and their target can be listed in the `attributeMap` object. Each property of these objects defines a mapping in the following way:

1. The property name identifies the widget property name where the value to set is stored.
2. The property value identifies the widget property name where the target DOM node is stored. If an empty string is specified the `domNode` is used.

Listing 9 shows an example.

Listing 9. Declaring a dojo widget

```
dojo.declare(
 "MyWidget",
   dijit._Widget,
 {
 // the attribute to map (name and value)
 title: "myTitle",

 // the DOM node to be used to set the
 // title attribute to the "myTitle" value
 titleNode: null,

 // the attribute map
 // the title attribute is mapped to the titleNode
       attributeMap: {title: "titleNode"},
// the template string
       templateString: "<div><span dojoAttachPoint=\"titleNode\"></span></div>",
```

The resulting widget template will be:

```
<div><span title="myTitle"></span></div>
```

Note that because the `base _Widget` class already maps some basic HTML attributes (like the ID, class, style), implementers should not override the

© Copyright IBM Corporation 2009

`attributeMap` property. Instead they must mix the base class `attributeMap` property with their values. Listing 10 shows an example retrieved from the `_FormWidget` class.

Listing 10. Mixing base class attributeMap property with custom values

```
attributeMap:
 dojo.mixin(
 dojo.clone(dijit._Widget.prototype.attributeMap),
 {id:"focusNode", tabIndex:"focusNode", alt:"focusNode"}
),
```

The template methods

During the widget creation phase, some methods are called to give implementers a chance to perform some actions in specific initialization phases. The following methods are invoked:

1. `postMixInProperties`: This method is called after all the widget parameters have been set (mixed in) as widget instance properties. This is a good time to perform further property initializations, if based on properties provided by the instantiate.
2. `buildRendering`: This method is called when it's time to produce rendering for the widget. This hook is implemented by the `dijit._Templated` mix-in class, which extends bare widgets with HTML templates. Implementers can potentially provide their implementation to refine the `_Templated` class job.
3. `postCreate`: The method is called after the widget's DOM is ready and inserted in the page. The implementer can operate on the widget's DOM structure at this point. Children widgets have not yet been started.
4. `startup`: The final chance to operate on the widget. At this time, children widgets have been started.
5. `uninitialize`: Called when the widget is being destroyed. The implementer has a chance to perform some non-automatic clean-up.

Template methods can be provided by the implementer, who must not forget that the superclass can potentially have implemented its template methods. To guarantee the correct initialization for all the classes in the chain, the implementer must manually code the superclass method invocation as shown in Listing 11.

Listing 11. Example of superclass method invocation.

© Copyright IBM Corporation 2009

```
startup: function()
{
 // call the superclass' method
   this.inherited("startup", arguments);
 // your code here
}
```

As a general convention, for all the creation methods (all the template methods except for `uninitialize`), the superclass' method should be called as first to be sure that superclass' items are correctly initialized before working on them.

The inverse rule should be followed by the `uninitialize` method.

A final note should be said for the `startup` method. Don't forget that the startup method is automatically called by the `dojo.parser.parse` method, in case the widget is being created through a declaration. This is not done if the widget is programmatically created and must be manually done like in the example below.

```
var w = new MyWidget({});
w.startup();
```

The widget destroy phase

After being created, widgets live until an explicit destroy request is executed. Implementers are in charge of managing the whole widget lifecycle, including the widget destruction. Otherwise, it will result in a leaked widget until the whole page clean-up happens. The widget provides four destroy methods:

- `destroyRendering`: This method cleans up the rendering items of the widget. Unless the implementer needs a partial clean-up, it is usually called by the other destroy methods.
- `destroyDescendants`: Destroys all the children widgets. Unless the implementer needs a partial clean-up, the method is called by the `destroyRecursive` method.
- `destroy`: Destroys the widget items (not children widgets).
- `destroyRecursive`: Destroys the widget items and children widgets. This method must be called if the widget contains inner widgets.

