Dow Jones MVC Framework Overview

# High-Level Components

## DowJonesHttpApplication

The base class for any Dow Jones MVC application. This base class configures the Ninject kernel as well as executes any IBootstrapperTasks it finds in the loaded assemblies.

## Custom Routing (RouteAttribute)

The framework offers a RouteAttribute which provides the ability to apply specific routes to individual actions. Prefer this attribute over adding a route mapping when actions need “one-off” routes that do not follow a specific pattern.

## ScriptRegistry and StylesheetRegistry

These two classes (one each per request) provide a declarative way to register script and stylesheet includes. They can (and do) implement features such as script combining, however their declarative usage abstracts all of this from developers. This declarative nature also allows the Infrastructure team to introduce additional features without changes to additional code.

## IViewComponent (and ViewComponentBase)

ViewComponentBase’s role is to make component development simple and straight-forward by abstracting much of the core logic for rendering a component and its related script and/or stylesheet resources (AKA “client resources”) as well as providing a standard approach for dealing with common functionality such as component options and token handling.

At its core it offers a few important features:

* A powerful inheritance model so all components that inherit from a component implementing the ViewComponentBase class will also be provided with their ancestor’s client resources.
* “Client State Mapping”: extracts important information from the component’s State (Model), allowing any or all of this information to be generated elsewhere and passed via an intermediary object instead of requiring access to an instance of the view component. The mapper supports the following types of data:
  + **Client Data**: the data to be serialized into JSON and rendered to the page for the component’s initialization.
  + **Client Event Handlers:** key-value pair mappings of client event names to the names of their respective Javascript handler functions
  + **Client Properties**: the options sent to the component
  + **Tokens:** placeholder values used for globalization [**Obsolete** – to be removed]

The **RazorViewComponentClassGenerator** – as the name implies – is our custom tool which converts Razor templates into classes that implement **ViewComponentBase** (as opposed to WebViewPage as with the standard MVC views) so that developers can specify view markup in the powerful Razor syntax, yet still realize the benefits of the ViewComponentBase class.

## IViewComponentState (and ViewComponentState)

Since the Canvas requirements dictated that a component be created via a model alone, the IViewComponentState interface and ViewComponentState implementations where introduced to make this task easier via “auto-mapping” of ViewComponentBases and ViewComponentStates. This mapping is a simple convention stating that any component that implements *IViewComponent*<*TViewComponentState*> where “*ViewComponentBase*” is any class implementing IViewComponent and TViewComponentState is any class implementing IViewComponentState.

As the framework grows into more of a general-usage framework, this class becomes less important.

## ViewComponentFactory

This factory serves as a helper to create instances of view components, wiring them into the framework as needed (which usually translates to registering them with the Script and Stylesheet Registries). This class is exposed to views via the Html.DJ() helper method and provides a number of methods to render components in a view. The most notable methods are:

* *Render (object data = Html.ViewData.Model, object htmlAttributes = null)*This method is the entry point to the object -> View Component mappings. Given any object “data”, the ViewComponentFactory will attempt to locate the mapping for its appropriate view component, typically the component following the ViewComponent<TData> convention.
* *RenderComponent (string componentName, object model = Html.ViewData.Model, object htmlAttributes = null)*  
  Like the Html.RenderPartial() helper, this helper method takes the name of a component and an optional model object to pass to the component. The component name is the simple class name of the component (e.g. DowJones.Web.UI.HeadlineListComponent would simply be “HeadlineListComponent”).
* *RenderComponent <TComponent>(object model = Html.ViewData.Model, object htmlAttributes = null)*The generic RenderComponent method overload. Allows the developer to explicitly specify which component type to render instead of looking it up by name.

## Custom Razor Views

Developers may use the Dow Jones MVC View Component item template to create a Razor View that compiles into a class that implements ViewComponentBase. Different flavors of this item template exist, from a very simple (Razor view only) template to the more advanced template that includes a ViewComponentState class, an empty Stylesheet file, and a Javascript file that provides a default template of our inheritance-based client-side component classes. Any or all of these items (e.g. the stylesheet file) can be removed if they are not used in a control.

