|  |
| --- |
| **Web forms to MVC Refactoring guide** |
|  |
|  |
| Provides a comprehensive step by step guide to help you refactor your existing ASP.Net web forms controls to the new MVC pattern. Also provides instructions to refactor your MS Ajax based JavaScript files to jQuery based plugin architecture. |
|  |
| **Hrusikesh Panda** |
| **11/15/2010** |
|  |

At a glance

[Assumptions/Requirements 3](#_Toc278989613)

[Building a Canvas Module 3](#_Toc278989614)

[Canvas Module Files 3](#_Toc278989615)

[Canvas Module Files Explained 3](#_Toc278989616)

[DemoCanvasModule.cs 3](#_Toc278989617)

[DemoCanvasModule.css 4](#_Toc278989618)

[DemoCanvasModule.js 4](#_Toc278989619)

[DemoModuleState.cs 4](#_Toc278989620)

[JavaScript Refactoring 5](#_Toc278989621)

[General Template 5](#_Toc278989622)

[Template Parts Explained 6](#_Toc278989623)

[Deriving from DJ.UI.AbstractCanvasModule 6](#_Toc278989624)

[Plugin definition 6](#_Toc278989625)

[Plugin initialization 6](#_Toc278989626)

[Public and private methods 7](#_Toc278989627)

[Plugin Authoring Best Practices 8](#_Toc278989628)

[Naming your plugin 8](#_Toc278989629)

[Name spacing events 8](#_Toc278989630)

[jQuery plugin development Best practices 8](#_Toc278989631)

[jQuery Plugin Authoring References 8](#_Toc278989632)

Web forms to MVC Refactoring guide

# Assumptions/Requirements

This guide assumes that you have the following installed on your system and are familiar with them:

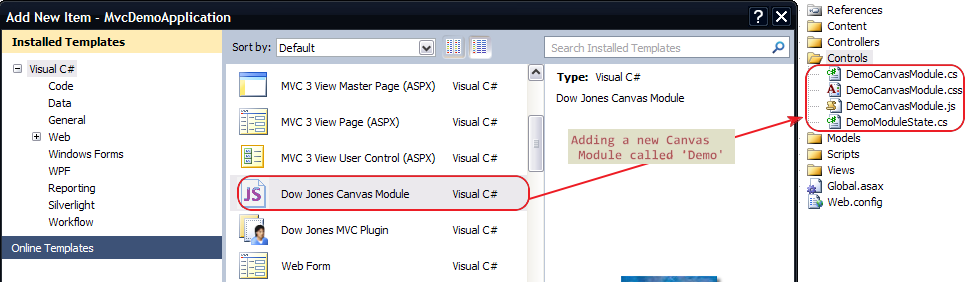
* Visual Studio 2010
* ASP.Net MVC 3 Framework
* Razor View Engine
* jQuery Library

# Building a Canvas Module

## Canvas Module Files

* DemoCanvasModule.cs
* DemoCanvasModule.css
* DemoCanvasModule.js
* DemoModuleState.cs

These names are auto-generated if you add a new item using the “Dow Jones Canvas Module” template (as shown below). It only shows the recommended practice and the names can be changed to anything you like.



## Canvas Module Files Explained

The following section summarizes the role of each file. The auto-generated files are self documented and explain each section in the file elaborately.

### DemoCanvasModule.cs

A canvas module derives from’ AbstractCanvasModule’. This is the main codebehind file responsible for rendering the HTML markup of the module. To generate your markup, override ‘WriteBody’ method of base class. This will place the markup inside a div generated in the base class.

If you prefer to have full control of the HTML (and basically do not want the parent div), then you can override ‘WriteHtml’ method of base class instead of ‘WriteBody’.

Apart from rendering HTML, this file also helps wiring up the .js and .css files as embedded resources for your control. If you’re worried about performance and network traffic, then the good news is that the framework handles resource combining and caching of these embedded resources.

### DemoCanvasModule.css

Include any control specific style definitions here. This file is mostly not required as controls rarely have their own custom styles. The styles should be governed by the hosting Page and the module should obey those unless there is a strong reason not to.

### DemoCanvasModule.js

This is where the JavaScript for your control resides. It follows the jQuery plugin model of development. This should derive from ‘DJ.UI.AbstractCanvasModule’ which exposes some common functionality and basic properties. Some examples are automatically binding meta information such as data, tokens and event handlers. Gives you basic methods such as ‘paint’ and ‘clear’ which can be overridden by the developer. See the auto-generated file for details on each specific property, method as well as best practices to create your own methods.

### DemoModuleState.cs

This is an important file as it contains the data, client events, tokens and options for the module.

* Data can be any class that the control understands (such as ControlData, HeadlineListDataResult etc.). It is optional to populate this. This data is rendered on client side in JSON format. So you can use it in place of ViewState or to write any other control data that might be needed on client side.
* Client Events is basically a Dictionary<string, string> of client event names and the name of the JavaScript event handler
* Tokens is a class that derives from IDictionary<string, object> and can be used on client side for templating
* Options is a class that derives from IDictionary<string, object> and can be used on client side for controlling UI options such as showing/hiding sections, flags, any additional meta data associated with the module

# JavaScript Refactoring

## General Template

A sample boiler plate code for migrating your AJAXToolkit based JavaScript library to jQuery plugin is listed below. All of this boiler plate code is needed to leverage the full capabilities of jQuery and DJ base object.

