DEVELOPERS GUIDE

JDASH FOR ASP.NET MVC

Kalitte Inc.

www.jdash.net

Version 1.0

TABLE OF CONTENTS

GETTING STARTED

PRODUCT OVERVIEW

JDash Asp.Net MVC is an Asp.Net MVC and JavaScript library which allows you to integrate end user designed dashboards into your application.

Simply, as a developer you develop MVC controller, views and models for a JDash dashlet module. JDash automatically converts your module to drag-drop dashlets which in turn your users use dashlets to design their dashboards.

QUICK START

This article discusses how to create a "Hello World!" application using JDash. Source code can be found inside installation folder.

PREREQUESTS

JDash.Net should be installed.

STEPS

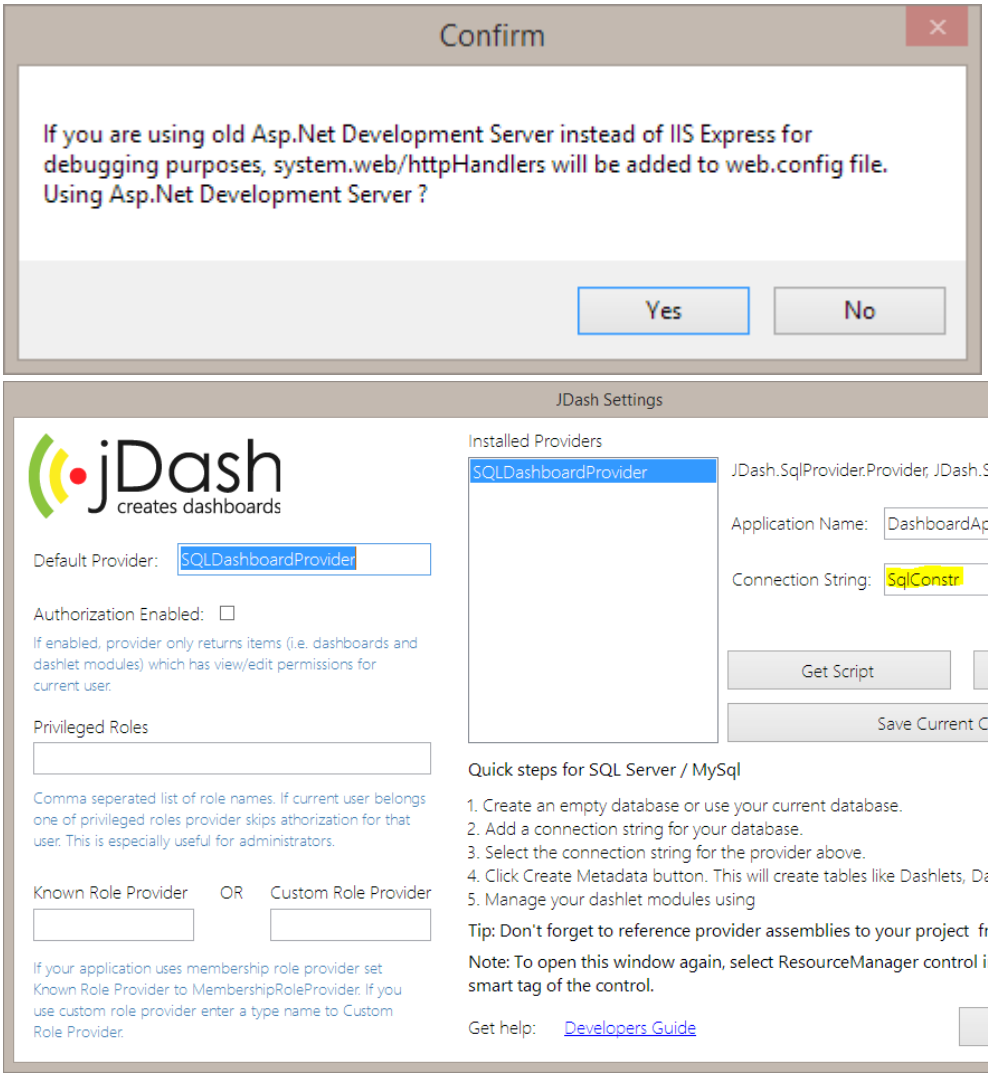
STEP 1 CREATE PROJECT AND REFERENCE REQUIRED ASSEMBLIES

1. Create a new empty Asp.Net MVC project.
   Using project references window (Inside solution explorer window, right click on *References* and click *Add Reference* menu item) add reference to *JDash.dll, JDash.Mvc.dll* and *JDash.SqlProvider.dll*. All assemblies can be found inside installation directory (by default *C:\Program Files (x86)\Kalitte\JDash Asp.Net MVC\packages\JDash.Mvc.Reference*).
2. Rebuild your project.

STEP 2 CONFIGURE SQL SERVER PROVIDER

Built-in provider *SQLDashboardProvider* uses Microsoft SQL Server to retrieve and persist metadata. In this tutorial we will use *SQLDashboardProvider*.

1. Using SQL Server Management Studio create new SQL database.
2. Return to Visual Studio and add a connection string named *SqlConstr* to *web.config* for the database you setup on previous step. Ensure connection string has the necessary authentication information for connecting to the database.
3. Using JDash MVC menu inside Visual Studio open Settings Dialog.
   Probably you will use local IIS Express server to run your project but if you want to use Asp.Net Development Server click *Yes* to add http handlers inside *system.web* section.



4. Select connection string named *SqlConstr* and click Create Metadata button. This will create tables inside your database. Click OK. *Web.config* file will be updated automatically based on your settings.

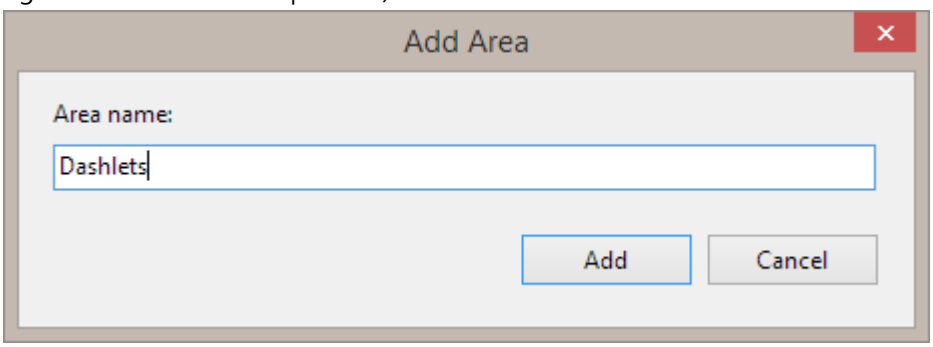
STEP 3 DEFINE DASHLET

1. Using JDASH MVC menu open Management Portal.
2. Click create new module link on top of page.

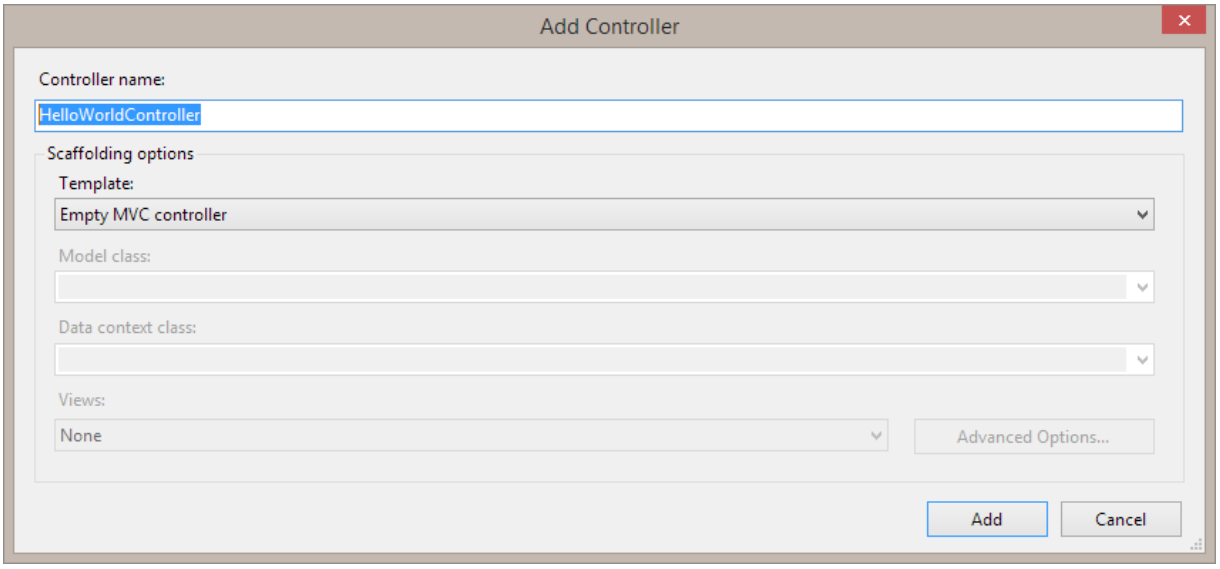
3. Set title as “Hello World!” and controller as */Dashlets/HelloWorld*

STEP 4 IMPLEMENT DASHLET

1. Add a new area to your project named Dashlets. (Select your project inside solution explorer, right click and click Add | Area ...)

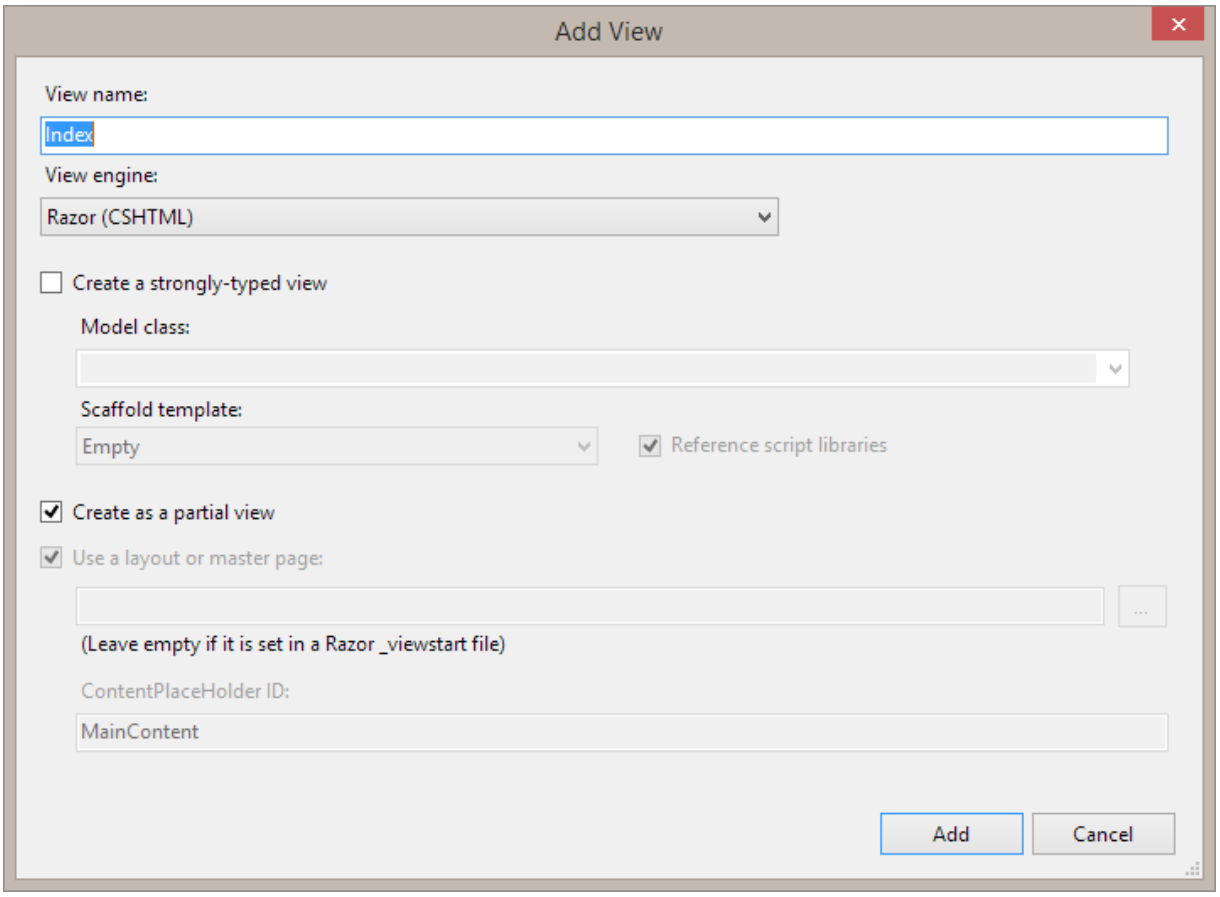


2. Add a new controller named *HelloWorldController* inside Areas -> Dashlets -> Controllers.



3. Create a directory named *HelloWorld* inside Areas -> Dashlets -> Views. (Select Areas -> Dashlets -> Views inside solution explorer, right click and select New Folder.

