EPiServer: writing Dojo widget

Fri Apr 11 2014

I was working on open source library Geta Tags (https://github.com/Geta/Tags) and wanted to improve editor user experience. Starting from EPiServer 7, EPiServer uses Dojo Toolkit (http://dojotoolkit.org/) for administrative interface. Searching for tutorials and articles on how to create Dojo widgets for EPiServer didn't get expected results. So I had to get through all the Dojo creation and configuration process through trial and errors. In this article I will try to describe how to create simple and more advanced Dojo widgets for EPiServer.

Problem

Dojo provides lot of *UI controls* which are used by EPiServer, but there are times when you need more advanced user experience. In my case I had to create user friendly tag selection. I found several articles (here (http://world.episerver.com/Blogs/Linus-Ekstrom/Dates/2012/10/Creating-a-Dojo-based-component/) and here (http://world.episerver.com/Blogs/Linus-Ekstrom/Dates/2013/12/Auto-suggest-editor-in-EPiServer-75/)) how to create and extend Dojo widgets - those did not explain much why and how those work, but was good starting point.

For Geta Tags I started with simple widget which extended Dojo MultiComboBox (http://dojotoolkit.org/reference-guide/1.9/dojox/form/MultiComboBox.html). While it work, it didn't provide user friendly interface. So I started to look for solutions. I searched for Dojo widgets which might be used and found nothing, but found a lot of jQuery/jQuery UI plugins for tag rendering. So the only task left was how to make jQuery plugin to work with Dojo in EPiServer.

Dojo widgets

There is a great book about Dojo - Mastering Dojo: JavaScript and Ajax Tools for Great Web Experiences (http://pragprog.com/book/rgdojo/mastering-dojo). This book describes how to work, setup Dojo and also how to extend it.

First thing to understand about widget is it's lifetime - in what order which methods are called. It will let you better understand why something works in particular way. Here is widget lifetime described in the book:

- widget constructor is called
- mixin parameters applied
- postMixinProperties() method called
- id assignment, if not provided
- buildRendering() method called
- copy attribute map
- postCreate() method called
- expand child widgets
- startup() method called

There are several methods which you might want to use to extend the widget. Extensibility points are these methods:

- *postMixinProperties* - it is called after the properties have been initialized. Override default properties or add custom properties in this method.
- *buildRendering* - it gets the template and fills in the details. Usually you will not need to override this method. Default implementation is provided by dijit._Templated (https://dojotoolkit.org/reference-guide/1.9/dijit/_Templated.html#dijit-templated). If you want to handle rendering yourself, then override this method.
- *postCreate* - this is the main extensibility point. Widget has been rendered, but not it's child widgets and not containerNode (https://dojotoolkit.org/documentation/tutorials/1.7/templated/#containerNode). You can set custom attributes, access *this.domNode* (reference to parent node of the widget itself) and manipulate with it in this method.
- *startup* - this is called when widget and all child widgets have been created. This is the place where to access child widgets.

For more information on widget lifetime see these articles:

- Understanding _Widget (http://dojotoolkit.org/documentation/tutorials/1.6/understanding_widget/)
- Chapter 12. Dijit Anatomy and Lifecycle (http://chimera.labs.oreilly.com/books/1234000001819/ch12.html)
- _WidgetBase (https://dojotoolkit.org/reference-guide/1.9/dijit/_WidgetBase.html)

There are two ways how to build your widget - extend existing one or build new widget from scratch. I will explain how to extend existing widget, because it is the easier way to get something done. For creating new widget see article Writing Your Own Widget (https://dojotoolkit.org/reference-guide/1.9/quickstart/writingWidgets.html).

Minimal implementation

To create new widget for EPiServer you will need:

- Dojo widget JavaScript implementation
- EPiServer editor descriptor class for your widget
- properly configured module.config

Dojo widget

Let's start with simple Dojo widget. First version of Dojo widget for Geta Tags (https://github.com/Geta/Tags/blob/v0.9.8/ClientResources/Scripts/Editors/TagsSelection.js) extends Dojo MultiComboBox (http://dojotoolkit.org/reference-guide/1.9/dojox/form/MultiComboBox.html). It allows to select multiple values from autosuggestion and adds them as comma separated string to the input.

```
define([
    "dojo/_base/declare",
    "dojo/store/JsonRest",
    "dojox/form/MultiComboBox"
],
function (
    declare,
    JsonRest,
    MultiComboBox) {

    return declare([MultiComboBox], {
        postMixInProperties: function () {
            var store = new JsonRest({ target: '/getatags' });
            this.set("store", store);
            // call base implementation
            this.inherited(arguments);
        },
        _setValueAttr: function (value) {
            value = value || '';
            if (this.delimiter && value.length != 0) {
                value = value + this.delimiter + " ";
                arguments[0] = this._addPreviousMatches(value);
            }
            this.inherited(arguments);
        }
    });
});
```

Dojo uses AMD for structuring the application. First step is to call *define* function and define what are your dependencies. In this case I am defining *declare*, *JsonRest* and *MultiComboBox*. All three are included in Dojo package and available in EPiServer [1]. Dojo module dependencies are passed in as a first parameter of *define* function. Those are defined as an array of string paths to the modules.