In the case where the widget contains inner widgets (for example, there are some widgets in its template), the implementer must not forget to trigger the children, destroying them in some way without making the widget user decide which destroy method to call. Listing 12 shows one possible way to do that.

Listing 12. Example of triggering the destruction of a widget

© Copyright IBM Corporation 2009

```
uninitialize: function()
{
 // destroy the descendants
 this.destroyDescendants();
 // call the superclass' method
   this.inherited("uninitialize", arguments);
}
```

Event management

A Dojo widget can react to external events that could come from both DOM nodes and objects. Such events can be manually connected by using the widget's `connect` method, which has the following signature (which is quite similar to the `dojo.connect` method):

```
connect: function(/*Object|null*/ obj, /*String*/ event, /*String|Function*/ method);
```

Or, it can be automatically connected by declaring the connection in the template using the `dojoAttachEvent` attribute. In both cases, the connection is internally tracked by the widget in the _connects array. All the connections will be automatically disconnected at destruction time. In this way, the references between the widget and the DOM nodes are broken.

If, during the widget lifecycle, some connections are not needed any longer, the implementer can manually disconnect them by calling the `disconnect` method to reduce the event handling.

Widget template

As previously seen, widgets must inherit from the `dijit._Widget` class, which defines and provides the basic behavior for a Dojo widget. Such a base class defines the `buildRendering` method that is responsible for building the widget rendering elements. For instance, a widget implementer can create in this method the widget mark-up and set it into the widget DOM node. Another option could be to create the widget structure by using the DOM API. In both cases, the implementer must program in some way the `buildRendering` method.

Dojo provides a powerful abstraction that decouples the widget rendering definition from the widget behavior implementation: the `dijit-._Templated` mix-in class. Implementers who want to exploit an abstraction such as this need to inherit from the `_Templated` class, like in the Listing 13.

Listing 13. Code to inherit from the _Templated class

```
dojo.declare(
 "widgets.MyWidget",
   dijit._Widget, dijit._Templated
 {
```


© Copyright IBM Corporation 2009

```
 templatePath: dojo.moduleUrl("widgets", "templates/MyWidget.html"),
 }
);
```

The `_Templated` class provides its own `buildRendering` implementation that leverages an HTML-like definition. This definition can potentially reside in two different places.

1. An external file. In this case, the file is referred by the *templatePath* property.
2. An internal string property. In this case, the template is directly defined in the widget in the `templateString` property. If the `templateString` is specified, the `templatePath` is ignored, even if specified.

The first option represents the cleanest way to organize the widget source code, because the mark-up can be written in a different file, can be formatted in a readable way without being worried about string concatenation, and string-delimiters must not be escaped. On the other hand, the second option is better performing, because a second file does not need to be loaded from the browser. However, you shouldn't worry about that because Dojo provides a build tool that internalizes external templates into the widget source code.

A template can be parameterized by referring widget properties. This is useful when the widget exposes some configuration parameters that directly affect the mark-up or if the mark-up must be conditionally produced depending on external preferences. Widget properties are referred by using the following syntax: `${propertyName}`.

Templates can contain other widget declarations. However, for them to be considered, the widget developer must set the `widgetsInTemplate` property to true, which by default is set to false to skip a processing that may not be needed.

Templates can contain the two following special attribute declarations:

- *dojoAttachPoint*: This tag attribute, if specified, must be set to a widget property name. The DOM corresponding to the tag will be set to the widget property. Several tags in the template can have one or more attach points.
- *dojoAttachEvent*: This tag attribute lists widget methods that must be called back when a DOM event is triggered.

Listing 14 shows a template example.