4. Select *HelloWorld* folder, right click and select Add | View ... menu.



5. Implement your dashlet view as below.

```
<div>
    <h1>Hello World!</h1>
</div>
```

STEP 5 IMPLEMENT APPLICATION

1. Add a new controller named *HomeController* to your application. (Do not use *Controllers* directory inside Areas -> Dashlets folder, use root directory relative your project)
2. Implement your controller as below.

```
using System.Threading;
using JDash.Models;
using JDash;

...
public class HomeController : Controller
{
  public ActionResult Index()
  {
    var user = Thread.CurrentPrincipal.Identity.Name;

    // Try to get a dashboard, if not create a default dashboard
    var dashboard = JDashManager.Provider.GetDashboardsOfUser(user).FirstOrDefault();
    if (dashboard == null)
    {
        dashboard = new DashboardModel()
        {
            title = "My Dashboard",
        };
        dashboard.metaData.created = DateTime.Now;
        dashboard.metaData.createdBy = user;
        JDashManager.Provider.CreateDashboard(dashboard);
    }

    // Get a list of dashlet modules
    ViewBag.DashletModules = JDashManager.Provider.SearchDashletModules().data;
    ViewBag.CurrentDashboard = dashboard.id;

    return View();
 }
}
```

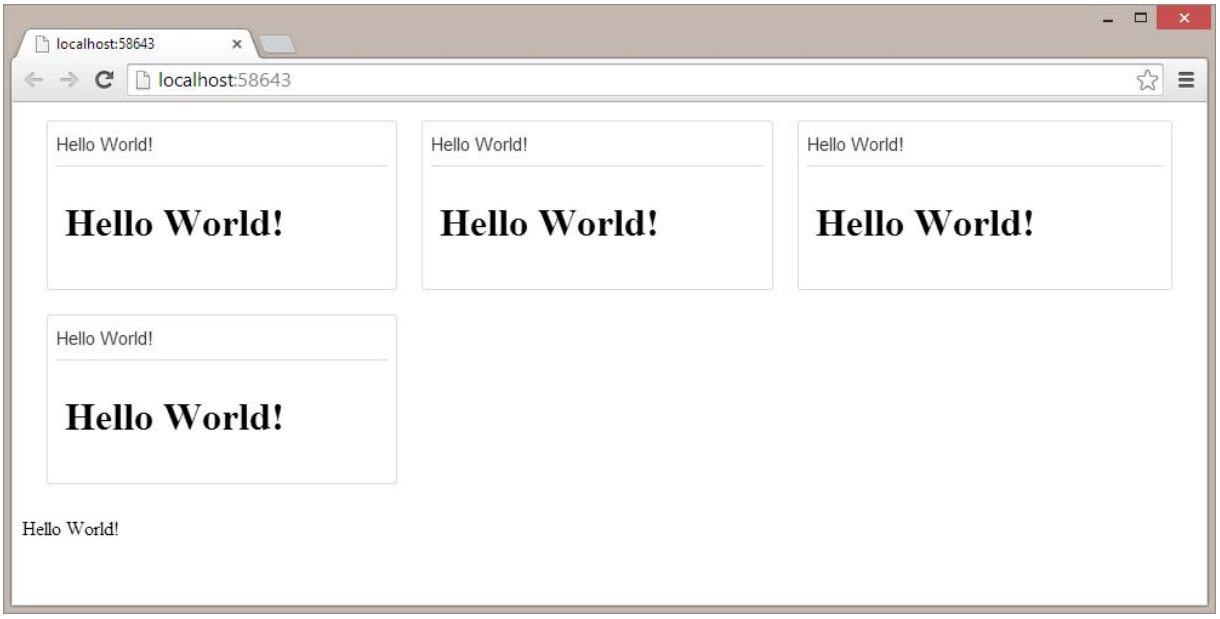
3. Add a new folder named *Home* inside project root -> *Views* folder.
4. Add a new view inside *Home* folder named *Index*. Implement your view as below.

```
@using JDash.Mvc
@{
    ViewBag.Title = "JDash App";


    foreach (var module in ViewBag.DashletModules)
    {
         using (Html.JDash().DashletModulesContainer().Tag("ul").Content())
         {
             @(Html.JDash().DashletCreateLink()
                .Tag("div")
                .Module(module.id)
                .InnerText(module.title)
                .DashboardView("myDashboard")
                .Behaviour(DashletCreateBehaviour.Both)
                .Render());
         }
    }

    @(Html.JDash().DashboardView()
           .ID("myDashboard")
           .DesignMode(DashboardDesignMode.full)
           .Load(ViewBag.CurrentDashboard)
           .Render());


    @(Html.JDash().ResourceManager()
        .Theme("flat")
        .Style("w")
        .CookieForTheme(true)
        .ClientInitHandler("window.runApp && window.runApp();")
        .Render())
}
```

5. Run your application. Try to drag Hello World inside dashboard.

Hello World!

This will create a new dashlet inside dashboard. You will probably get Http 500 error after adding new dashlet. Locate *web.config* file inside Areas -> Dashlets -> Views folder. Remove reference to *System.Web.Optimization* inside *system.web.webPages.razor* section or install this assembly using *Nuget*.



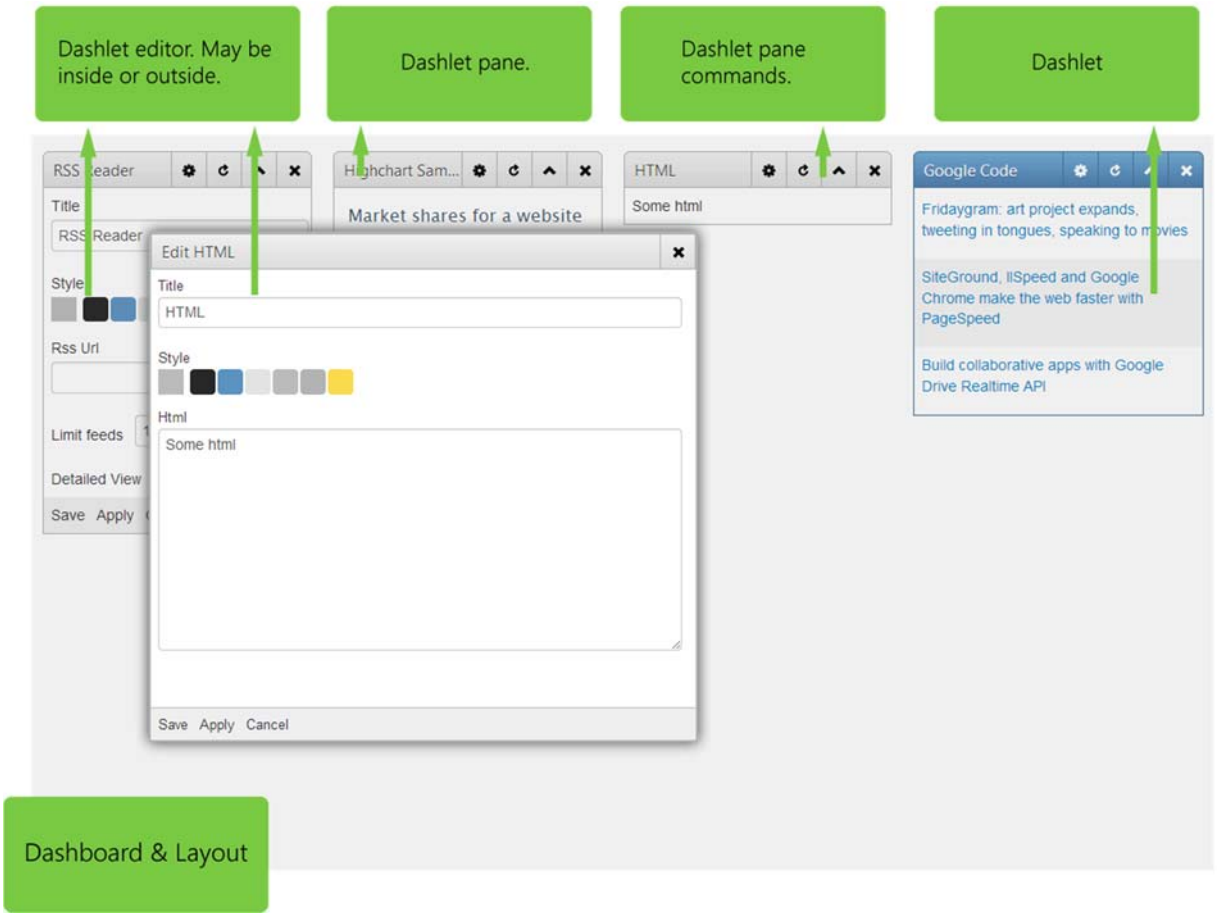
BASIC CONCEPTS AND ARCHITECTURE

BASIC CONCEPTS

Below table summarizes basic concepts of JDash world.

Concept	Description
Dashboard	Dashboard contains one Layout and optional dashlets. Dashboard is responsible to manage lifecycle of layout and dashlets inside layout.
Layout	Layout is the main container for dashlets and is located inside a dashboard. Layout is responsible positioning and drag-drop of dashlets.
Dashlet Module	Dashlet module defines a type which includes controller path, title, description and optional initialization data for a dashlet.
Dashlet	Dashlet is an instance of dashlet module. It inherits properties and behavior from dashlet module.
Dashlet Editor	Dashlet editor is the control which provides a user interface for configuration of a dashlet by end user.
Metadata	Metadata contains data structures for dashboards, dashlet modules and dashlets.
Provider	Provider is responsible to provide metadata to JDash.Net framework. JDash.Net is database independent and it is provider's responsibility to store and retrieve metadata.

Below image shows a loaded dashboard and some of its parts.



ASSEMBLY INFORMATION

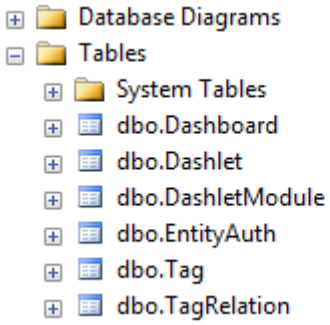
There exists three key assemblies you generally need to reference during development.

Assembly	Description
JDash.dll	Contains core data types and classes. All metadata (i.e. *DashboardModel*, *DashletModel*, etc.), *JDashProvider* and *JDashManager* classes are located inside this assembly.
JDash.Mvc.dll	Contains core structures and helper classes for Asp.Net MVC.
JDash.SqlProvider.dll	Contains JDash provider for Microsoft SQL database.
JDash.MySqlProvider.dll	Contains JDash provider for MySQL database.

