



TABLE OF CONTENTS

Getting Started	5
Product Overview	5
Installation [TODO]	5
Architecture	5
UI JavaScript Library	5
Javascript Client Provider	6
Server Library	6
Features Summary	6
Before You Start	7
Javascript IDEs	7
Debugging Tools	7
Quick start	12
Using Mvc Dashboard	12
Inside DD	12
About AMD Modules	12
Object Oriented Javascript	14
Stateful Objects	16
Deferreds and Promises	17
Messagebus	18
Basic Concepts	19
Working with Client Providers	20
Creating Client Provider Instance	21
Developing new Providers	21
Working With Dashboards	22
Creating new dashboard	22

Querying Dashboards	23
Updating dashboard	23
Deleting dashboard	24
Displaying dashboard	24
Dashboard Design Modes	26
Working with Dashlet Modules.....	26
Registering new dashlet module	27
Querying dashlet modules	28
Updating dashlet modules	28
Deleting dashlet module	29
Dashlet Development.....	29
Hello World dashlet.....	29
Setup Client Environment	29
Setup Server Environment	30
Develop Your Dashlet.....	30
Register Your Dashlet	30
Creating a Test Dashboard	31
Adding Dashlet Instances To Dashboard	31
Developing Dashlets.....	32
Dashlet DOM Structure and Lifecycle	32
Working with DashletContext	33
Working with Dashlet Editors.....	35
Managing Dashlet Configuration	37
Setting Initial Configuration	39
Advanced Topics	40
Using Javascript Template Engines	40
Using Dijit Widgets	40

Using Backbone Views.....	44
Loading Resources On Demand	45
Loading CSS Files Dynamically.....	46
Loading JavaScript Files Dynamically.....	47
Customizing Dashlet Pane	48
Setting Pane Properties.....	49
Working with Pane Commands.....	51
Working With Layouts.....	54
About Layouts And Dashlet Position	54
Using Ready To Use Layouts.....	55
Developing New Layouts	57
Working With Themes.....	58
Retreiving Themes and Styles	58
Changing Theme.....	59
Persisting Selected Theme and Style.....	59
Responding to Theme and Style Changes	60
Developing New Themes.....	60
Working with Server Side	62
How to Configure ASP.NET MVC Application	62
Configure Server Side With Java	67
Create Custom Dashboard Provider With Java	68

GETTING STARTED

PRODUCT OVERVIEW

Kalitte MVC Dashboard lets you to add fantastic looking, HTML5 based, end user customizable dashboards and dashlets to your application with ease. You can integrate DD into your product, develop your dashlets and let your users design their own dashboards.

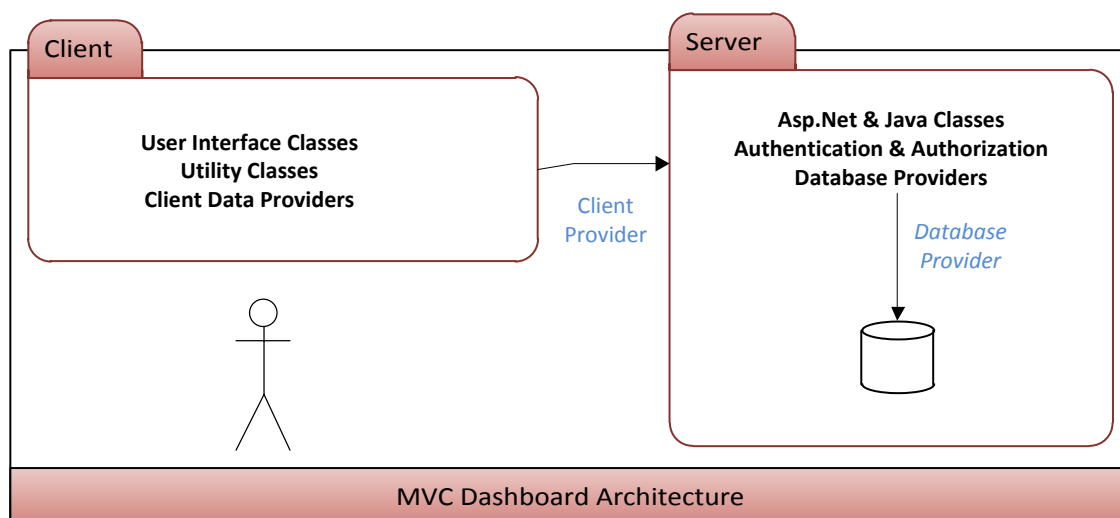
INSTALLATION [TODO]

- Using with Asp.Net
- Using with Java

ARCHITECTURE

DD includes three layers.

- ✓ Fully featured, HTML5 + CSS3 based UI JavaScript library
- ✓ Javascript Client Provider
- ✓ Server Application (Asp.Net MVC & Java, Database Providers for SQL Server, Oracle and MySQL)



UI JAVASCRIPT LIBRARY

Javascript library includes user interface. This is the layer which is responsible to retrieve dashboard data (i.e. list of dashboards, dashlets inside a dashboard and dashlet positions etc.) using active client provider and display dashboard and dashlets to end user.

Layer also has utility classes, interfaces (*mixins*), ready to use dashlet library. You can directly use those classes, extend them or easily integrate them into your application.