Listing 14. Template example of special attribute declarations

© Copyright IBM Corporation 2009

```
<span
 class="textInputDefault"
 dojoAttachEvent="onclick:_onClick">
 <img
 class="textInputIcon"
 src="${constants.DEFAULT_ICON}"
 dojoAttachPoint="_iconNode"/>
 <input
 class="textInputNode"
 size="${size}"
 type="${type}"
 value=""
 dojoAttachPoint="_inputNode, _focusNode"
 dojoAttachEvent="onkeyup:_onKeyUp, onkeypress:_onKeyPress, onchange:_onChange"
 />
 <span
 class="textInputRight"> 
 </span>
</span>
```

Tivoli Dynamic Workload Console and Dojo

The Tivoli Dynamic Workload Console (TDWC) is the graphical user interface of the Tivoli Workload Scheduler (TWS). This section shows how the TDWC v.8.5 takes advantage of the Dojo capability.

Introduction to Tivoli Workload Scheduler

The TWS is a production automation solution designed to help manage workloads in today's complex operating environments. The main scheduling elements are jobs and job streams. A *job* is the representation of a task, such as an executable file, program, or command. A *job stream* represents a container for related jobs and organizes them in terms of run time, sequencing, concurrency limitations, and repetitions. The TWS helps plan jobs for execution, resolves interdependencies, and launches and tracks each job. This is because jobs begin as soon as their dependencies are satisfied; Therefore, idle time is minimized, and throughput improves significantly. Jobs never run out of sequence, and if a job fails, the TWS handles the recovery process with little or no operator intervention.