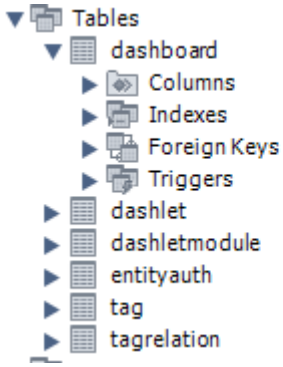
JDASH SETTINGS AND PROVIDERS

JDash is database independent and uses providers to get and store metadata like dashboard definitions, dashlet positions and configuration. There exists two built-in providers for MS SQL Server and MySQL Server. You can also develop your own provider for other databases by inheriting from *JDash.Provider.JDashProvider* abstract class.

As an example below screen shot shows how *SQLDashboardProvider* stores metadata using tables.



And similarly below is for *MySqlProvider*.



Abstract class *JDash.Provider.JDashProvider* defines properties and methods to manage metadata. At runtime, *web.config* settings section JDash governs the settings of providers.

Below is a sample JDash section inside *web.config* file.

```
<configuration>
  <configSections>
    <section type="JDash.Configuration.DashboardSettingsSection,JDash" name="JDash" />
  </configSections>

  <JDash defaultProvider="SQLDashboardProvider">
    <providers>
      <clear />
        <add applicationName="DashboardApp" connectionString="SqlConstr"
       name="SQLDashboardProvider" type="JDash.SqlProvider.Provider, JDash.SqlProvider" />
    </providers>
  </JDash>

  <system.webServer>
    <handlers>
      <add name="JDashNlsHandler" path="nls/*.js" verb="GET"
                type="JDash.Mvc.Core.NlsHandler,JDash.Mvc" />
    </handlers>
  </system.webServer>
```

Above section defines *SQLDashboardProvider* as a valid JDash provider and sets this provider as default. You can use *JDashManager.Provider* static property to get a reference to the active provider.

Generally speaking, key concepts for a provider is the connection string and metadata. Providers use connection string to connect to database. Metadata is the tables or stored procedures which allows provider to retrieve and persists metadata.

To create metadata programmatically you can use *JDash.Provider.JDashProvider.CreateMetadata method*.

```
public abstract bool CreateMetadata(string connectionString, bool checkMetadata = false);
```

First parameter is the connection string. If you set *true* for optional second parameter, provider checks if metadata is already created and does not try to create metadata.

```
JDashManager.Provider.CreateMetadata("Data Source=your data source ...", true);
```

Return value indicates if metadata is created. Providers are expected to return *false* if second parameter is true and metadata already exists.

You can also use *autoCreateMetadata* configuration property to create metadata automatically during initialization of provider.

```
<JDash defaultProvider="SQLDashboardProvider" autoCreateMetadata="true"
apiPath="jdash/api/">
    <providers>
      <clear /> …
```

A third method is creating metadata manually. To get SQL Script you can use *GetScript* method of provider.

```
public abstract string GetScript();
```

Note Before creating metadata a new database should be created or an existing database should be used.

JDASH JAVASCRIPT CLIENT ENGINE

JDash mainly includes three layers.

- ✓ Fully featured, HTML5 + CSS3 based User Interface JavaScript library
- ✓ JavaScript Client Provider
- ✓ Server Application (Asp.Net MVC, Database Providers for SQL Server and MySQL + custom providers)

Client side JavaScript user interface classes are called as client widgets. Client widgets are used to display a dashboard or render a theme, register a custom theme or create a dashlet. Every client widget should have an id value which uniquely identifies the widget. If you don't specify a value JDash client engine automatically assigns a unique value.

JavaScript singleton *jdash.ui.registry* is the main singleton to manage client widgets. For example you can use *jdash.ui.registry.byId* method to get a reference to a widget with an id.

Details of JDash client engine is out of scope of this document and JDash for Asp.Net MVC provides helper methods which isolates you from the complexity of JavaScript client engine.

HTML HELPERS

JDash for Asp.Net MVC provides various client widget wrappers. Wrappers allow developers to easily create JDash client widgets without using JavaScript.

Below table summarizes common client widget wrapper classes located inside *JDash.Mcv* namespace. From now on we will call client side widget wrappers as widgets.

Widget Class	Description

ResourceManager	Is responsible to manage resources i.e. client JavaScript, cascading style sheet (*.css*) and image files. Every JDash page should include this widget.
DashboardView	Use to display a dashboard.
ThemeChangeLink	Renders an anchor element to change theme of JDash.
ThemeStylesList	Renders a control which allows your users to change style of current theme of JDash.
DashletTitleEditor	Renders an input control so that user can change title of dashlet.
DashletCssEditor	Renders an input control so that user can set cascading style sheet class of a dashlet.
DashletStylesList	Renders a control which allows your users to change style of a dashlet.
DashletModulesContainer	Renders a container which allows users to add new dashlets using drag-drop.
DashletCreateLink	Renders an anchor element or custom content to create a dashlet inside a dashboard.

To create a widget inside a view use *Html.JDash()* methods. This extension returns a new *JDash.Mvc.JDashBuilder* instance which you can use its methods to create widgets.

Below code demonstrates sample usage of *JDashBuilder* for a Razor View.

```
@using JDash.Mvc

@(Html.JDash().ResourceManager()
        .Theme("flat")
        .Style("w")
        .Render())

@(Html.JDash().DashboardView()
        .ID("myDashboard")
        .Load(ViewBag.dashboardId)
        .Render())
```

Chaining allows you to call various methods to change visual and behavioral properties.

Tip To avoid repeating to declare *JDash.Mvc* namespace inside your views, you can use *system.web/pages/namespaces* section of *web.config* file to define this namespace as a global namespace.

```
<system.web>
  <compilation debug="true" targetFramework="4.0" />
  <authentication mode="Forms">
    <forms loginUrl="~/Account/Login" timeout="2880" />
  </authentication>
  <pages>
    <namespaces>
      <add namespace="System.Web.Routing" />
      <add namespace="System.Web.WebPages" />
      <add namespace="JDash.Mvc"/>
    </namespaces>
```

Every widget inherits from *JDash.Mvc.DomElement* and has the following common methods.

Method	Description
Tag	Specifies rendering tag i.e. div, a, etc.
ID	Specifies id of the dom/widget element. This also becomes the id of client widget and dom node.
InnerText	Specifies *innerText* property for dom element.
InnerHtml	Specifies *innerHTML* property for dom element.
Css	Sets a *css* class value for dom element.
Attr	Use to add additional attributes for the dom element.
Render	Renders the configured dom element on to the page.
Content	Use to set additional content inside dom element.

As an example, below code creates an instance of *ThemeStylesList* widget which allows your users to change style of the current theme JDash uses.

```
@Html.JDash().ThemeStylesList()
```

Creating an instance is not enough to render the widget. After creating the instance *Render* or *Content* method should be called so that widget renders itself on to the page.

```
@(Html.JDash().ThemeStylesList().Render())
```

Content method is useful to set Html content for the widget.

```
@using (Html.JDash().DashletModulesContainer().Tag("ul").Css("slidee").Content())
{
    // Your html content
}
```

RESOURCEMANAGER BASICS

JDash.Mvc.ResourceManager is responsible for managing resources i.e. client JavaScript, cascading style sheet (*.css*) and image files. Every JDash view should include this widget.

When *JDashBuilder* instance is created by a call to *Html.JDash()* extension method, a *ResourceManager* instance is automatically created by the builder. JDash keeps this unique instance and does not create new instances.

Although *ResourceManager* instance is created automatically, developers should use *Html.JDash().ResourceManager()* method to configure and render resource manager. Best place for this initialization is the end of the view.

```
@Scripts.Render("~/bundles/jquery")
@RenderSection("scripts", required: false)

@(Html.JDash().ResourceManager()
                .Theme("flat")
                .Style("w")
                .ClientInitHandler("window.runApp && window.runApp();")
                .Render())
```

Preferably you can set a common layout (*i.e.*"*~/Views/Shared/_Layout.cshtml*) and put resource manager at end of this layout view. This ensures all views using this layout you will have the resource manager.

Below table summarizes common methods of *ResourceManager*.

Method	Description
Theme	Use to set a theme. You can use *Themes* property to get a list of built-in themes.
Style	Use to set style for the current theme. Styles allows changing main colors without changing the whole theme.

CookieForTheme	Use to set if JDash uses cookie to load previously set theme if exists. This is useful to automatically load the last selected theme.
ClientInitHandler	Use to execute a JavaScript code after JDash client engine initialization fully completed. JDash client engine initialization is completed when dom tree is ready and all resources are loaded successfully by the client engine.

WORKING WITH DASHBOARDS

BASIC CRUD OPERATIONS

CREATING DASHBOARDS

To create a dashboard use *JDashProvider.CreateDashboard* method.

```
var newDashboard = new DashboardModel()
{
   title = "Title for dashboard",
};
newDashboard.metaData.created = DateTime.Now;
newDashboard.metaData.createdBy = "me";
newDashboard.metaData.group = "Startup Dashboards";
newDashboard.share = ShareModel.shared;
JDashManager.Provider.CreateDashboard(newDashboard);
```

Above code will create a new dashboard. After a dashboard is created you can check *DashboardModel.id* property to get a unique value to identify your dashboard.

Generally you give your users the right to create dashboards and adding dashlets to their dashboards. *DashboardModel.metadata.createdBy* is a key property for this and specifies owner of the dashboard.

Another key property is the *DashboardModel.share*. A user can share his dashboard and this value is set to *ShareModel.shared*. By sharing his dashboard user grants right to others to view that dashboard.

SEARCHING FOR DASHBOARDS

If you know *id* of a dashboard simply use *JDashProvider.GetDashboard* method.

```
var dashboard = JDashManager.Provider.GetDashboard("12");
```

If you want to get a list of dashboards with some criteria use *JDashProvider.SearchDashboards* method and pass a *JDash.Query.Dynamic.DynamicQuery* instance.

DynamicQuery class allows you to define filtering, sorting and paging information. Below is a sample *DynamicQuery* instance which demonstrates filtering, paging and sorting.

```
var  query = new DynamicQuery() {
     filter = new FilterParam(),
     paging = new Paging(),
     sort = new List<Sort>() };

query.filter.op = FilterOperator.or;

query.filter.filters.Add(new Filter() {
    field = "metaData.group",
    value = "work group",
    op = CompareOperator.eq });

query.filter.filters.Add(new Filter() {
    field = "title",
    value = "Dash",
    op = CompareOperator.contains });

query.sort.Add(new Sort() {
    field = "viewOrder",
    op = global::JDash.Query.SortDirection.desc });

query.paging.skip = 0;
query.paging.take = 2;

// this query returns top two dashboards whose group is work group or title contains Dash
// and ordered by viewOrder.
var dashboards = JDashManager.Provider.SearchDashboards(query);
```

UPDATING AND DELETING EXISTING DASHBOARDS

Use *JDashProvider.SaveDashboard* and *JDashProvider.DeleteDashboard* methods to update and delete existing dashboards.

DISPLAYING AND DESIGNING A DASHBOARD

JDash.Mvc.DashboardView widget allows you to display a dashboard. To create an instance use *DashboardView* method of *JDashBuilder*.

```
@(Html.JDash().DashboardView()
        .ID("myDashboard")
        .Load("12")
        .Render())
```

Above code creates a new instance of *jdash.ui.DashboardView* JavaScript widget on the client side which has a widget id of "*myDashboard*" and loads the specified dashboard.

SETTING DESIGN MODE

By default a dashboard is displayed read only to the user. *JDash.Mvc.DashboardDesignMode* enumeration values can be set to make it designable.