More information about each class in JavaScript library can be found at API documentation.

JAVASCRIPT CLIENT PROVIDER

This layer is implemented using Javascript and responsible to provide dashboard data to UI Javascript Library. It is completely up to the implementation of client provider how to provide dashboard data to UI layer.

Jsonrest provider (*klt/dash/provider/JsonRest/JsonRestProvider*) can be used to communicate with restful json data sources.

You can also develop your own client provider by implementing *klt/dash/provider/_ProviderMixin*.

SERVER LIBRARY

Server library includes ready to use restful json data sources and designed to be database independent.

Server library is responsible to provide/update metadata which may include dashboard definitions and dashlet modules and instances.

ABOUT ASP.NET SUPPORT

If you prefer Microsoft Windows on your server we provide a fully featured Asp.Net MVC application. Asp.Net MVC 3+ is officially supported and you can use Microsoft SQL Server, MySQL or Oracle as your database server.

If you want to develop new provider to access another data sources or web services, see Working on Server Side section.

ABOUT JAVA SUPPORT

We provide a fully featured Java Web Application with Restful service. You can use MySQL as your database server.

ABOUT DATABASE PROVIDERS

Server library is designed to be database independent and currently there are three providers implemented for Microsoft SQL Server, MySQL and Oracle. You can also implement your own provider, register it and use your favorite database server to store dashboard data.

More information about how to install and configure Asp.Net MVC can be found inside Working on Server Side section.

FEATURES SUMMARY

Below is a summary of list of features DD has or supports.

- Modern, Javascript + HTML5 + CSS3 based browser independent client library which lets you to easily integrate DD into your application or product.
- Asp.Net support for Windows, J2SE support for all platforms.
- Loosely coupled architectural design, which provides high level customization both on server and client side.

- Premise and cloud support.
- Ready to use for Microsoft SQL Server, Oracle and MySQL databases.
- Database independent architecture which lets you to use your favorite database server.
- Mobile support for iOS, Android and Windows.
- Modular dashlet development using AMD (Asynchronous Module Definition) format.
- Unlimited dashboard layouts including table, row-column based and absolute positioned.
- Theming, also supports custom styles and templates.
- Ready to use open source dashlet library.

BEFORE YOU START

Before going further, if you are new to JavaScript development, we recommend you to setup a development environment and use helper tools.

JAVASCRIPT IDES

If you are a Microsoft.Net developer, you may prefer using Visual Studio 2010 and upper versions (Visual Studio 2012 is recommended for JavaScript development) for JavaScript development.

For Java developers, you may prefer using *Eclipse*, *Aptana* or *KomodoEdit*.

For a discussion on JavaScript IDEs see <http://stackoverflow.com/questions/925219/best-javascript-editor-or-ide-with-intellisense-and-debugging-possibly>.

DEBUGGING TOOLS

We recommend Firefox web browser + Web Developer plug-in + Firebug plug-in as primary development tools. They improve your development comfort by easily finding and fixing bugs and detect visual problems easier. You can download them by using following links:

- Firefox® Web Browser : <http://www.mozilla.org/en-US/firefox/new/>
- Web Developer® Plug-in : <https://addons.mozilla.org/en-US/firefox/addon/web-developer/>
- Firebug® : <https://addons.mozilla.org/en-US/firefox/addon/firebug/>

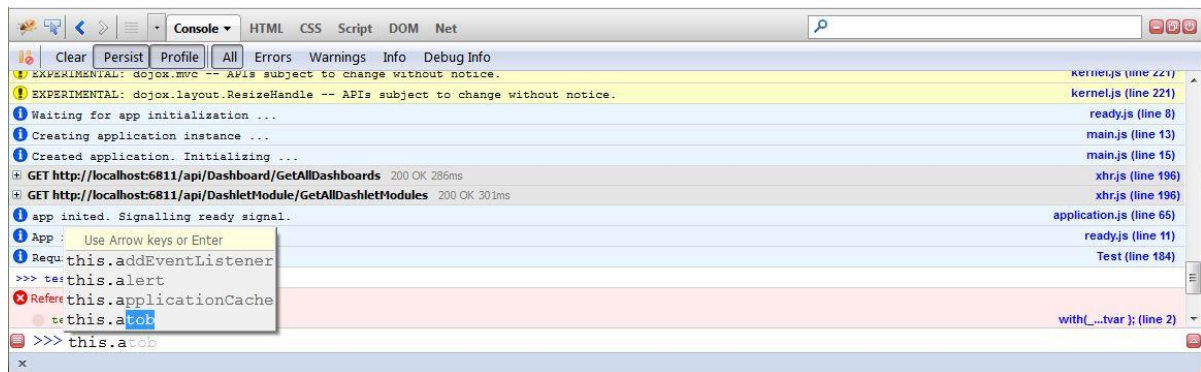
FIREFOX®

Firefox is one of the popular web browsers in the world and also provides you a development environment with the help of lots of plugins.

FIREBUG®


Firebug lets you to debug javascript and analyse document structure.