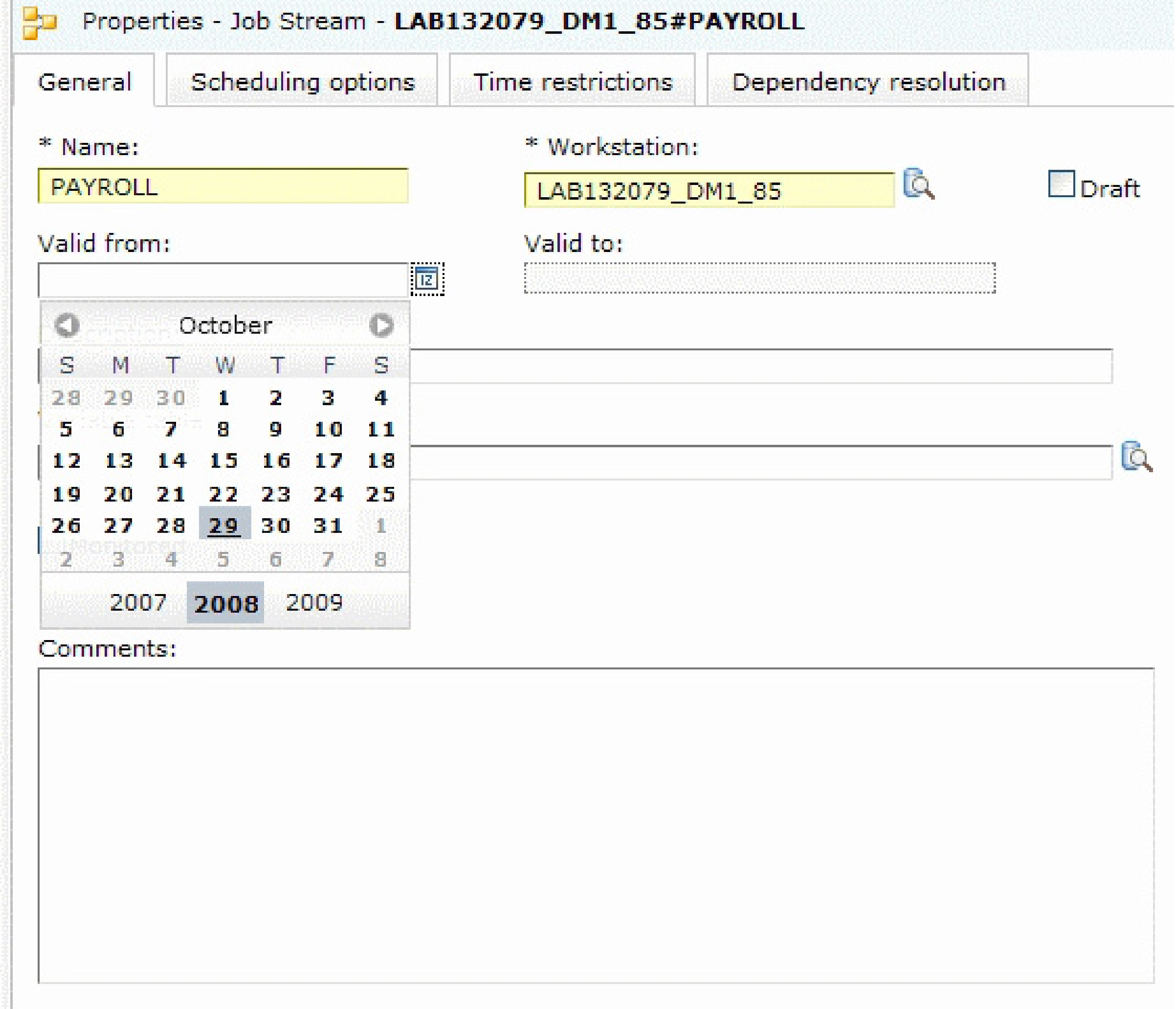
Using Dojo

The TDWC v.8.5 includes a fully-featured workload editor written entirely in JavaScript using the Dojo Toolkit. This kind of implementation allowed moving "intelligence" and behavior closer to the user, using code that runs as much as possible inside the browser.

Take a look at an example of a properties panel of a TWS job stream displayed by the TDWC. Figure 3 represents the Web panel of the general properties defined for the job stream PAYROLL.

© Copyright IBM Corporation 2009

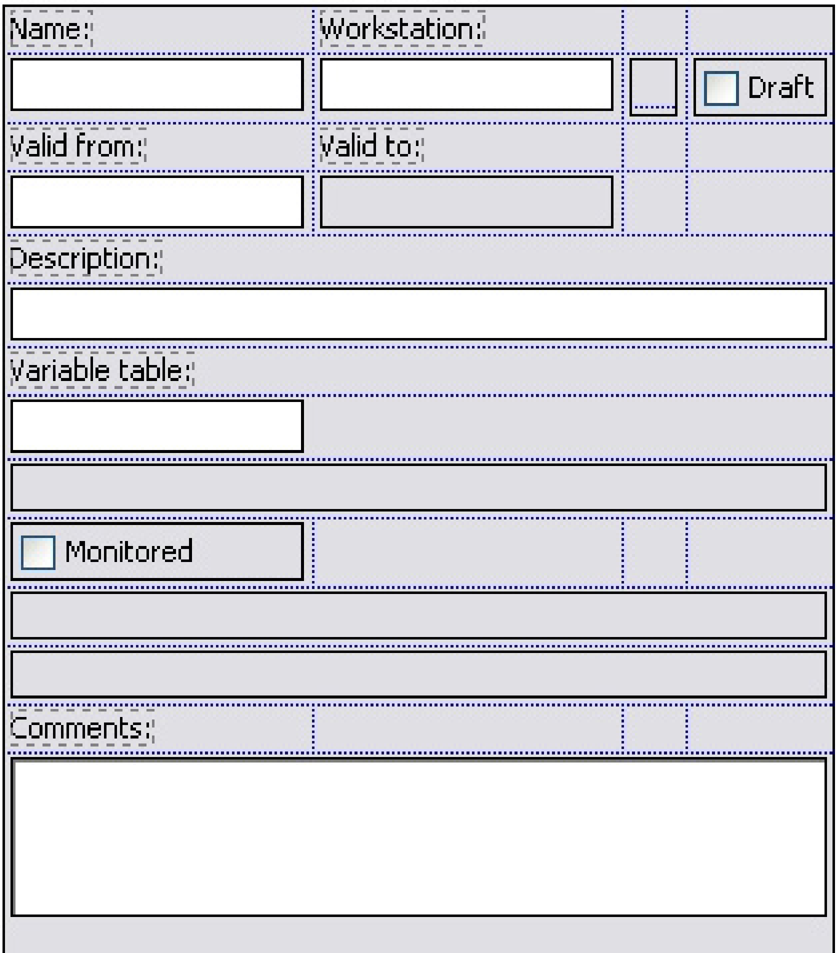
Figure 3. Web Panel for job stream PAYROLL