Property value	Description
DashboardDesignMode.full	Dashboard layout can be changed and new dashlets can be added to dashboard.
DashboardDesignMode.dashboard	Only layout can be changed.
DashboardDesignMode.dashlets	Only new dashlets can be added.
DashboardDesignMode.none	Read only view.

```
@(Html.JDash().DashboardView()
                .ID("myDashboard")
                .DesignMode(DashboardDesignMode.full)
                .Load(ViewBag.dashboardId)
                .Render())
```

LOADING DASHBOARD

Use *JDash.Mvc.DashboardView.Load* method to load a dashboard with a specified id.

```
public DashboardView Load(object id,
                          string callback = null,
                          string errback = null,
                          string progback = null)
```

You can set an optional JavaScript handler which executes after dashboard has loaded successfully, a handler if an error occurs and a last handler to get the loading progress.

```
@(Html.JDash().DashboardView()
              .ID("myDashboard")
              .DesignMode(DashboardDesignMode.full)
              .Load(ViewBag.dashboardId,
                 "function() { alert('Loading completed');  }",
                 "function(err) { alert ('Loading failed'); }",
                 "function(prog) {  }")
              .Render())
```

WORKING WITH LAYOUTS

Every dashboard has a layout and layout is responsible to organize dashlets. JDash.Net supports grid and absolute[1] layout types and allows creating new layout classes[2].

A layout has sections and each section has zones. On client side, section and zone information is used to create actual dom nodes which dashlets will be placed in.

JDash.Models.LayoutModel class (inside *JDash.dll*) represents a layout.

If no layout is specified while creating a dashboard it defaults to grid layout.

Below code demonstrates creating a custom grid layout.

```
var newDashboard = new DashboardModel()
{
  title = "My Dashboard"
};

var sections = new Dictionary<string, SectionModel>();
var zones = new Dictionary<string, ZoneModel>();
zones.Add("zone1", new ZoneModel()
{
  cls = "zoneCssClass"
});
zones.Add("zone2", new ZoneModel());
zones.Add("zone3", new ZoneModel());
sections.Add("section1", new SectionModel() { zones = zones });
newDashboard.layout = new LayoutModel() { type = LayoutModel.Grid, sections = sections };
JDashManager.Provider.CreateDashboard(newDashboard);
```

Static property *LayoutModel.PredefinedLayouts* returns a list of commonly used predefined layouts.

```
public static Dictionary<string, LayoutModel> PredefinedLayouts
```

You can bind predefined layouts to a repeater and combine with the built-in css classes to display a list of predefined layouts to end users.

```
@foreach (var item in JDash.Models.LayoutModel.PredefinedLayouts)
{
    var css = "d-layout-predefined d-layout-predefined" + item.Key;
    <div class="@css"></div>
}
```

Result is

[1] Absolute layout should be considered at experimental stage.
[2] Currently only supported by JDash.Net client engine.

You can pass key of the selected predefined layout to helper method *LayoutModel.GetPredefinedGridLayout* to get a configured grid layout and use this value to create a dashboard.

WORKING WITH DASHLETS

DASHLET MODULES

A dashlet module is a template for a dashlet. Dashlet modules and dashlets can be considered like classes and objects respectively in object oriented programming paradigm.

JDash.Models.DashletModuleModel class (located in *JDash.dll* assembly) defines a dashlet module. Below table summarizes key properties of a *DashletModuleModel* class.

Property	Description
id	Unique identifier of the module.
title	Title of the module. This value becomes default value for dashlets of the module.
path	This is the client JavaScript path of the module.
config	Key value pairs which defines a configuration for the module.
paneConfig	Key value pairs which defines pane properties for the module i.e. commands, base css class etc.
dashletConfig	Key value pair for default dashlet configuration. Dashlets also have their own configuration. When a new dashlet is created this value is automatically copied to the dashlet configuration.
metaData	Metadata information like description, group, created and modified date of the item.

Generally you use JDash Management Portal to define your dashlet modules. It provides a user interface which allows you to define common properties.

Below example demonstrates how to create a dashlet module programmatically.

```
var newModule = new DashletModuleModel();
newModule.title = "My module";
newModule.path = "[MVCDefault]";

newModule.config.Add("mvcConfig", new
{
   controller = "/Dashlets/Html",
});

newModule.config.Add("editor", new {
   useWindow = true,
   paneConfig = new {
      cssClass = "editorClass",
      width= "300px",
      Height= "300px"
   }
});

newModule.dashletConfig.Add("html", " Hello world! Change me soon!");
newModule.paneConfig.Add("cssClass", "dashletClass");
newModule.paneConfig.Add("builtInCommands",
                new string[] { "restore", "maximize", "remove", "clone" });

JDashManager.Provider.CreateDashletModule(newModule);
```

JDash assumes there is no editor defined for your dashlet and does not create an editor command on header of pane if you don't specify an *editor* key for *config* property. This makes your dashlet non editable.

Use *JDashProvider.SearchDashletModules* method to get a list of modules. As can be expected, *JDashProvider.SaveDashletModule* and *JDashProvider.DeleteDashletModule* methods updates and deletes dashlet modules respectively.

CONTROLLER

mvcConfig.controller configuration value defines the server side controller which provides viewing, editing and saving a dashlet. Every dashlet should have a controller.

When a new dashlet is added by the user, JDash makes a request for the *Index* action of the controller with the unique id of the dashlet. This method is expected to return a view and this view is used to render dom for the dashlet. For the above example, */Dashlets/Html/Index* action is requested.

When a user tries to configure a dashlet, i.e. to set parameters, JDash makes a request for the *Editor* action of the controller with the unique id of the dashlet. This method is expected to return a view which will actually be used to create the editor dom of the dashlet.

When user wants to save settings, a request is made for the *Save* action of the controller with the unique id of the dashlet and optional parameters. If saving is successful Index action is requested and a new dom is created for the dashlet.

As a result, at least *Index* action should be implemented for the controller. If dashlet is editable/configurable by end user, *Editor* and *Save* methods should also be implemented.

CLIENT MODULES

Client modules are optional JavaScript objects which allows you to have much more control on dashlet lifecycle, client side events and dom manipulation. Without client modules dashlet controller can provide HTML content only. As an example, imagine you have a JavaScript based chart. When a new dashlet is created, you have to create the chart object and bind it to a dom element. When dashlet is dropped to a new position probably you have to redraw chart with new size and finally when dashlet is removed chart object should also be destroyed with the dashlet.

In next section more information will be provided about client modules.

DASHLET LIFECYCLE

When end user creates a dashlet JDash follows these steps.

1. A *context* object is created which will be a bridge between dashlet and JDash client engine.
2. *Index* action of the dashlet controller is requested. Index action is expected to return a view and therefore HTML string.
3. JDash tries to compile HTML string. Compiling means creating a dom node using the received HTML string. If jQuery is defined JDash tries to create dom node using jQuery library. If not, JDash uses an internal compiler. Created dom node is called view node.
4. Linking phase begins. Linking means taking created dom node and parsing it to create necessary widgets. This step is useful for creating JavaScript objects defined by dom node i.e. data- attributes. Note that both container node and view node is not appended to the dashboard in this step.
5. A container dom node is created and view node is pushed inside. Container node is available using *this.domNode* and view node is available using *this.viewNode* inside your dashlet. Container node is appended to the dashlet pane.
6. If dashlet has a client module and client module defines *initialize* method, it is called. This method is where your dashlet starts.
7. When dashlet is removed from dashboard, if dashlet has a client module and client module defines *destroy* method, this method is called.

Below steps are performed when a user tries to edit a dashlet.

1. A *context* object is created which will be a bridge between dashlet editor and JDash client engine. Note this context is different than the dashlet context object.

2. *Editor* action of the dashlet controller is requested. *Editor* action is expected to return a view and therefore HTML string.
3. JDash tries to compile HTML string.
4. Linking phase begins.
5. A container dom node is created and editor view node is pushed inside. Container node is appended to the dashlet pane.
6. If dashlet has a client module and client module defines *initialize* method, it is called. This method is where your dashlet starts.
7. If user tries to save dashlet and if dashlet has a client module and client module defines *validate* method, it is called.
8. When dashlet editor is removed from dom node, if dashlet has a client module and client module defines *destroy* method, this method is called.

ADDING DASHLETS TO DASHBOARD

Use *JDash.Mvc.DashletCreateLink* helper to render a dom element which will allow end user to add a dashlet to a dashboard. Consider the following.

```
@* Get a list of dashlet modules using provider *@

@foreach (var item in JDash.JDashManager.Provider.SearchDashletModules().data)
{
    @* For each dashlet module create an anchor element *@

    @(Html.JDash().DashletCreateLink()
                  .DashboardView("myDashboard")
                  .InnerText(item.title)
                  .Module(item.id)
                  .Render())
}

@(Html.JDash().DashboardView()
              .ID("myDashboard")
              .DesignMode(DashboardDesignMode.full)
              .Load(ViewBag.dashboardId)
              .Render())
```

This will render anchor elements which are bound to the dashboard named "*myDashboard*" and creates a new dashlet when clicked.

JDash also supports creating dashlets by using drag-drop. For this to work set behavior of *DashletCreateLink* to *Drag* and encapsulate it inside a *JDash.Mvc.DashletModulesContainer*.

```
@using (Html.JDash().DashletModulesContainer().Content())
{
    foreach (var item in JDash.JDashManager.Provider.SearchDashletModules().data)
    {
        var widget = Html.JDash().DashletCreateLink()
                                 .DashboardView("myDashboard")
                                 .Behaviour(DashletCreateBehaviour.Drag)
                                 .Module(item.id);

        using (widget.Content())
        {
            <span>@item.title</span>
        }
    }
}
```

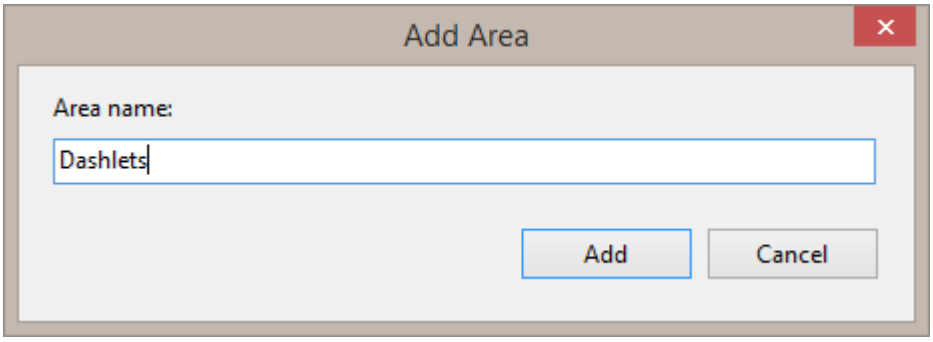
Below screen shot demonstrates adding a dashlet to a dashboard by dropping it on to the dashboard.

DEVELOPING DASHLETS

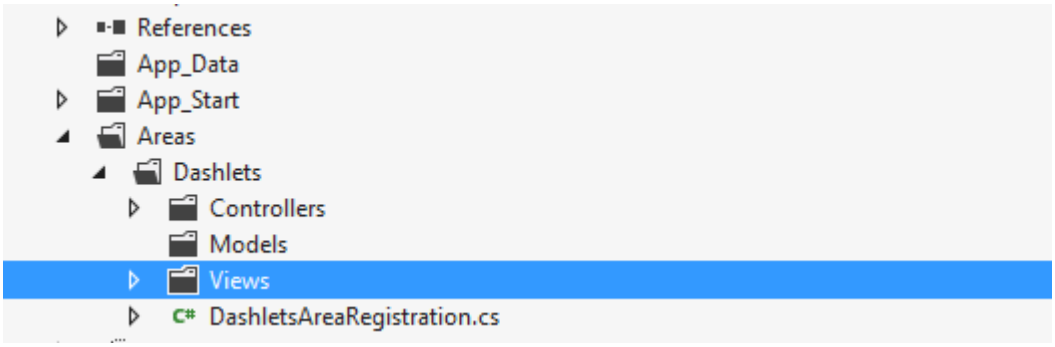
JDash for Asp.Net MVC simplifies dashlet development by using server side helpers and integrating into known methods Asp.Net MVC uses like data validation.