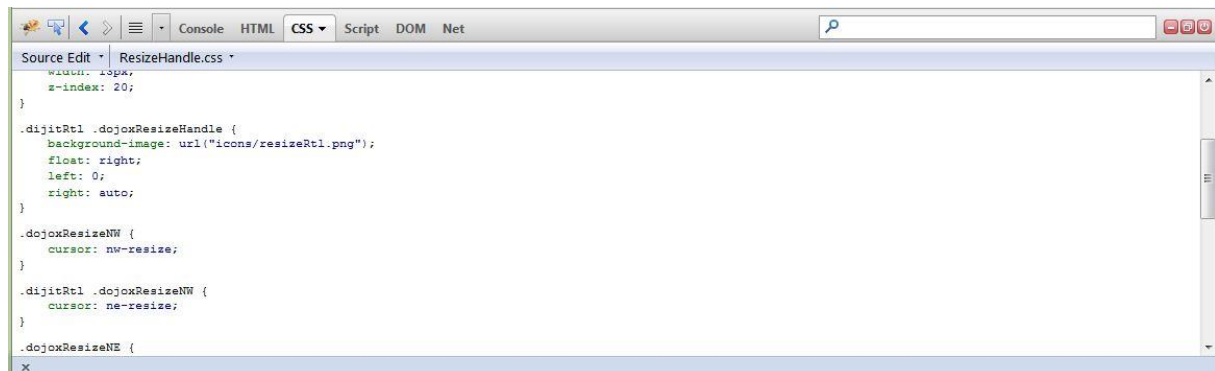
After you install Firebug plug-in to your Firefox, a button () will be added to your browsers toolbar. Click to open plugin at bottom of your browser.



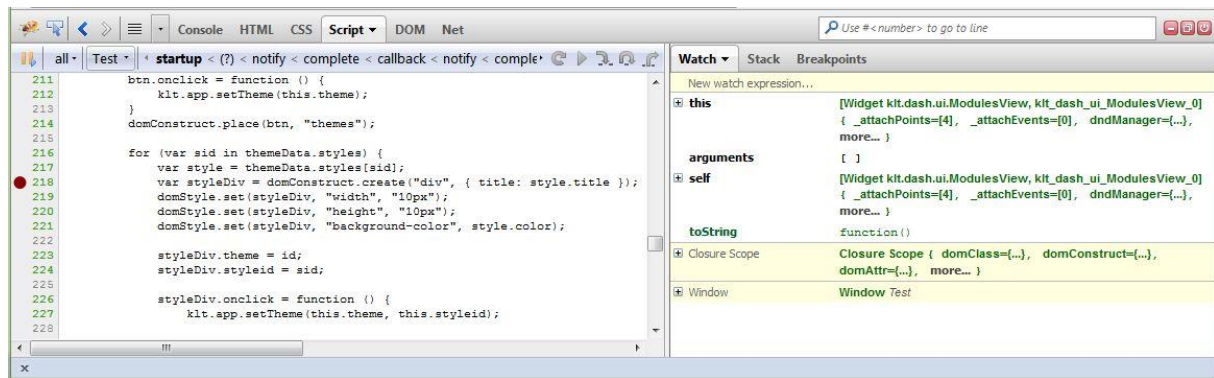
This is Firebug console tab. You can observe every browser feed inside, i.e. errors, warnings, network communications, etc. You can also use command line to run custom javascript codes.



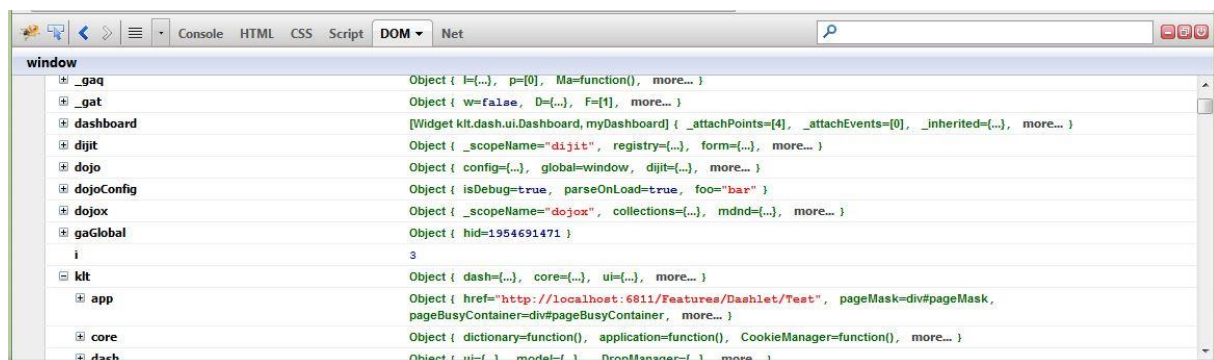
HTML tab view helps views source code and also allows you to edit html code of the page. Any changes will effect browser in real time. You can also edit style definations. If you use  button you can select any html element from browser by using your mouse and display it in HTML tab.