In a previous article ("The Abstract User Interface Markup Language Web Toolkit: An AUIML renderer for JavaScript and Dojo", see Resources), we described how to design the panel using the AUIML toolkit and how to implement the logical code in JavaScript using the AUIML Web Toolkit (AWT). Figure 4 represents the AUIML panel:

Figure 4. AUIML panel

© Copyright IBM Corporation 2009



Let's focus on the *Valid from* field. It has been defined as Edit Box, Date in the AUIML editor; the AWT couples this element with the HTML code showin in Listing 15.

Listing 15. HTML code to couple the element

```
<span type='text' dojoType='ajaxcommon.widgets.DateInputBox' id='validFrom'>
<script type='dojo/method'event='onValueChanged'>AWT.dispatchOnChange(this.id);</script>
</span>
```

The widget *ajaxcommon.widgets.DateInputBox* is defined in Listing 16:

Listing 16. Code to define the ajaxcommon.widgets.DateInputBox

```
dojo.provide("ajaxcommon.widgets.DateInputBox");

dojo.require("ajaxcommon.resources.Images");
dojo.require("ajaxcommon.widgets._DateTimePicker");
dojo.require("ajaxcommon.widgets.picker.PickerInputBox");
dojo.require("ajaxcommon.widgets.picker.DatePicker");
dojo.declare(
```


© Copyright IBM Corporation 2009

```
    "ajaxcommon.widgets.DateInputBox",
      [ajaxcommon.widgets.picker.PickerInputBox,
       ajaxcommon.widgets._DateTimePicker],
      {
       /** the picker icon */
       pickerIcon: ajaxcommon.resources.Images.get()["CALENDAR_ICON"],
       /** the picker disabled icon */
       pickerDisabledIcon: ajaxcommon.resources.Images.get()["DISABLED_CALENDAR_ICON"],
       /** the picker icon title */
       pickerIconTitle: ajaxcommon.resources.Labels.get()["PICK_DATE"],
       /** constraints */
       constraints: {selector: "date", formatLength: "short", wideYear: true},

       /**
        * Constructor.
        */
       constructor: function()
       {
          // even if the format length is short, ensure the wide year (yyyy)
          // by overriding the datePattern
          if (this.constraints.formatLength === "short" && this.constraints.wideYear) {
             this.constraints.datePattern = this._getWideDatePattern();
          }
          // set the regex for the text
          this.textRegExp = "^(" + dojo.date.locale.regexp(this.constraints) + "){0,1}$";
          // set the regex message for the text
          this.textRegExpMessage =
             this._labels.format("DATE_INVALID_VALUE", {example: this._getExampleValue()});
          // set the picker class
          this.pickerClass = "ajaxcommon.widgets.picker.DatePicker";
       },

       /**
        * Returns the date pattern with the wide-year (yyyy)
        */
       _getWideDatePattern: function()
       {
          // get the bundle
          var bundle = dojo.date.locale._getGregorianBundle();
          // get the pattern
            var pattern = bundle["dateFormat-short"];
            // replace the yy to yyyy if not yet yyyy
            if ( pattern.search(/yyyy/gi) === -1 ) {
               // the year is not in the wide form
               pattern = pattern.replace(/yy/gi, "yyyy");
            }
            return pattern;
       },
      /**
        * Returns a string containing an example of accepted value.
        */
       _getExampleValue: function()
       {
          // get a sample date object (my birthday!!!)
          var d = new Date();
          d.setDate(15);
          d.setMonth(4);
          d.setYear(1971);
          // format the date in the current locale and return
          return dojo.date.locale.format(d, this.constraints);
       }
    }
);
```

The constructor allows the initialization the widget's properties declared in the superclass widget `ajaxcommon.widgets.picker.PickerInputBox` that is

© Copyright IBM Corporation 2009

defined in Listing 17.