CREATING CONTROLLERS

As a best practice, we recommend creating an area for your dashlets. To create an area using Vs.Net, right click on your Asp.Net MVC project item in solution explorer window and use Add | Area command.



This will allow you to isolate dashlet controllers, views and models from your application logic.



Below can be used as a startup template for a dashlet controller.

```
public class HtmlController : Controller {

    public ActionResult Index(string id) {
            // Return view for the specified dashlet by id parameter
            return View();
    }

    public ActionResult Editor(string id) {
            // Return editor view for the specified dashlet by id parameter
            return View();
    }

    [HttpPost]
    public ActionResult Save(string id, FormCollection form) {
            // Save new configuration parameters for the specified dashlet by id parameter
            return new EmptyResult();
    }
}
```

FormCollection can be used as second parameter to retrieve form field values for *Save* action. Although this is possible and works we recommend creating a model.

CREATING MODELS

A model is a bridge between controller and view. There is no special requirement for a dashlet model.

Below code snippet demonstrates a sample model for Html Dashlet.

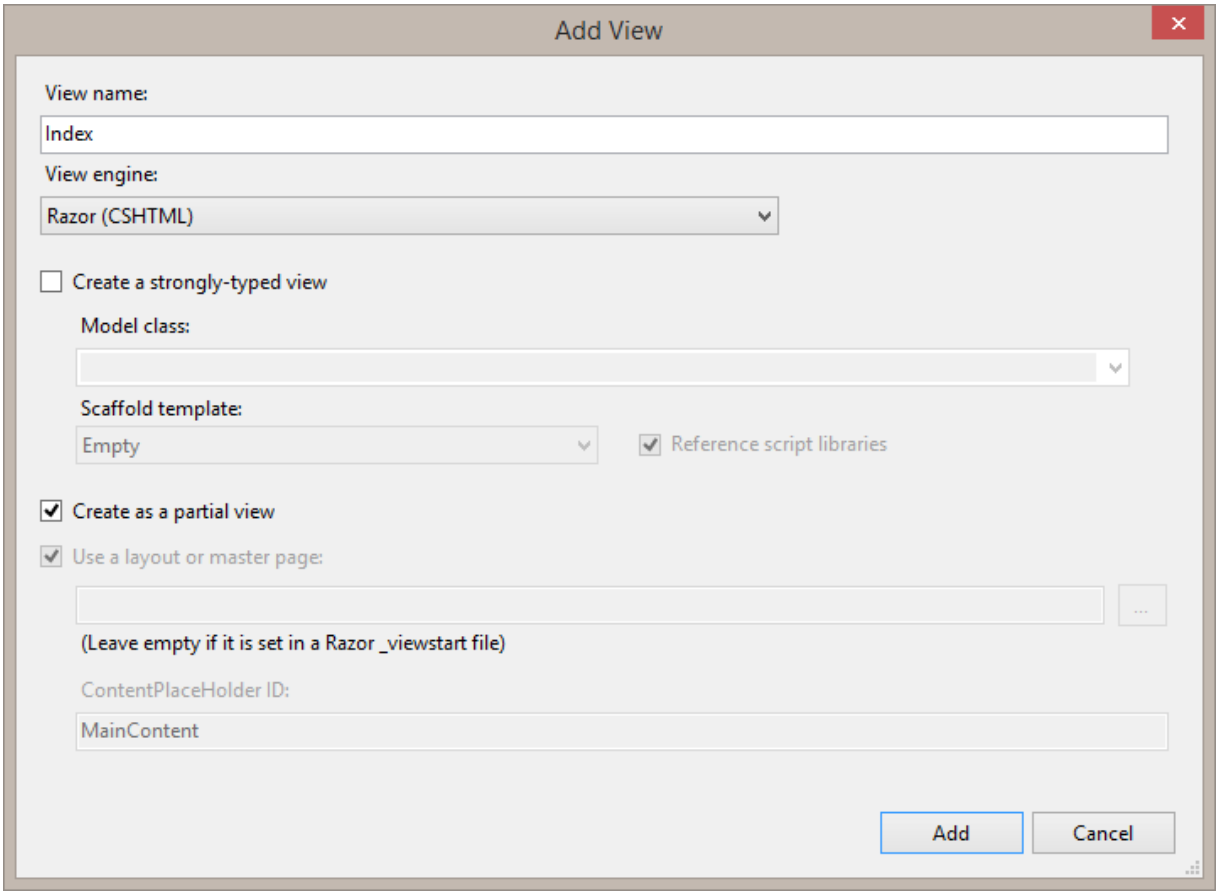
```
public class HtmlModel
{
    [Required]
    [Display(Name = "Html")]
    public string Html { get; set; }
}
```

Data annotations would be very useful for data validation and naming your controls. If a model is available, *Save* method of your controller can be changed as below.

```
public ActionResult Save(string id, HtmlModel model) {
    // Save new configuration parameters for the specified dashlet by id parameter
    return new EmptyResult();
}
```

CREATING VIEWS

Views provides HTML string which in turns JDash converts to dom elements. One of the power of the views is that you can use models so that ready to use Html is sent to client.



Below is a sample Razor view for Html dashlet.

```
<div>
    @Html.Raw(Model.Html)
</div>
```

JDash requires and expects views to return HTML string which can be used to create dom nodes. So, considering this requirement, model data is used inside div elements for the above view.

Below is a sample view using ASPX rendering engine.

```
<%@ Control Language="C#" Inherits="System.Web.Mvc.ViewUserControl<dynamic>" %>

<div>
    <%=Html.Raw(Model.Html) %>
</div>
```

Index method of controller can be implemented as below so that model is passed to the view.

```
public ActionResult Index(string id) {
  // get dashlet using provider
  var dashlet = JDashManager.Provider.GetDashlet(id);

  // get configuration value named html from dashlet.
  var savedHtml = dashlet.config.Get<string>("html", "");

  // create model
  var model = new HtmlModel() { Html = savedHtml };

  //pass model to view
  return View(model);
}
```

EDITOR VIEWS

Same rules valid for dashlet views also applies to editor views like the returned Html string should be parsable to dom nodes.

Although you are not required to use a form element inside an editor view it is strongly recommended to add an Ajax form. JDash client engine will recognize Ajax form and automatically setup it to use with *Save* method of the controller.

Below is a sample editor.

```
@using JDash.Mvc
@model Models.HtmlModel

<div>
    @using (Ajax.BeginForm(new AjaxOptions() { }))
    {
        @Html.AntiForgeryToken()
        @Html.ValidationSummary(true)

        <fieldset>
            @Html.TextAreaFor(m => m.Html, new {  rows = "10" })
            @Html.ValidationMessageFor(m => m.Html)
        </fieldset>
    }
</div>
```

While parsing form element, JDash client engine looks for *data-ajax* attribute for form element. If this attribute is found then form element is configured to post to dashlet controller action *Save*. Also client engine makes an asynchronous request instead of regular form post.

To pass model to the editor view *Editor* action can be implemented as below similar to *Index* action.

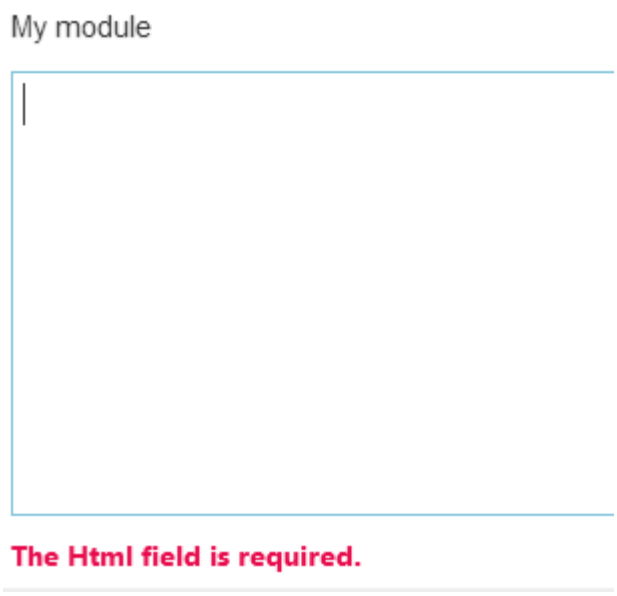
```
public ActionResult Editor(string id) {
  // get dashlet using provider
  var dashlet = JDashManager.Provider.GetDashlet(id);

  // get configuration value named html from dashlet.
  var savedHtml = dashlet.config.Get<string>("html", "");

  // create model
  var model = new HtmlModel() { Html = savedHtml };

  //pass model to view
  return View(model);
}
```

An important requirement for dashlet editor is the data validation. Asp.Net MVC provides several built-in methods for data validation and jQuery validation plugin + Unobtrusive validation support is one of them.

If you use unobtrusive data validation with jQuery data validation plugin, JDash integrates itself into the validation process and if form validation fails JDash cancels saving settings.

To add unobtrusive data validation support you can use the following helper on your razor page or alternatively you can put this on a layout view assuming bundles are configured correctly.

```
@Scripts.Render("~/bundles/jqueryval")
```



SAVING SETTINGS

Saving settings means persisting configuration data for dashlet. Save method can be implemented as below.

```
[HttpPost]
public ActionResult Save(string id, HtmlModel model)
{
    var dashlet = JDashManager.Provider.GetDashlet(id);
    dashlet.config["html"] = model.Html;
    JDashManager.Provider.SaveDashlet(dashlet);
    return new EmptyResult();
}
```

When user tries to save settings, JDash client engine automatically creates a post request which includes dashlet id and form fields. Asp.Net MVC locates the Save action and creates model. Remaining part is persisting settings as the above code.

After a successful saving operation *Index* action of the controller is requested and dom node of dashlet is re-created.

WORKING WITH CLIENT MODULES

Client modules allow you to develop dashlets with JavaScript and also use benefits and power of Asp.Net MVC controllers, views and models.

JDash supports client modules for both dashlets and dashlet editors and they are both optional.

```
var newModule = new DashletModuleModel();

newModule.title = "My Chart Dashlet";

// instead of [MVCDefault]
newModule.path = "dashlets/chart/index";

newModule.config.Add("mvcConfig", new
{
   controller = "/Dashlets/Chart",
});

newModule.config.Add("editor", new
{
    // instead of [MVCDefault]
    path= "dashlets/chart/editor"
});
newModule.paneConfig.Add("builtInCommands",
                            new string[] {"restore", "maximize", "remove", "clone" });

JDashManager.Provider.CreateDashletModule(newModule);
```

Above code snippet creates a dashlet module and sets client modules for both dashlet and editor. When a new dashlet is created, JDash retrieves the HTML from dashlet controller using *Index* action and executes client module which is located in */scripts/dashlets/chart/index.js*.

By default JDash uses */scripts* as root directory for client packages. You can use *ResourceManager.Package* method to change this behavior. For example;

```
@(Html.JDash().ResourceManager()
               .Theme("flat")
               .Style("w")
               .Package("dashlets", "/MyNewLocation")
               .Render())
```

Above code snippet will change root path of package named *dashlets* as */MyNewLocation* and JDash will search for */MyNewLocation/dashlets/chart/index.js instead of /scripts/dashlets/chart/index.js*.

Note Although JavaScript file has *.js* extension please note you don't specify extension when defining your client module.

Note Controller is not optional for dashlets. Even you use client modules you have to implement controller of your dashlet.

AMD FORMAT

JDash uses AMD (Asynchronous Module Definition) module format for client modules. This allows JDash client engine to load client modules when they are required and significantly reduces page load time.

Below paragraphs include basic information about AMD module format and some parts are inspired from documentation of *RequireJs*. If you are new to AMD module format or JavaScript just try to have an idea about the problem and solution.

The AMD API specifies a mechanism for defining modules such that the module and its dependencies can be asynchronously loaded. This is particularly well suited for the browser environment where synchronous loading of modules incurs performance, usability, debugging, and cross-domain access problems.