Custom Razor views currently support the following keywords in addition to most of the standard Razor keywords:

* @**ClientPlugin**: specifies the jQuery plugin name to use when initializing the component. This implements the abstract ClientPluginName property on ViewComponentBase. If left empty, it is null (which means the framework will not emit initialization lines for instances of this component)
* @**ScriptResource**, @**StylesheetResource**, and @**FrameworkResource**: the Razor implementation of the attributes of the same name. Used to register Script and Stylesheet resources with their respective Registries at runtime. These keywords take a semicolon-delimited key-value pair (in the format of [Name]=[Value]) list of the following properties:
  + **Name**: Optional property to help identify the resource within the collection
  + **Url**: the absolute URL of the resource. When specified, this takes priority over all other properties. Though the value of this property may be whatever you like, the Framework assumes it is an external URL.
  + **ResourceName**: the full embedded resource name
  + **RelativeResourceName**: a shortened version of the resource’s **ResourceName**, assuming the path to the resource is the namespace of the current Razor view.
  + **DeclaringType**: the type name of any type in the assembly containing **ResourceName**. This is used to get the correct assembly. If left empty, the type of the current Razor view is assumed.
  + **ResourceKind**: a reference to a **ClientResourceKind** value. Script, Stylesheet, etc. As would be expected, the **ScriptResource** and **StylesheetResource** set the appropriate value for this property.
  + **DependencyLevel**: a reference to a ClientResourceDependencyLevel value indicating the relative importance/priority of this resource. E.g. Global, MidLevel, Component, etc. – links to higher-level resources are rendered first.
  + **PerformSubstitution**: specifies whether or not this resource should be processed on the server-side. E.g. has tokens that need to be replaced or AppSettings to be generated. **[Obsolete** – as of the v1 release, all framework resources participate in the substitution process **]**
* **@ClientTemplate**: a reference to the resource to be used as a client-side JavaScript template. This resource – written using the Underscore syntax – will be compiled on the server side and dynamically injected into the component’s client-side Javascript object. This keyword accepts the following properties:
  + **RelativeResourceName**: a shortened version of the resource’s **ResourceName**, assuming the path to the resource is the namespace of the current Razor view.
  + **TemplateID:** the name of the client-side template. For example, if a ClientTemplate is defined on the Portal Headline List component with a TemplateID of “success”, the framework will dynamically populate the *DJ.UI.PortalHeadlineList.prototype.templates.success* property with the compiled client template. Alternatively, a “fully-qualified” name can be provided in this field, in which case the framework will use that verbatim (e.g. “TemplateID = MyTemplates.success” would translate to “window[‘MyTemplates.success’]”).
  + **PerformSubstitution**: specifies whether or not this resource should be processed on the server-side. E.g. has tokens that need to be replaced or AppSettings to be generated. **[Obsolete** – as of the v1 release, all framework resources participate in the substitution process **]**

# Client-Side Scripting Model

Client-side component development follows the Javascript inheritance model [discussed by Jon Resig in this post](http://ejohn.org/blog/simple-javascript-inheritance/). As such, we offer a few base “classes” which provide core framework functionality:

* **DJ.Component:** offers the default implementation of data-, options-, and event delegate-handling, as well as core helpers such as \_debug, \_publish, and \_subscribe which help simplify these things by scoping them to the current instance. Classes deriving from DJ.Component usually have some kind of data and/or state but are not necessarily visual (DOM-related) classes.
* **DJ.UI.Component**: extends DJ.Component and adds DOM-related functionality such as the ability to “attach” to a DOM element. This functionality includes things such as painting and tokens.

Once defined, classes following the inheritance model can be exposed as jQuery plugins using the $.plugin() helper ([based on the concept discussed in this post](http://alexsexton.com/?p=51)). For example, to expose the custom class DJ.UI.MyNewComponent as a jQuery plugin named “dj\_MyNewComponent”, the following call would be made:   
 *$.plugin(‘dj\_MyNewComponent’, DJ.UI.MyNewComponent);*

The custom Dow Jones MVC View Component item templates include working templates in which you can add the custom logic for your component as well as override any of the functionality derived from the base class.