/\*!  
\* DemoCanvasModule  
\*/  
   
(function ($) {  
   
    DJ.UI.DemoCanvasModule = DJ.UI.AbstractCanvasModule.extend({  
   
        /\*  
        \* Properties  
        \*/  
   
        // Default options  
        defaults: {  
            debug: false,  
            cssClass: 'DemoCanvasModule'  
            // ,name: value     // add more defaults here separated by comma  
        },  
   
        // Localization/Templating tokens  
        tokens: {  
        // name: value     // add more defaults here separated by comma  
        },  
   
   
        /\*  
        \* Initialization (constructor)  
        \*/  
        init: function (element, meta) {  
            var $meta = $.extend({ name: "DemoCanvasModule" }, meta);  
   
            // Call the base constructor  
            this.\_super(element, $meta);  
   
            // TODO: Add custom initialization code like the following:  
            // this.\_testButton = $('.testButton', element).get(0);  
        },  
   
   
        /\*  
        \* Public methods  
        \*/  
   
        // TODO: Public Methods here  
   
   
        /\*  
        \* Private methods  
        \*/  
   
        // DEMO: Overriding the base \_paint method:  
        \_paint: function () {  
   
            // "this.\_super()" is available in all overridden methods  
            // and refers to the base method.  
            this.\_super();  
   
            alert('TODO: implement DemoCanvasModule.\_paint!');  
        },  
   
   
    });  
   
   
    // Declare this class as a jQuery plugin  
    $.plugin('dj\_DemoCanvasModule', DJ.UI.DemoCanvasModule);  
   
   
})(jQuery);

## Template Parts Explained

This section explains the above skeletal code in detail.

### Deriving from DJ.UI.AbstractCanvasModule

To derive from DJ base object, the plugin should start like:

    DJ.UI.DemoCanvasModule = DJ.UI.AbstractCanvasModule.extend({

The base class takes care of common functionality such as wiring up the event handlers, initializing the plugin with defaults, options, showing the loading area, error handling etc.

### Plugin definition

To create a jQuery plugin out of your control, simply follow the typical jQuery plugin definition:

// Declare this class as a jQuery plugin  
    $.plugin('dj\_DemoCanvasModule', DJ.UI.DemoCanvasModule);

### Plugin initialization

To initialize your plugin, call this.\_super() along with the arguments as shown in the ‘init’ method.

        /\*  
        \* Initialization (constructor)  
        \*/  
        init: function (element, meta) {  
            var $meta = $.extend({ name: "DemoCanvasModule" }, meta);  
   
            // Call the base constructor  
            this.\_super(element, $meta);  
   
            // TODO: Add custom initialization code like the following:  
            // this.\_testButton = $('.testButton', element).get(0);  
        },

This takes care of extending the metadata into the plugin object. Sample ‘meta’ looks like this:

{prop1: {…}, options: {…}, propn: {…}, clickHandler: OnClick}

where prop1 and propn are additional properties and can be named anything. You can also add any number of additional properties in ‘meta’

### Public and private methods

You can add your own public and private methods in the sections as marked. Note that you can add them anywhere inside the root function; this is shown as a best practice to maintain readability and maintainability of the code. You can also override base class methods as shown below.

        /\*  
        \* Public methods  
        \*/  
   
        // TODO: Public Methods here

        /\*  
        \* Private methods  
        \*/  
   
        // DEMO: Overriding the base \_paint method:  
        \_paint: function () {  
   
            // "this.\_super()" is available in all overridden methods  
            // and refers to the base method.  
            this.\_super();  
   
            alert('TODO: implement DemoCanvasModule.\_paint!');  
        },

## Plugin Authoring Best Practices

This section enlists the best practices to follow while developing your own jQuery plugin. These are not mandatory things to do but rather recommended rules that will help you avoid common pitfalls and make you life easier.

### Naming your plugin

In order to better distinguish your plugin from other on the page (including external plugins), as a best practice, you should name your plugin as per this pattern: “dj\_ myPlugin”

### Name spacing events

To avoid conflict with events from other authors or even from within other scripts in your project, it is a good practice to add a namespace to your events. This way, if you need to unbind it later, you can do so without interfering with other events that might have been bound to the same type of event. The recommended convention is “dj. myPlugin.eventName”

### jQuery plugin development Best practices

Read more [here](http://docs.jquery.com/Plugins/Authoring#Summary_and_Best_Practices).

## jQuery Plugin Authoring References

* [**How jQuery Works**](http://docs.jquery.com/Tutorials:How_jQuery_Works)

A basic introduction to jQuery and the concepts that you need to know to use it.

* [jQuery Plugins/Authoring](http://docs.jquery.com/Plugins/Authoring)