THE PROBLEM AND SOLUTION

The overall goal for the format is to provide a solution for modular JavaScript that developers can use today. It was born out of Dojo's real world experience using XHR+eval and proponents of this format wanted to avoid any future solutions suffering from the weaknesses of those in the past.

The AMD module format itself is a proposal for defining modules where both the module and dependencies can be asynchronously loaded. It has a number of distinct advantages including being both asynchronous and highly flexible by nature which removes the tight coupling one might commonly find between code and module identity. Many developers enjoy using it and one could consider it a reliable stepping stone towards the module system proposed for ES Harmony.

Today it's embraced by projects including Dojo (1.x), MooTools (2.0), Firebug (1.8) and even jQuery (1.7+).

The specification defines a single function "define" that is available as a free variable or a global variable. The signature of the function:

```
define(id?, dependencies?, factory);
```

id

The first argument, id, is a string literal. It specifies the id of the module being defined. This argument is optional, and if it is not present, the module id should default to the id of the module that the loader was requesting for the given response script. When present, the module id MUST be a "top-level" or absolute id

dependencies

The second argument, dependencies, is an array literal of the module ids that are dependencies required by the module that is being defined. The dependencies must be resolved prior to the

execution of the module factory function, and the resolved values should be passed as arguments to the factory function with argument positions corresponding to indexes in the dependencies array.

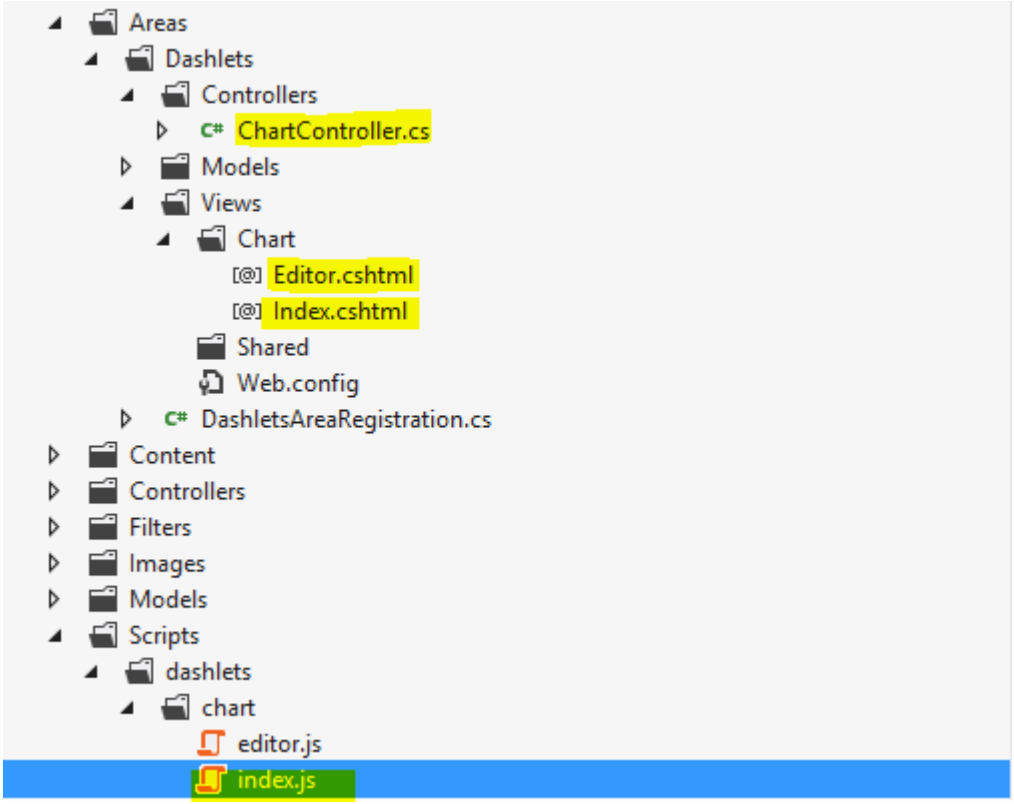
The dependencies ids may be relative ids, and should be resolved relative to the module being defined. In other words, relative ids are resolved relative to the module's id, and not the path used to find the module's id.

factory

The third argument, factory, is a function that should be executed to instantiate the module or an object. If the factory is a function it should only be executed once. If the factory argument is an object, that object should be assigned as the exported value of the module.

DEFINING CLIENT MODULES

Assuming the below configuration:



Sample Controller:

```
public class ChartController : Controller
{
  public ActionResult Index(string id)
  {
      var dashlet = JDashManager.Provider.GetDashlet(id);
      return View();
  }

  public ActionResult Editor(string id)
  {
       var dashlet = JDashManager.Provider.GetDashlet(id);
       return View();
  }

  [HttpPost]
  public ActionResult Save(string id, FormCollection model)
  {
       var dashlet = JDashManager.Provider.GetDashlet(id);
       JDashManager.Provider.SaveDashlet(dashlet);
       return new EmptyResult();
  }
}
```

Index View

```
<div>
    <h2>Chart Title</h2>
    <canvas class="chart"></canvas>
</div>
```

You can define your client module (/scripts/dashlets/chart/index.js) like the below example.

```
define({
    initialize: function (context, viewNode, reason) {
        // context also available as this.$context()
        // viewNode also available as this.viewNode
        console.log(context.model.title + " initialized with id " +
                    context.model.id + " because of " + reason);
    },

    destroy: function() {
        console.log(this.$context().model.title + " destroyed.")
    }
})
```

There exists two important methods for client modules. When dom for your dashlet is ready JDash client engine automatically calls *initialize* method of your dashlet with *context*, *dom node* and *reason* parameters.

context is the bridge between your dashlet and JDash client engine and provides many useful methods which you can use. *viewNode* is created using *Index* action of the dashlet controller. *reason* parameter specifies the reason for initialization. This parameter is useful to get if dashlet is re-initializing after a successful edit.

Alternatively you can define the same dashlet as below.

```
define(function () {

    // Best place for one time initialization of module
    // since this code executes for the first dashlet instance

    console.log("Module initialized.");

    var instanceCount = 0;

    return {
        initialize: function (context, viewNode, reason) {
            console.log("Now dashlet count is " + (++instanceCount).toString());
        },

        destroy: function() {
            console.log("Now dashlet count is " + (--instanceCount).toString());
        }
    }
})
```

Above definition is useful if you need to make one time initialization among instances. This is also useful if your dashlet has other JavaScript dependencies. Assuming you have another AMD module inside file /scripts/dashlets/charts/templates.js which has built-in templates.

```
define({
    line: {
        // properties
        title: 'Line Chart'
    },

    bar: {
        title: 'Bar Chart'
    },

    pie: {
        title: 'Pie Chart'
    }
})
```

And chart library inside /scripts/lib/chart.min.js, you can use below code snippet to load them dynamically when first instance of your dashlet is created.

```
define(["./templates", "/scripts/lib/chart.min.js"], function (templates) {
    return {
        initialize: function (context, viewNode, reason) {
            console.log('templates.line.title is ' + templates.line.title);
        }
    }
})
```

Note If a JavaScript file is AMD formatted extension should not be specified. For the above example note templates.js is AMD formatted but chart library is not.

Same rules apply for dashlet editor client modules.

BASE METHODS AND PROPERTIES

If you define a client module your client module automatically inherits from *jdash.mvc.DashletBase* class. This is the base class for both dashlets and editors and provides useful methods and properties.

Below table summarizes methods and properties provided by *DashletBase* class. Inside your dashlet / editor client module these methods/properties are available i.e. *this.$json*.

Method/Property	Description
$json	Helper for parsing and creating JSON documents and objects. Use *this.$json.parse* to parse a JSON document, *this.$json.stringify* to convert an object to JSON document. See http://dojotoolkit.org/reference-guide/1.9/dojo/json.html
$xhr	Is a provider that uses *XMLHttpRequest. See* http://dojotoolkit.org/reference-guide/1.9/dojo/request/xhr.html
$style	Helper for manipulating dom styles. See http://dojotoolkit.org/reference-guide/1.9/dojo/dom-style.html
$class	Helper for manipulating dom classes. See http://dojotoolkit.org/reference-guide/1.9/dojo/dom-class.html
$query(selector)	Returns a list of DOM nodes based on a CSS selector. This helper dashlet dom only to search for, not whole document. See http://dojotoolkit.org/reference-guide/1.9/dojo/query.html

$get(selector)	Returns first node based on a CSS selector.
$handlerError(err)	Automatically called if an error occurs. Base class displays an alert dialog by default.
resize	If you define a function named *resize*, JDash client engine will automatically call it when your dashlet needs resizing.
refresh	If you define a function named *refresh*, JDash client engine will automatically call it when your dashlet is refreshed.

WORKING WITH DASHLETCONTEXT

When a new dashlet is created JDash client engine creates a context object which is available by *this.$context()* method. Context provides various methods which can be used to get a reference to model object, to get size of dashlet or to broadcast events.

Below table summarizes some common methods.

Method/Property	Description
dashboard	Reference to *jdash.ui.DashboardView* object in which dashlet is hosted.
model	Reference to the *jdash.model.DashletModel* object which contains definition of dashlet.
pane	Reference to the *jdash.ui.DashletPane* object in which dashlet is located.
paneConfig	Reference to the *jdash.model.ConfigModel* object which holds pane configuration values.
config	Reference to the *jdash.model.ConfigModel* object which holds dashlet configuration values.
publish(topic, event)	Publishes a topic.
subscribe(topic,handler)	Uses *jdash.bus* function to subscribe to a topic. When context is destroyed (user removed the dashlet or dashboard unloaded) automatically removes the subscription.
save	Saves dashlet.
remove	Removes dashlet.

openEditor	If dashlet is editable opens editor.
getDashletSize	Returns width and height of dashlet. Useful to resize contents of dashlet.

Below code demonstrates common usage scenarios of base methods and dashlet context.

```
define(["./templates", "/scripts/lib/chart.min.js"], function (templates) {
    return {
        setChartSize: function (canvas) {

            // get size of dashlet
            var size = this.$context().getDashletSize();

            // set a style for container node
            this.$style.set(this.domNode, "height", Math.max(size.h, 300) + "px");

            // also set new sizes for canvas
            canvas.width = size.w;
            canvas.height = Math.max(size.h, 300);
        },

        drawChart: function () {

            // use $get function to get a reference to canvas inside dashlet.
            var canvas = this.$get("canvas");

            this.setChartSize(canvas);

            //TODO: Draw chart using canvas
        },

        resize: function (event) {
            // redraw when dashlet needs resizing
            this.drawChart();
        },

        initialize: function (context, viewNode) {
            // start drawing.
            this.drawChart();
        }
    }
})
```

WORKING WITH EDITORS

Dashlet editors allow your dashlets to be configured by end users. For example, a user may want to configure a chart dashlet and change chart type and title of the dashlet via your dashlet editor.

IMPLEMENTING A MODEL

Although not required, defining a model for configuration data is recommended. Consider the following model class for a chart dashlet.

```
public class ChartModel
{
    public static IEnumerable<SelectListItem> ChartTypes
    {
        get
        {
            var dict = new List<SelectListItem>(5);
            dict.Add(new SelectListItem() { Value = "Line", Text = "Line Chart" });
            dict.Add(new SelectListItem() { Value = "Bar", Text = "Bar Chart" });
            dict.Add(new SelectListItem() { Value = "Pie", Text = "Pie Chart" });
            return dict;
        }
    }

    [Required]
    [Display(Name = "Chart Type")]
    public string ChartType { get; set; }

    [Required]
    [Display(Name = "Chart Title")]
    public string ChartTitle { get; set; }
}
```

Model allows end user to set a chart type and title. Model also provides a static method which lists supported chart types.