Listing 18 shows the related widget template.

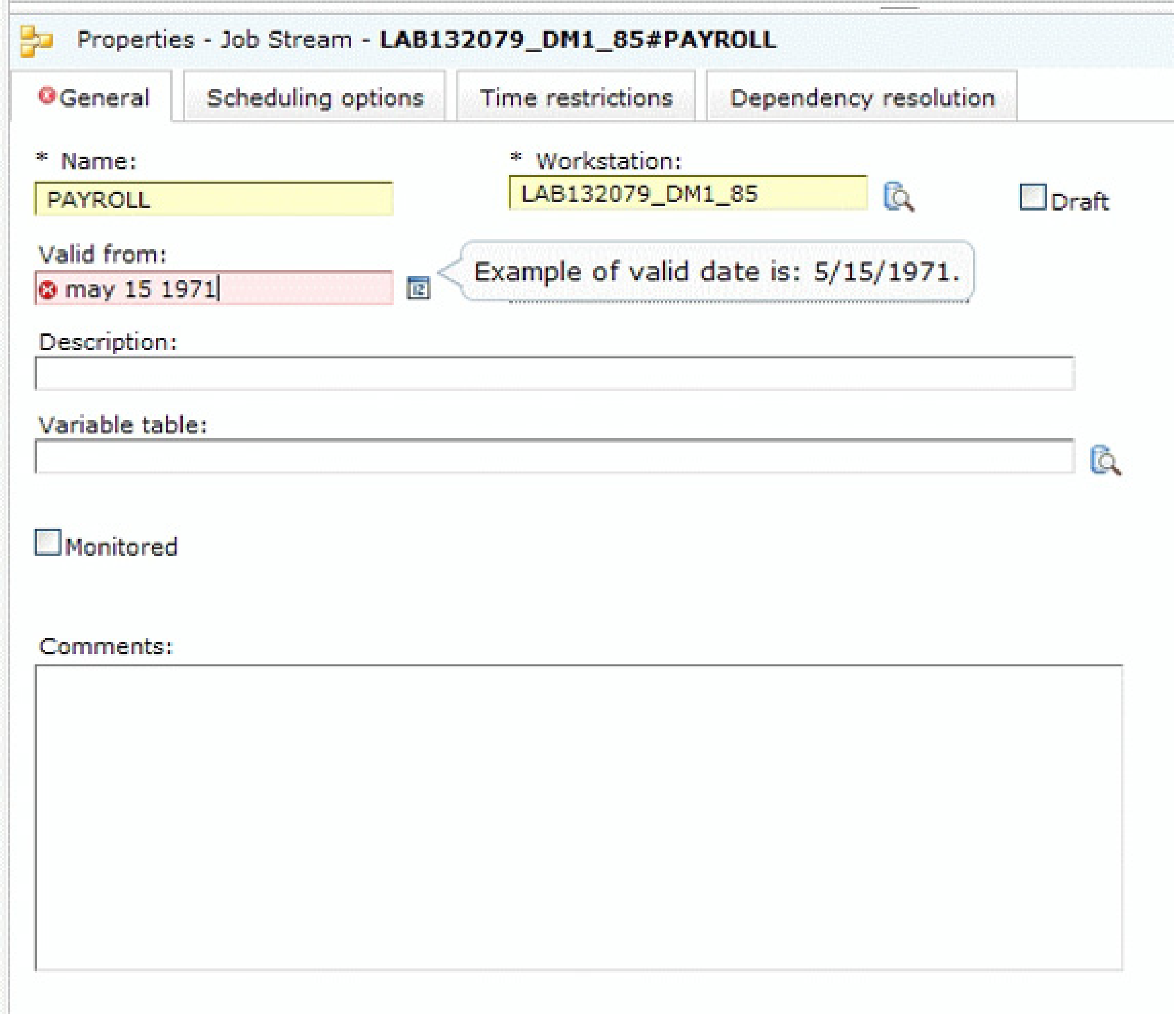
Listing 18. Widget template for ajaxcommon.widgets.picker.PickerInputbox

```
<span
    class="picker"
    ><span
        tabindex="${tabindex}"
        dojoType="${_textBoxWidget}"
        dojoAttachPoint="_textNode,_focusNode"
        required="${required}"
        size="${size}"
        minLength="${textMinLength}"
        maxLength="${textMaxLength}"
        regExp="${textRegExp}"
        regExpReference="${textRegExpReference}"
        regExpMessage="${textRegExpMessage}"
        minWidth="${minWidth}"
    ></span
    ><span
        dojoAttachPoint="_pickerButtonContainer">
        <img
            tabindex="${tabindex}"
            src="${pickerIcon}"
            dojoAttachPoint="_pickerButtonNode"
            dojoAttachEvent="onclick:_onClick,onkeyup:_onKeyUp"
            class="pickerButton"
            title="${pickerIconTitle}"
    /></span>
</span>
```