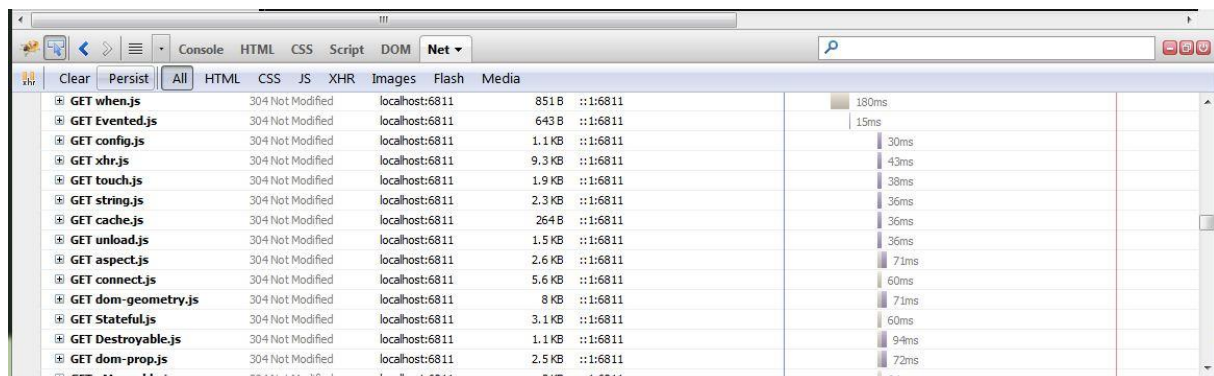
Above screen shot is CSS tab of Firebug. You can browse and edit style sheet file here.



The script tab is the most important and probably the most widely used feature of Firebug. It allows you to debug javascript. You can place breakpoints and analyze your code step by step. You can also define any javascript expression in watch section.



Every single dom object of your app will be listed in the DOM tab. You can browse dom object in their own hierarchy.

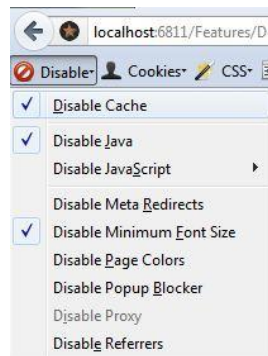


You can use Net tab to observe network traffic of your application. Any single request and its result will be listed here.

WEB DEVELOPER®

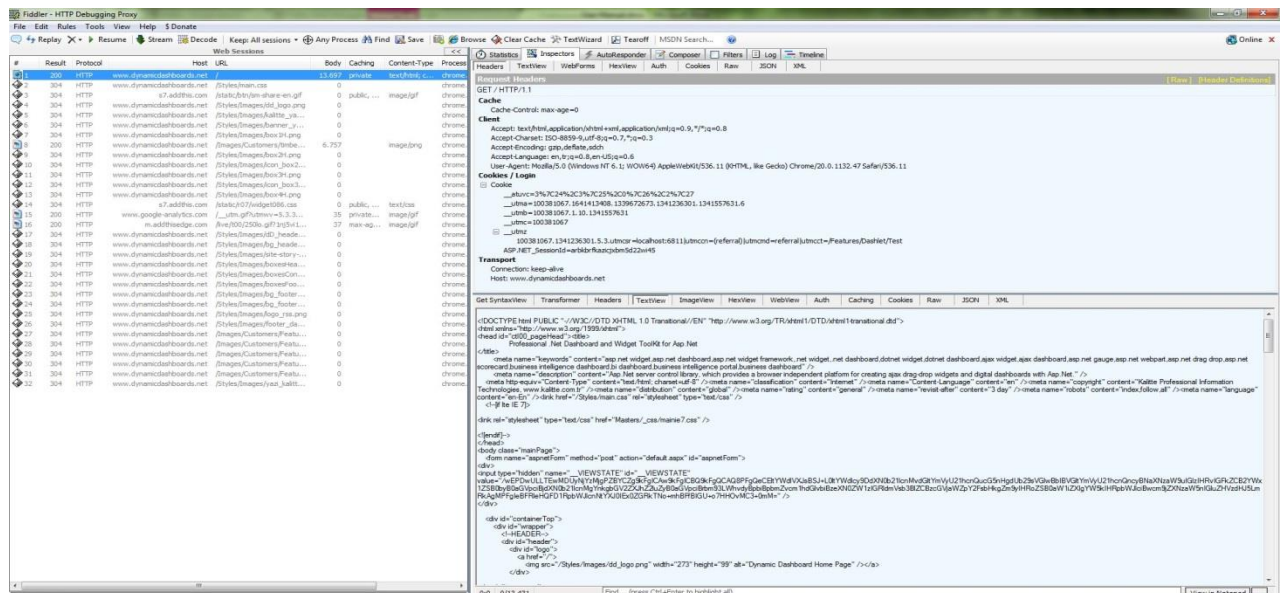
Web Developer is a plug-in that has many development features.

When you add Web Developer to Firefox a tool bar will appear. In “Disable” menu of bar there is an option called “Disable Cache”. Be sure it is checked when you are developing with JavaScript.



FIDDLER®

Fiddler logs all web traffic between client and internet. Here is a general view of Fiddler:



Left menu displays list of data packages.