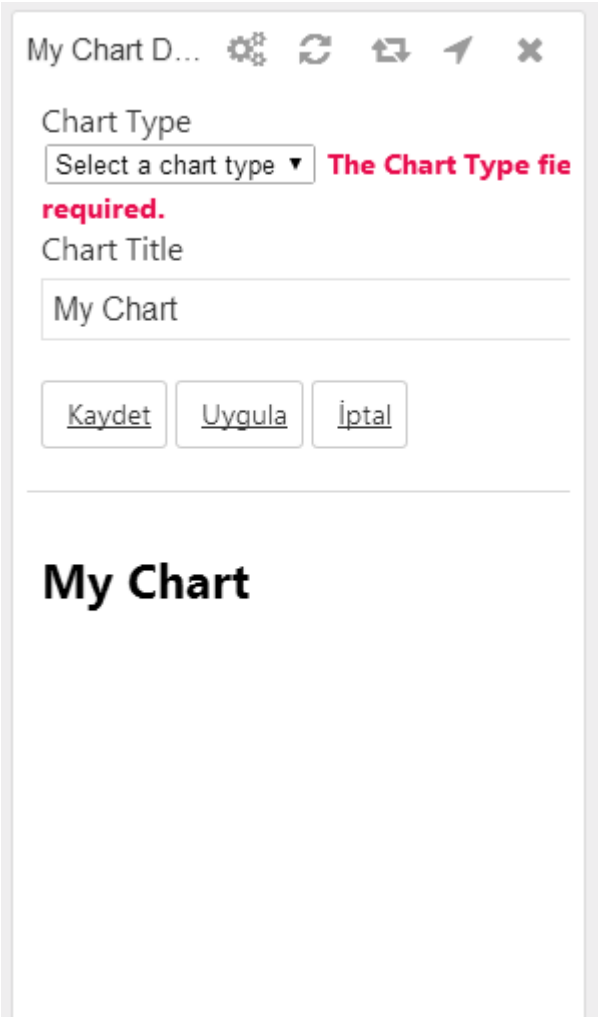
IMPLEMENTING VIEWS

Below is the updated view (i.e. *index.cshtml*) for dashlet.

```
<div>
    <h2>@Model.ChartTitle</h2>
    <canvas></canvas>
</div>
```

By using helpers editor can be implemented as below.

```
@using JDash.Mvc

@*replace with your namespace *@
@using Documentation.Samples.Areas.Dashlets.Models

@*replace with your model class*@
@model Documentation.Samples.Areas.Dashlets.Models.ChartModel


<div>
    @using (Ajax.BeginForm(new AjaxOptions() { }))
    {
        @Html.AntiForgeryToken()
        @Html.ValidationSummary(true)

        <fieldset>
            <legend>Set Chart Options</legend>
            <div class="form-group">
                @Html.LabelFor(m => m.ChartType)
                @Html.DropDownListFor(m => m.ChartType,
                                      ChartModel.ChartTypes, "Select a chart type")
                @Html.ValidationMessageFor(m => m.ChartType)
            </div>
            <div class="form-group">
                @Html.LabelFor(m => m.ChartTitle)
                @Html.TextBoxFor(m => m.ChartTitle)
                @Html.ValidationMessageFor(m => m.ChartTitle)
            </div>
        </fieldset>
    }
</div>
```



IMPLEMENTING CONTROLLERS

JDash.Models.DashletModel (located inside JDash.dll) allows you to set and persists configuration data using active provider.

Below is a sample implementation of *Save* action.

```
[HttpPost]
public ActionResult Save(string id, ChartModel model)
{
    // Get dashlet instance
    var dashlet = JDashManager.Provider.GetDashlet(id);

    // Set configuration value
    dashlet.config["model"] = model;

    // Save dashlet. Configuration is automatically serialized as JSON
    JDashManager.Provider.SaveDashlet(dashlet);

    // return updated config
    return new ConfigResult(dashlet.config);
}
```

Note that *Save* action returns *JDash.Mvc.ConfigResult*. JDash client engine understands this result type and updates the model on the client side automatically.

Below code listing shows updated *Index* and *Editor* Methods.

```
public ActionResult Index(string id)
{
    // Get dashlet instance
    var dashlet = JDashManager.Provider.GetDashlet(id);

    // get JSON serialized model from config. If there is model serialized
    // create an empty model instance
    var model = dashlet.config.JsonParsed<ChartModel>("model", new ChartModel());
    return View(model);
}

public ActionResult Editor(string id)
{
    // Get dashlet instance
    var dashlet = JDashManager.Provider.GetDashlet(id);

    // get JSON serialized model from config. If there is model serialized
    // create an empty model instance
    var model = dashlet.config.JsonParsed<ChartModel>("model", new ChartModel());
    return View(model);
}
```

CLIENT SIDE EDITORS

You can customize various behaviors if your dashlet uses an editor client module.

```
define({
    validate: function () {
        console.log('Client side validation in progress')
        // implement validation logic.
        // if validation fails return false
    },

    $submitForm: function () {
        console.log('Submitting form');
        return this.inherited(arguments);
    },

    $handleSubmitResult: function (event) {
        console.log('Processing return result');
        return this.inherited(arguments);
    }
})
```

If you define a validate function inside your client module, JDash calls it automatically for data validation. You can do client based data validation and return *false* if validation fails.

You can also override *$submitForm* and *$handleSubmitResult* functions. Note the following syntax.

```
this.inherited(arguments);
```

This will allow to call the base method, in this case methods of *DashletEditor* class which your editor is inheriting from.

CLIENT SIDE CONFIGURATION

You will probably want to get dashlet configuration on the client side so that you can change client side behavior according to the configuration.

DashletContext.model.config or simply *DashletContex.config* allows you to get a reference to the active configuration object.

```
drawChart: function () {
    var model = this.$context().config.get("model") || { ChartTitle:'',ChartType:'Line' };

    console.log(model);

    //TODO Draw chart based on model
}
```

COMMON TASKS

JDash for Asp.Net MVC provides helper classes to create client widgets which will isolate you from JavaScript. Client widgets are JavaScript objects which generally creates a dom element and simplify common tasks i.e. creating dashlets or changing a theme.

WORKING WITH THEMES

Themes allow you to change visual representation of dashboard and dashlets.

A theme is represented using *JDash.Mvc.ThemeInfo* class and has a name, *css* path and optional styles. Styles generally allow changing colors without loading a new cascading style sheet document.

BUILT-IN THEMES

JDash provides two built-in themes named *flat* and *default*. To get a list of available themes use *ResourceManager.Themes* property. Please note *ResourceManager.Themes* property returns a list of built-in themes only.

CUSTOM THEMES

JDash also supports custom themes which allows you to design your own theme.

To register a custom theme use *ResourceManager.RegisterTheme* method.

```
@{
    var myTheme = new ThemeInfo("MyTheme", "/Content/Theme/Custom/main.css");
    myTheme.Styles.Add("a", new ThemeStyleInfo("Black", "black"));
    myTheme.Styles.Add("b", new ThemeStyleInfo("Blue", "blue"));
}
@(Html.JDash().ResourceManager()
                .Theme("MyTheme")
                .Style("a")
                .RegisterTheme(myTheme)
                .Render())
```

Above code snippet registers a theme named *MyTheme*, which has its cascading style sheet location at /Content/Themes/Custom/main.css and has two styles.

Demo application contains *less* and *css* templates inside Content/Themes/ folder which you can use as a startup to design a new theme.

SETTING INITIAL THEME

To set initial theme and style use *ResourceManager.Theme* and *ResourceManager.Style* methods as above code snippet demonstrates.

CHANGING THEME AND STYLE

JDash.Mvc.ThemeChangeLink allows you to render a dom node which allows changing theme when clicked.

```
@foreach (var item in Html.JDash().ResourceManager().Themes)
{
     @Html.JDash().ThemeChangeLink().Theme(item.Name).InnerHtml(item.Name).Render()
}
```

To change style of current theme use *JDash.Mvc.ThemeStylesList* widget.

```
@(Html.JDash().ThemeStylesList().Render())
```

When you switch to a new theme (built-in or custom) JDash.Net client engine sets *j-theme-[themeId] j-style-[styleId]* classes for document body. This allows you to set custom styles for all document when user changed dashboard theme.

PERSISTING SELECTED THEME

ResourceManager.CookieForTheme method allows you to persist last selected theme and load it during initialization if exists.

```
@(Html.JDash().ResourceManager().CookieForTheme(true).Render())
```

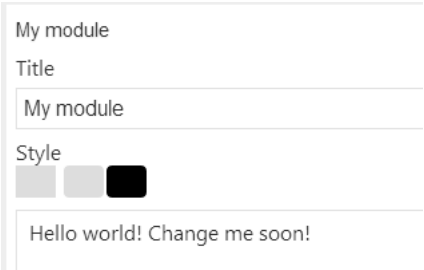
DASHLET TITLE AND STYLE EDITORS

JDash provides three helpers which you can use inside your dashlet editor.

Widget Class	Description
DashletTitleEditor	Renders an input control so that user can change title of dashlet.
DashletCssEditor	Renders an input control so that user can set cascading style sheet class of a dashlet.
DashletStylesList	Renders a control which allows your users to change style of a dashlet.

Below is a sample dashlet editor view.

```
@model Documentation.Samples.Areas.Dashlets.Models.HtmlModel
@using JDash.Mvc
<div>
    @using (Ajax.BeginForm(new AjaxOptions() { }))
    {
        <div>
            @Html.Label("Title")
            @Html.JDash().DashletTitleEditor().Render()
        </div>

        <div>
            @Html.Label("Style")
            @Html.JDash().DashletStylesList().Render()
        </div>

        @Html.AntiForgeryToken()
        @Html.ValidationSummary(true)

        <fieldset>
            @Html.TextAreaFor(m => m.Html, new { rows = "10" })
            @Html.ValidationMessageFor(m => m.Html)
        </fieldset>
    }
</div>
```



If you are using a form element inside your dashlet editor, JDash client engine automatically adds an hidden input named *DashletProperties* and uses this to post updated properties i.e. title or style.

Unfortunately, since posting is directly made to the Save action of your controller, a few changes should be done to your model and *Save* action to get updated values.

```
public class HtmlModel
{
    public string DashletProperties { get; set; }

    [Required]
    [Display(Name = "Html")]
    public string Html { get; set; }
}
```

By adding a new property named *DashletProperties* to your model, updated values will be available. Last change should be done to *Save* action of the controller as below.

```
public ActionResult Save(string id, HtmlModel model)
{
    var dashlet = JDashManager.Provider.GetDashlet(id);
    // Load updated properties
    dashlet.LoadProperties(model.DashletProperties);

    dashlet.config["html"] = model.Html;
    JDashManager.Provider.SaveDashlet(dashlet);
    return new EmptyResult();
}
```

WORKING ON THE CLIENT SIDE

Although JDash for Asp.Net MVC provides lots of helper classes to help you develop using Microsoft.Net sometimes it can be necessary to directly use JavaScript objects.

CLIENT SIDE INITIALIZATION

JDash.Mvc.ResourceManager is responsible to start and finalize client side initialization. Client side initialization means loading all external JavaScript and *css* files.

You can define a handler to execute JavaScript after all initialization is completed using *ResourceManager.ClientInitHandler* method.

```
@(Html.JDash().ResourceManager()
        .ClientInitHandler("window.runApp && window.runApp();")
        .Render())
```

After initializing completed successfully JDash automatically calls *runApp* function if exists. You can use this function as a startup for your application.

ABOUT CLIENT SIDE WIDGETS

JDash client engine uses widgets to create user interface on client side. Widgets are simply JavaScript objects which has methods and properties to configure behavior and visual appearance. Every widget

should have an ID property which uniquely identifies it. You can specify this value using ID property of *JDash.Mvc.DomElement* class which all widgets on the server side is inheriting from. If you don't specify an ID value JDash automatically creates a unique value.