Figure 5 shows the advantage of delegating the widget logic to the browser side. The widget has been implemented to display the related error message and to change the look and feel in case any error occurs while setting its value.

Figure 5. panel showing browser-side widget logic

© Copyright IBM Corporation 2009



Conclusion

This article described basic concepts for developing HTML widgets using the Dojo JavaScript toolkit, starting from version 1.0 and starting with simple widgets and moving up to more complex ones (such as the widgets implemented in the TDWC v.8.5). The article also highlighted the strong points of the Dojo toolkit.

In a next article, we will review common performance pitfalls encountered during the development of rich Internet applications, and will show how to address them using the best practices we have developed, mainly based on the Dojo toolkit.

© Copyright IBM Corporation 2009

Resources

- Check out developerWorks blogs and get involved in the developerWorks community.
- Download the latest released version of the Dojo toolkit (dojo-release-1.x.x-src.zip or dojo-release-1.x.x-src.tar.gz).
- Download FireBug from the project Web site.

About the authors

Marco Lerro
Marco has been a staff engineer at the IBM Rome Tivoli Lab since 1999. He is currently involved in the TDWC design and development. His current area of interest is in Web 2.0 development, with a particular emphasis on Dojo-based technologies and user-experience design patterns.

Roberto Longobardi
Roberto has been working in the software development area for twelve years, covering several roles ranging from developer, analyst, Web interface designer, and interaction designer. He is currently working in the Tivoli organization as a Web user interface designer for the Tivoli Workload Scheduler and Dynamic Workload Broker products. As such, he is also often involved in client-facing activities such as requirements gathering, design validation, and sales enablement. Roberto has published several articles on developerWorks, presented at IEEE conferences, and is involved in several academic collaborations.

Gianluca Perreca
Gianluca Perreca is a staff engineer at the IBM Rome Tivoli Lab. He is a member of the Tivoli Workload Scheduler development team, and he has been actively involved in the design and implementation of the IBM Tivoli Dynamic Workload Console, the Web interface for IBM Tivoli Workload Scheduler.

© Copyright IBM Corporation 2009

Alessandro Scotti
Alessandro Scotti is the chief designer of the IBM Tivoli Dynamic Workload Console, a Web interface for IBM Tivoli Workload Scheduler that leverages the AUIML Toolkit as well as AUIML and other Web 2.0 technologies. Alessandro has authored or contributed to several open source projects, mostly in the areas of graphical user interface, high-performance, and real-time programming.

© Copyright IBM Corporation 2009