#	Result	Protocol	Host	URL	Body	Caching	Content-Type	Process
1	200	HTTP	www.dynamicdashboards.net	/	13.697	private	text/html; c...	chrome.
2	304	HTTP	www.dynamicdashboards.net	/Styles/main.css	0			chrome.
3	304	HTTP	s7.addthis.com	/static/btn/sm-share-en.gif	0	public, ...	image/gif	chrome.
4	304	HTTP	www.dynamicdashboards.net	/Styles/Images/dd_logo.png	0			chrome.
5	304	HTTP	www.dynamicdashboards.net	/Styles/Images/kalitte_ya...	0			chrome.
6	304	HTTP	www.dynamicdashboards.net	/Styles/Images/banner_y...	0			chrome.
7	304	HTTP	www.dynamicdashboards.net	/Styles/Images/box1H.png	0			chrome.
8	200	HTTP	www.dynamicdashboards.net	/Images/Customers/timbe...	6.757		image/png	chrome.
9	304	HTTP	www.dynamicdashboards.net	/Styles/Images/box2H.png	0			chrome.
10	304	HTTP	www.dynamicdashboards.net	/Styles/Images/icon_box2...	0			chrome.
11	304	HTTP	www.dynamicdashboards.net	/Styles/Images/box3H.png	0			chrome.
12	304	HTTP	www.dynamicdashboards.net	/Styles/Images/icon_box3...	0			chrome.
13	304	HTTP	www.dynamicdashboards.net	/Styles/Images/box4H.png	0			chrome.
14	304	HTTP	s7.addthis.com	/static/r07/widget086.css	0	public, ...	text/css	chrome.
15	200	HTTP	www.google-analytics.com	/__utm.gif?utmwv=5.3.3...	35	private...	image/gif	chrome.
16	200	HTTP	m.addthisedge.com	/live/t00/250lo.gif?1nj5vi1...	37	max-ag...	image/gif	chrome.

Selected package details can be viewed in the right menu. Top section displays request details and the bottom section displays response details. Both sections allow user to read request and response data such as JSON and XML formats.

Statistics
Inspectors
AutoResponder
Composer
Filters
Log
Timeline

Headers
TextView
WebForms
HexView
Auth
Cookies
Raw
JSON
XML

Request Headers
[Raw] [Header Definitions]

GET /watch/13264411?m=901324&wmode=5&callback=_ymjsip361804&page-ref=http%3A%2F%2Fwww.sozluk.net%2F&page-url=http%3A%2F%2Fwww.sozluk.net%2Findex.php%3Fword%3D%25C3%25A7e%25C5%259F&id%3D%25C3%25A7e%25C5%259F&browser-info=vc%3A%2F%2Fwww.sozluk.net%2Findex.php%3Fword%3D%25C3%25A7e%25C5%259F&id%3D%25C3%25A7e%25C5%259F

Client
Accept: */*
Accept-Charset: ISO-8859-9,utf-8;q=0.7,*;q=0.3
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en,tr;q=0.8,en-US;q=0.6
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/536.11 (KHTML, like Gecko) Chrome/20.0.1132.47 Safari/536.11

Cookies / Login
Cookie
yabs-sid=1116906851341558349
yandexuid=2459648861340351275

Miscellaneous
Referer: http://www.sozluk.net/index.php?word=%C3%A7e%C5%9F&id%3D%25C3%25A7e%25C5%259F&browser-info=vc%3A%2F%2Fwww.sozluk.net%2Findex.php%3Fword%3D%25C3%25A7e%25C5%259F&id%3D%25C3%25A7e%25C5%259F

Transport
Connection: keep-alive
Host: mc.yandex.ru

Get SyntaxView
Transformer
Headers
TextView
ImageView
HexView
WebView
Auth
Caching
Cookies
Raw
JSON
XML

Response Headers
[Raw] [Header Definitions]

HTTP/1.1 200 OK

Cache
Cache-Control: private, no-cache, no-store, must-revalidate, max-age=0
Date: Fri, 06 Jul 2012 07:05:57 GMT
Expires: Fri, 06 Jul 2012 07:05:57 GMT
Pragma: no-cache

Cookies / Login
P3P: CP="NOI DEVa TAIa OUR BUS UNI STA"

Entity
Content-Length: 74
Content-Type: text/javascript
Last-Modified: Fri, 06 Jul 2012 07:05:57 GMT

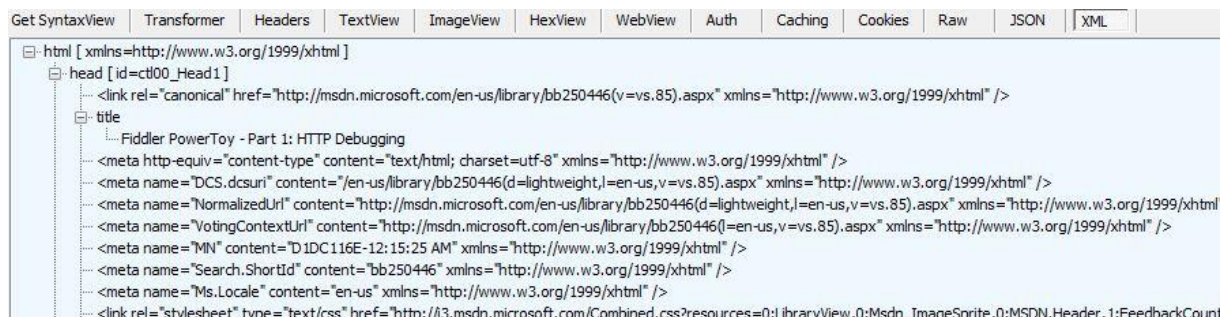
Miscellaneous
Server: Phantom/0.0.0

Transport
Connection: Keep-Alive
Proxy-Connection: Keep-Alive
Via: 1.1 GATE

Here is a JSON response sample in Fiddler:



A XML response in Fiddler:



QUICK START

[TODO]

USING MVC DASHBOARD

INSIDE DD

In this section, core classes and approaches used inside DD are discussed. If you are new to JavaScript, for now, we recommend you just have an idea about these concepts. When you start developing dashlets these concepts will be simpler to understand and use.