Second parameter of *define* function is function which defines the module. The function has arguments with dependencies defined previously. This function then should call *declare* function which declares your widget.

As a first parameter it takes an array of objects which will be mixed in to your new widget. You can think about this as extending the object you pass in. In this case we will use *MultiComboBox*. The second argument is object which defines your widget. Here you can override methods and properties of your base widget.

For Geta Tags I needed to set *store* property for *MultiComboBox*. The right method to override in this case is *postMixInProperties*. *store* property defines the source for autosuggestion. To use some server side API as a source you have to define *JsonRest* (http://dojotoolkit.org/reference-guide/1.9/dojo/store/JsonRest.html) object by passing in options object which should contain property - *target* with URL to the resource. Then use *set* method of base widget to set the *store* property. The last step is to call base implementation (http://dojotoolkit.org/reference-guide/1.9/dojo/_base/declare.html#calling-superclass-methods) of *postMixInProperties* by calling *inherited(arguments)* (http://dojotoolkit.org/reference-guide/1.9/dojo/_base/declare.html#inherited) method that base widget can do it's stuff.

MultiComboBox has one issue that it do not handle the case when value provided to it is not defined. EPiServer usually do not provide initial value for the field so *MultiComboBox* can't handle it. To solve the issue I redefined *_setValueAttr* function of *MultiComboBox* to set empty value if it is not defined. You can redefine your base widget methods if needed similar way, but be careful - those methods are *internal* and might change in future versions of Dojo.

The last (or first) step is to place the JavaScript file in correct folder. All client scripts for EPiServer modules should be placed in *ClientResources* folder. I would suggest to create new folder with the name of your module and place JavaScript implementation there. For example, *ClientResources/Simple.Module/* folder. First versions of Geta Tags used *ClientResources/Scripts/Editors* folder, but it might collide with other modules. After placing JavaScript module in the correct place, you should configure it that it will be loaded by EPiServer. First step is to add correct path to your module in *module.config*.

module.config

module.config is configuration file for modules used in your project and should be placed in the root of your project (for shell modules in the root folder of shell module (http://world.episerver.com/Documentation/Items/Developers-Guide/EPiServer-Framework/75/Modules/Modules/)). For more information see module.config documentation (http://world.episerver.com/Documentation/Items/Developers-Guide/EPiServer-Framework/75/Configuration/Configuring-moduleconfig/).

Basic configuration for Dojo module is simple - you have to register assembly of the project which defines editor descriptor and register path to your dojo module. First version of Geta Tags module.config looked like this:

```
<?xml version="1.0" encoding="utf-8"?>
<module>
  <assemblies>
    <add assembly="Geta.Tags" />
  </assemblies>

  <dojoModules>
    <add name="geta" path="Scripts" />
  </dojoModules>
</module>
```

First of all I am defining Geta.Tags assembly and then dojo module - setting name of the module and path to the folder which contains dojo module scripts. The folder is relative to *ClientResources* folder. In this case folder path is - *ClientResources/Scripts/*.

Editor descriptor

Last step is creating editor descriptor. This class *connects* EPiServer content type property with your Dojo module.

Here is the Geta Tags editor descriptor:

```
[EditorDescriptorRegistration(TargetType = typeof(string), UIHint = "Tags")]
public class TagsEditorSelectionEditorDescriptor : EditorDescriptor
{
    public TagsEditorSelectionEditorDescriptor()
    {
        ClientEditingClass = "geta.editors.TagsSelection";
    }
}
```

Create the class which derives from EditorDescriptor class. Decorate class with EditorDescriptorRegistration attribute and provide the type of the property and UIHint. UIHint is just the string name of the UIHint attribute which you will use later on the content type's property. EPiServer checks the property's UIHint and EditorDescriptor's UIHint and applies the editor descriptor to the property if they match.

Then in the constructor of the class set *ClientEditingClass* which is in form of *{module name}.{path relative to module root}.{Dojo module file name}*. For Geta Tags this is *geta.editors.TagsSelection*:

- *geta* is the module name we defined in *module.config*
- *editors* is the folder relative to *Scripts* folder (it is defined as the root folder for module in *module.config*)
- *TagsSelection* is Dojo module file name without extension

After defining editor descriptor you can use your custom property in content type:

```
[UIHint("Tags")]
public virtual string Tags { get; set; }
```

Just decorate your property with *UIHint* attribute with value you defined in editor descriptor. Now custom Dojo widget will be used when you start editing the field.

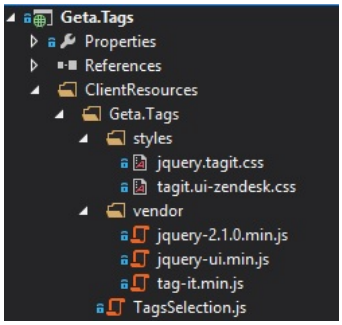
Advanced implementation

While you can create complex Dojo widgets yourself, most of the times it is better and faster to use existing libraries. Dojo do not have much custom plugins, but jQuery and jQuery UI [2] have and it would be great to use them in your Dojo widgets.