Consider the following on your view.

```
@(Html.JDash().DashboardView()
              .ID("myDashboard")
              .DesignMode(DashboardDesignMode.full).Render())
```

This will actually create a *jdash.ui.DashboardView* object on client side and set properties accordingly.

You can use *byId* method of *jdash.ui.registry* singleton to get a reference to a widget. For the above example below JavaScript code will get a reference to client widget.

```
var clientDashboard = Jdash.ui.registry.byId("myDashboard");
```

Some objects are static and only one instance is available. For example to get a reference to *ThemeManager* which allows managing themes you can use *jdash.ui.ThemeManager* singleton.

Below code changes active theme.

```
jdash.ui.ThemeManager.select(("flat", "q", true);
```

USING MESSAGE BUS

jdash.msgBus provides a centralized hub for publishing and subscribing to global messages by topic. One can subscribe to these messages by using *jdash.msgBus.subscribe*, and one can publish by using *jdash.msgBus.publish*.

```
jdash.msgBus.subscribe("some/topic", function () {
    console.log("received:", arguments);
});
// ...
jdash.msgBus.publish("some/topic", "one", "two");
```

DD uses message bus instead of events to broadcast messages. This loosely coupled architecture enables testing simpler.

Below code snippet illustrates how to check sender of message using message bus.

```
var self = this;
// subscribe to some/topic.
jdash.msgBus.subscribe("some/topic", function (event) {

  // check if this is the message coming from me
  if (event.sender == self) {
      console.info("Got message published from me!");
      // event.args.someArg will hold value 12
  }
});

// create an event.
var event = jdash.msgBus.createEvent(self, { someArg: 12 });
console.info("Publishing ...");
jdash.msgBus.publish("some/topic", event);
```

This is very useful when other objects publish same topic and you want to only process messages coming from a specific object.

Message bus supports cancelling messages also. For example, when a dashlet is about change visual state (i.e. user is maximizing it), it publishes *jdash/dashlet/visualStateChanging* topic. You can subscribe to this topic, check a condition and cancel the operation.

```
jdash.msgBus.subscribe("jdash/dashlet/visualStateChanging", function (event) {
    console.info("visualStateChanging received. Cancelling");
    event.cancel = true;
});
```

Publishing object can use deferred to check if a message is cancelled.

```
jdash.msgBus.publish("some/topic/happening", event).then(

// handler for success
function () {
  console.info("Successfully happened. Publishing...");
  messageBus.publish("some/topic/happened");
},

// handler for failure
function () {
  console.info("Cancelled");
});
```

jdash.msgBus.subscribe returns a simple object containing a *remove* method, which can be called to unsubscribe the listener. For better memory management you should consider calling this method when you don't need subscription anymore.

Below table common topics published.

Topic	Description
jdash/layout/dnd/dragStarting	Occurs when user tries to drag a dashlet.
jdash/layout/dnd/dragStarted	Occurs after user started to drag dashlet.
jdash/layout/dnd/dropped	Occurs after user dropped dashlet.
jdash/dashlet/visualStateChanging	Occurs before visual state of a dashlet is about to change.
jdash/dashlet/visualStateChanging	Occurs after visual state of a dashlet is changed.
jdash/dashlet/command/prepare	Occurs before creating pane commands.
jdash/dashlet/command/executing	Occurs before a command (remove, refresh, etc.) is executed.
jdash/dashlet/command/executed	Occurs after a command (remove, refresh, etc.) is executed.
jdash/dashlet/editor/validating	Occurs before saving/applying of dashlet editor.
jdash/dashlet/editor/saved	Occurs after saving/applying of dashlet editor.
jdash/dashlet/editor/canceling	Occurs before closing dashlet editor.
jdash/dashlet/editor/canceled	Occurs after closing dashlet editor.
jdash/dashboard/designMode/changing	Occurs before changing of design mode of a dashboard.
jdash/dashboard/designMode/changed	Occurs after design mode of a dashboard is changed.
jdash/dashboard/newdashlet	Occurs after adding a new dashlet to dashboard.
jdash/dashboard/loading/started	Occurs loading of a dashboard started.
jdash/dashboard/loading/completed	Occurs after loading of a dashboard is completed.
jdash/dashboard/unloaded	Occurs current dashboard is loaded.
jdash/dashboard/dashletContextCreated	Occurs when a DashletContext is created for a dashlet.
jdash/theme/changed	Occurs when theme/style has changed globally.

You can use *jdash.msgBus* to subscribe these topics. As a shortcut you can also use *subscribe* method of *DashletContext* if you are subscribing inside your dashlet. One benefit of using *DashletContext* is, when your dashlet is destroyed all subscriptions are removed automatically.

```
initialize: function (context, viewNode) {
    var context = this.$context();

    context.subscribe("jdash/dashlet/visualStateChanging", function (event) {
        // check if send is me
        if (event.sender == this) {
            console.info("Visual state is changing to " + event.args.state);

            //cancel it
            event.cancel = true;
        }
    });
}
```

MANAGING CLIENT SIDE ERRORS

JDash logs client side JavaScript errors to the console. Use console window of your browser to identify client side errors.

LOCALIZATION

JDash.Net uses *RequireJs* library *I18N* Bundle for localization and JDash for Asp.Net MVC automatically handles all requests by registering a route for localization files (*jdash/nls/{file}.js*).

nls is the directory inside root of JDash.Net client engine and it includes localization JavaScript files. Location can be changed using *ResourceManager.Package* method and by default it is inside *jdash* directory. So *scripts/jdash/nls* is the default root for localization.

When client engine needs a localized resource it makes a request to a JavaScript file named *jdash_XX.js* where *XX* is the language identifier like *fr* or *en-gb*. Handler first search for a physical file inside *nls* directory and if finds simply sends the contents of the file to client engine. If physical file cannot be located, it uses assembly resources.

This allows you to put a file inside *nls* directory and override default behavior. As an example if you want your dashboard to speak German simply copy language template file to *nls* directory and rename it as *jdash_de.js*. Client engine detects browsers language and Asp.Net handler loads your file automatically.

AUTHORIZATION

BASIC CONCEPTS

JDash supports both role and user based authorization which means permissions can be granted to both roles and users. Dashboards and dashlet modules can be authorized by JDash built-in authorization architecture and *authorizationEnabled* configuration settings must be set prior to get working.

Tip Although we provide a built-in authorization, with the help of Dynamic Query you can easily filter items.

To enable authorization either edit *web.config* JDash section.

PRIVILEGED ROLES

Sometimes you may need to skip authorization for users belonging to specific roles. As an example you may want administrators to view/edit everything.

Setting a comma separated roles names to *privilegedRoles* configuration settings allows you to do so. Below is a sample section.

```
<configSections>
    <section type="JDash.Configuration.DashboardSettingsSection,JDash" name="JDash" />
</configSections>
<JDash defaultProvider="SQLDashboardProvider" authorizationEnabled="true"
                                              privilegedRoles="Admin,Contributor" >
    <providers>
      ...
    </providers>
  </JDash>
```

This example demonstrates, users belonging to Admin or Contributor roles can access all dashboard and dashlet modules data from provider.

WORKING WITH ROLES

JDash requires a role provider to work with roles. Main responsibility of a role provider is telling assigned roles for a user.

For simplicity there exists two settings for providers. *knownRoleProvider* currently supports only *MembershipRoleProvider* value and if your application uses membership role provider settings this value allows JDash authorization architecture to work with membership provider.

roleProvider setting value allows you to set a type name which should be inherited from *System.Web.Security.RoleProvider* class.

```
<configSections>
    <section type="JDash.Configuration.DashboardSettingsSection,JDash" name="JDash" />
</configSections>
<JDash defaultProvider="SQLDashboardProvider" authorizationEnabled="true"
privilegedRoles="Admin,Contributor" >
    <providers>
      <!--Sample membership role provider-->
      <add  applicationName="DashboardApp" connectionString="SqlConstr"
name="SQLDashboardProviderForMembershipRoles" type="JDash.SqlProvider.Provider,
JDash.SqlProvider" knownRoleProvider="MembershipRoleProvider"  />

      <!--Sample of defining custom role providers-->
      <add  applicationName="DashboardApp" connectionString="SqlConstr"
name="SQLDashboardProviderForCustomRoles" type="JDash.SqlProvider.Provider,
JDash.SqlProvider" roleProvider="MyCustomRoleProvider.Provider, MyAssembly"  />
    </providers>
  </JDash>
```

WORKING WITH PERMISSIONS

Users can share their own dashboards and admins can set authorization information to dashlet modules. To do so, use *DashboardModel.authorization* and *DashletModule.authorization* property.

Authorization property is a list of *KeyValuePair* of <string, *PermissionModel*> objects.

Member	Description
Properties	
Key	Specifies a user name or role name. If *PermissionModel.authTarget* is set to everyone, can be unique identifier.
Value	Returns a permission of key value which is *PermissionModel*

Here is the members of this *PermissionModel*

Member	Description
Properties	
permission	Can be assigned to *view*, *edit* and *delete*. *view*: grant view permission. *edit*: grant edit permission.

	delete: grant delete permission.
authTarget	Can be *userName*, *roleName* or *everyone*.

DASHBOARD AUTHORIZATION

Below example demonstrates how to define authorization for a dashboard.

```
var newDashboard = new DashboardModel()
{
   title = dashTitle.Text,
};
newDashboard.metaData.created = DateTime.Now;
newDashboard.metaData.createdBy = "me";
newDashboard.metaData.group = "Startup Dashboards";
newDashboard.share = ShareModel.shared;
newDashboard.authorization = new List<KeyValuePair<string, PermissionModel>>();

//Below sample demonstrates that "Everyone  can view this dashboard"
newDashboard.authorization.Add(new KeyValuePair<string, PermissionModel>("users",
                               new PermissionModel()
                               {
                                   authTarget = AuthTarget.everyOne,
                                   permission = Permission.view
                               }));

//Below sample demonstrates that "Users in EditorTeam role can edit this dashboard"
newDashboard.authorization.Add(new KeyValuePair<string, PermissionModel>("EditorTeam",
                               new PermissionModel()
                               {
                                   authTarget = AuthTarget.roleName,
                                   permission = Permission.edit
                               }));

//Below sample demonstrates that "Joe user can delete this dashboard"
newDashboard.authorization.Add(new KeyValuePair<string, PermissionModel>("Joe",
                               new PermissionModel()
                               {
                                   authTarget = AuthTarget.userName,
                                   permission = Permission.delete
                               }));
JDashManager.Provider.CreateDashboard(newDashboard);
```

DASHLET MODULE AUTHORIZATION

Below example demonstrates how to define authorization for a dashboard.

```
var newModule = new DashletModuleModel();
newModule.title = "My module";
newModule.path = "[MVCDefault]";

newModule.config.Add("mvcConfig", new
{
   controller = "/Dashlets/Html",
});
```

```
newModule.authorization = new Dictionary<string, PermissionModel>();

//Below sample demonstrates that "Everyone  can add this dashlet module own dashboards"
newModule.authorization.Add(new KeyValuePair<string, PermissionModel>("users",
                                new PermissionModel()
                                {
                                    authTarget = AuthTarget.everyOne,
                                    permission = Permission.view
                                }));

//Below sample demonstrates that "Users in EditorTeam role can edit this Dashlet Module.
newModule.authorization. Add(new KeyValuePair<string, PermissionModel>("EditorTeam",
                                new PermissionModel()
                                {
                                    authTarget = AuthTarget.roleName,
                                    permission = Permission.edit
                                }));

//Below sample demonstrates that "Joe user can delete this dashlet module."
newModule.authorization.Add(new KeyValuePair<string, PermissionModel>("Joe",
                                new PermissionModel()
                                {
                                    authTarget = AuthTarget.userName,
                                    permission = Permission.delete
                                }));

JDashManager.Provider.CreateDashboard(newDashboard);
```

As an alternative JDash Management Portal can be used to set authorization information for dashlet modules.