ABOUT AMD MODULES

DD uses AMD format for both its internal modules and dashlets. The Asynchronous Module Definition (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.), MooTools (2.0), Firebug (1.8) and even jQuery (1.7).

API SPECIFICATION

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.

EXAMPLES

Assume we have a file named `chartDashlet.js`.

```
define(function () {  
  
    // implement first time initialization logic for your module.  
    return function () {  
        // implement your logic related with  
        // each dashlet instance created  
    }  
});
```

If your module has dependencies that should be loaded before your module loads, you can use the *dependencies* parameter.

```
// Load baseChart.js and chartSerie.js modules located in same directory  
// before defining module.
```



```
define(["./baseChart", "./chartSerie"], function (baseChart, chartSerie) {
    // Load baseChart.js and chartSerie.js files located in the same directory.
    // baseChart and chartSerie parameters are return value of dependent modules.
    return function () {
    }
});
```

`require` function loads a module.

```
// load chartDashlet.js and set its return
// value to chartDashlet parameter
require(["./chartDashlet"], function (chartDashlet) {
    // use chartDashlet
});
```

When you want to use `chartDashlet` using `require` function;

1. AMD loader tries to load `chartDashlet.js` file.
2. Since there exists dependencies to `baseChart` and `chartSerie`, they are loaded first.
3. Return value is returned.

BOOTSTRAPPING YOUR APPLICATION

Bootstrap file is responsible for configuring module loader and loading initially important dependencies.

In the below example we configure module loader to set base url in which scripts are located and configure packages. Also we tell module loader to load `main.js` file automatically inside `app` folder.

```
require({
    baseUrl: '/Scripts',
    packages: [
        'dojo',
        'dijit',
        'dojox',
        'app',
        'klt'
    ]
}, ['app']);
```

For more information about AMD use the following links.

- <http://www.sitepen.com/blog/2012/06/25/amd-the-definitive-source/>
- <https://dojotoolkit.org/documentation/tutorials/1.7/modules/>
- <http://addyosmani.com/writing-modular-js/>
- <https://github.com/amdjs/amdjs-api/wiki/AMD>
- <http://requirejs.org/>

OBJECT ORIENTED JAVASCRIPT

JavaScript uses prototype-based inheritance, not class-based inheritance (which is used by most programming languages). `klt/core/declare`, which is based on `dojo/declare`, provides the ability to simulate class-based inheritance.

```
// in my/Person.js
define(['klt/core/declare'], function(declare){
    return declare(null, {
        constructor: function(name, age, residence){
            this.name = name;
            this.age = age;
            this.residence = residence;
        }
    });
});

// elsewhere
require(['my/Person'], function(Person){
    var folk = new Person("phiggins", 42, "Tennessee");
});
```

```
declare(id?, bases?, factory);
```

The first optional parameter is a unique identifier for class. Optionally you can use second parameter for inheritance.

```
// in my/Employee.js
define(['klt/core/declare', 'my/Person'], function(declare, Person){
    return declare(Person, {
        constructor: function(name, age, residence, salary){
            // The "constructor" method is special: the parent class (Person)
            // constructor is called automatically before this one.

            this.salary = salary;
        },

        askForRaise: function(){
            return this.salary * 0.02;
        }
    });
});
```

To call superclass methods use `inherited`.

```
// in my/Boss.js
define(['klt/core/declare', 'my/Employee'], function(declare, Employee){
    return declare(Employee, {
        // override the askForRaise function from the Employee class
        askForRaise: function(){
            return this.inherited(arguments) * 20; // boss multiplier!
        }
    });
});
```

```
});

// elsewhere
require(['my/Employee', 'my/Boss'], function(Employee, Boss){
    var kathryn = new Boss("Kathryn", 26, "Minnesota", 9000),
        matt    = new Employee("Matt", 33, "California", 1000);

    console.log(kathryn.askForRaise(), matt.askForRaise()); // 3600, 20
});
```

More information and a detailed discussion about declare can be found at <http://livedocs.dojotoolkit.org/dojo/declare>.

STATEFUL OBJECTS

`klt/core/Stateful` is the base class (inheriting `dojo/Stateful`) for stateful objects. Stateful means;

- Ability to get and set properties which may have side effects.
- Ability to watch/monitor property changes.

Many classes inside DD framework inherit from `klt/core/Stateful` including `DashletContext`, `ConfigModel`, `DashletModel`, `DashboardModel` etc. This can be very useful for creating live bindings that utilize current property states and must react to any changes in properties. It also allows a developer to customize the behavior of accessing the property by providing auto-magic getters and setters.

```
require(["klt/core/Stateful", "klt/core/declare"],
function(Stateful, declare){
    // Subclass klt/core/Stateful:
    var MyClass = declare([Stateful], {
        foo: null,
        _fooGetter: function(){
            return this.foo;
        },
        _fooSetter: function(value){
            this.foo = value;
        }
    });

    // Create an instance and set some initial property values:
    myObj = new MyClass({
        foo: "baz"
    });

    // Watch changes on a property:
    myObj.watch("foo", function(name, oldValue, value){
        // Do something based on the change
    });

    // Get the value of a property:
    myObj.get("foo");
```



```
// Set the value of a property:
myObj.set("foo", "bar");
});
```

More information about stateful objects can be found at <http://livedocs.dojotoolkit.org/dojo/Stateful>.