Referencing scripts and CSS stylesheets

In usual Web project referencing script libraries is simple - just add script tags in your markup and start using the plugins, but in EPiServer's Dojo implementation it is little bit different. You can't get to the markup and add scripts manually to EPiServer administration interface.

There are two ways to use libraries - use CDN or use scripts from project. When you want to include the script in the project, you have to put them in your module folder which is defined in *module.config*, for example: *ClientResources/Geta.Tags* . Same applies for CSS. You can organize stylesheets and scripts in subfolders. For example, Geta Tags have *styles* folder for CSS and *vendor* folder for 3rd party libraries:



Next step is informing EPiServer to load these scripts and stylesheets when EPiServer administrative interface loads. This can be done in *module.config*:

```
<module>
  <assemblies>
    <add assembly="Geta.Tags" />
  </assemblies>

  <clientResources>
    <add name="geta-tags-vendor" resourceType="Script" sortIndex="1"
        path="Geta.Tags/vendor/jquery-2.1.0.min.js"  />
    <add name="geta-tags-vendor" resourceType="Script" sortIndex="2"
        path="Geta.Tags/vendor/jquery-ui.min.js"  />
    <add name="geta-tags-vendor" resourceType="Script" sortIndex="3"
        path="Geta.Tags/vendor/tag-it.min.js"  />
    <add name="geta-tags-styles" resourceType="Style" sortIndex="1"
        path="Geta.Tags/styles/jquery.tagit.css"  />
    <add name="geta-tags-styles" resourceType="Style" sortIndex="2"
        path="Geta.Tags/styles/tagit.ui-zendesk.css"  />
  </clientResources>

  <dojoModules>
    <add name="geta-tags" path="Geta.Tags" />
  </dojoModules>

  <clientModule>
    <moduleDependencies>
      <add dependency="CMS" type="RunAfter" />
    </moduleDependencies>
    <requiredResources>
      <add name="geta-tags-vendor" />
      <add name="geta-tags-styles" />
    </requiredResources>
  </clientModule>
</module>
```

Define *clientResources* section and add all scripts and styles using *add* tag. For each resource define:

- *name* - this is used to distinguish between different resources. You can use different name for each resource or use same name for group of resources. In the Geta Tags I used one name for scripts and other for styles.
- *resourceType* - type of the resource - *Script* or *Style*.
- *sortIndex* - the order in which resource will be rendered for resources with same name.
- *path* - path to the resource. The path can be full URL or local path. Full URL is useful to define resources from CDN. Local path is used for resources in your project. It is relative to *ClientResources* folder.

After that you have to define loading of these resources:

- define *moduleDependencies* section under *clientModule* section and add dependency on *CMS* with type *RunAfter*. *RunAfter* will inform EPiServer to render resources after EPiServer administration is loaded. If you skip this configuration, then your resources will not be loaded at all.
- define *requiredResources* section under *clientModule* section and add the names of your defined *clientResources*.

This is all we need to get your scripts loaded.

Using jQuery plugin in Dojo widget

When you have all libraries in place, it is easy to start using them. Each Dojo widget hase *domNode* which can be used to access widget's DOM node using jQuery. It means that jQuery plugin can be attached to this DOM node. In Geta Tags I am using tag-it (https://github.com/aehlke/tag-it) jQuery UI plugin to handle multiple tags:

```
define([
    "dojo/_base/declare",
    "dijit/form/TextBox"
],
function (
    declare,
    TextBox) {

    return declare([TextBox], {
        postCreate: function() {
            $(this.domNode).find('input').tagit({
                autocomplete: { delay: 0, minLength: 2, source: '/getatags' }
            });
        }
    });
});
```

First of all I am extending Dojo TextBox widget and overriding *postCreate* method with my implementation. *postCreate* in this case is the most appropriate place to put your functionality, because I can access *domNode* here. Dojo TextBox HTML has several wrappers around *input* so I have to find input and I am using jQuery for that. After that I am applying Tag-it plugin on the input field.

The widget implementation becomes much simpler and it is much easier to achieve better user experience using custom plugins.

Summary

Writing Dojo widget is not hard, but finding the way how to wire every piece together might take hours and hours. Also now I can see that two different frameworks - Dojo and jQuery can play together and make great user experience. I hope this article will help others and me too to implement new Dojo widgets and not fear of extending EPiServer administration user interface.

1. You can find all available Dojo modules in EPiServer VPP/appData folder: *Modules\Shell\3.0.1209\ClientResources*.
2. You are not limited to jQuery - use any JavaScript library you want.

Hi! I'm Māris Krivtežs. I am Web developer primarily working with ASP.NET, EPiServer and front-end development at Geta (http://geta.no), but same time learning different programming languages, learning new programming paradigms, architectures, and design. I spend my free time with my family, reading books, bicycling and traveling.

(http://stackoverflow.com/users/660154/marisks)(https://github.com/marisks)(https://twitter.com/pioners1001)(http://www.linkedin.com/pub/maris-krivtezs/31/98b/b53)(/rss.xml)