DEFERREDS AND PROMISES

`klt/core/Deferred` is used as the foundation for managing asynchronous threads. Use this class to return a promise that gets resolved when the asynchronous thread is complete. In order trigger a callback to occur when the thread is complete, the `then` method is used. As well as the thread can be informed to cancel itself by using the `cancel` method. If the thread has completed, then the `then` callback will be executed immediately.

```
require(["klt/core/Deferred"], function(Deferred) {
    var deferred = new Deferred(function(reason) {
        // do something when the Deferred is cancelled
    });

    // do something asynchronously

    // provide an update on progress:
    deferred.progress(update);

    // when the process finishes:
    deferred.resolve(value);

    // performing "callbacks" with the process:
    deferred.then(function(value) {
        // Do something when the process completes
    }, function(err) {
        // Do something when the process errors out
    }, function(update) {
        // Do something when the process provides progress information
    });

    // to cancel the asynchronous process:
    deferred.cancel(reason);
});
```

`klt/core/Deferred` is based on `Dojo/Deferred`. For a detailed documentation see <http://livedocs.dojotoolkit.org/dojo/Deferred>

`klt/core/when` is designed to make it easier to merge coding of synchronous and asynchronous threads. Accepts promises but also transparently handles non-promises. If no callbacks are provided returns a promise, regardless of the initial value. Also, foreign promises are converted.

If callbacks are provided and the initial value is not a promise, the callback is executed immediately with no error handling. Returns a promise if the initial value is a promise or the result of the callback otherwise.

```
require(["klt/core/when"], function(when) {
```

```

when(someValue, function(value) {
    // do something when resolved
}, function(err) {
    // do something when rejected
}, function(update) {
    // do something on progress
});
});

```

More information can be found at <http://livedocs.dojotoolkit.org/dojo/when>.

MESSAGEBUS

`klt/core/messageBus` provides a centralized hub for publishing and subscribing to global messages by topic. One can subscribe to these messages by using `messageBus.subscribe`, and one can publish by using `messageBus.publish`.

```

require(["klt/core/messageBus"], function(messageBus) {
    messageBus.subscribe("some/topic", function() {
        console.log("received:", arguments);
    });
    // ...
    messageBus.publish("some/topic", "one", "two");
})

```

DD uses `messagebus` 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.
messageBus.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 = messageBus.createEvent(self, { someArg: 12 });
console.info("Publishing ...");
messageBus.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. For example, when a dashlet is about change visual state, it publishes *klt/dash/dashlet/visualStateChanging* topic. You can subscribe to this topic, check a condition and cancel the operation.

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

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

```
messageBus.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");
    });
```

`messageBus.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.

BASIC CONCEPTS

Below concepts are used in this document.

Dashboard; Dashboard is the main container which should include a *layout* and a number of optional dashlets.

Layout; Layout is the container for *dashlets* and responsible for positioning and lay outing of them.

DashletPane; DashletPane or simply *Pane* is the container for a single dashlet. Pane includes a header and body which holds Dashlet instance.

DashletModule; DashletModule is a logical container which describes common properties and AMD formatted JavaScript dashlet file. It's class of a dashlet.

DashletEditorModule; DashletEditorModule is an AMD formatted JavaScript file which defines an editor for a dashlet module.

Dashlet; Dashlet is an instance of a DashletModule.

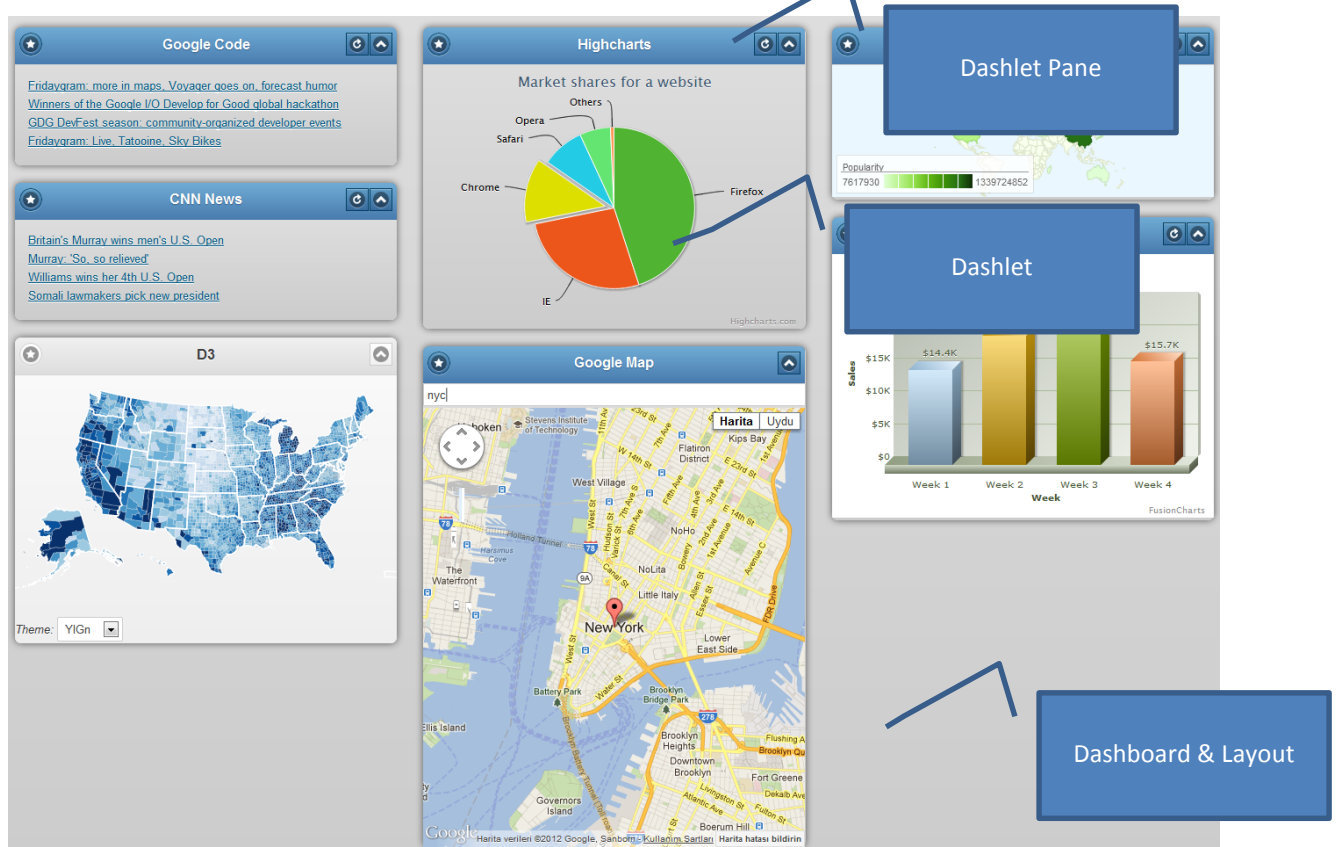
DashletContext; DashletContext is the bridge between your dashlet and DD Framework.

Client Provider; Client providers are JavaScript classes which are responsible for retrieving and persisting metadata. Metadata can be list of Dashboards for a user or position of a dashlet.

Server Provider; Server providers are server side classes which are used to retrieve and persist dashboard data.

When a dashboard is about to be displayed on a client browser;

- Using Client Provider, Dashboard data is retrieved and Layout is created.
- Dashlets which belong to that Dashboard are retrieved using client provider. For each Dashlet data, one DashletPane and one Dashlet is created and positioned by Layout.



WORKING WITH CLIENT PROVIDERS

`klt/dash/provider/Manager` is the main singleton to work with client providers.

Use `list` method to get a list of registered client providers, `getInstance` method to get an instance for a provider, `register` method to register a new provider.

```
require(["klt/dash/provider/Manager", "klt/dash/provider/Manager"],
function (Manager) {
    // Get a list of registered providers.
    var providers = Manager.list();
    for (var providerKey in providers) {
        var providerData = providers[providerKey];
    }
});
```

```

    // type returns type of provider.
    var providerType = providerData.type;

    // if provider is already used use instance
    // property to get a reference
    var providerInstance = providerData.instance;
    console.info(providerKey + "'s description says " +
        (providerType.description ?
            providerType.description : "nothing"));
    console.info(providerInstance ? "Provider instance is created":
        "Provider is only registered")
}
});

```

Sample output is:

```

JsonRest's description says Provider for Jsonrest data stores. © Kalitte.
Provider instance is created

```

CREATING CLIENT PROVIDER INSTANCE

To get an instance for a provider, you can use `getInstance` method. If you don't pass a value for `providername` parameter, default provider - that's the first registered provider's name - is used.

```

require(["klt/dash/provider/Manager",
    "klt/dash/provider/JsonRest/JsonRestProvider"],
function (Manager) {
    var jsonRest = Manager.getInstance("JsonRest");
    var defaultProvider = Manager.getInstance();
});

```

`getInstance` method checks if an instance for the requested provider is created before. If so, existing instance is returned. If not a new instance is created.

DEVELOPING NEW PROVIDERS

Out of the box, `JsonRest` provider is ready to use and most of time will be enough for your requirements. Although, when needed you can develop your own provider by starting from scratch or inheriting `JsonRest` provider and override its methods.

To start from scratch, use `klt/dash/provider/_ProviderMixin`.

```

define(["klt/core/declare",
    "klt/dash/provider/_ProviderMixin",
    "klt/dash/provider/Manager"],
function (declare, _ProviderMixin, Manager) {
    var providerClass = declare('klt.dash.provider.MyProvider',
        [_ProviderMixin], {
        // implement your provider.
    });
});

```

```

    // getDashboard: function (id) {},
    // saveDashboard: function (dashboard) {}
    // ...
  });

  Manager.register('MyProvider', providerClass);
  return providerClass;
});

```

See `klt/dash/provider/_ProviderMixin` class for the list of methods that your provider should support.

WORKING WITH DASHBOARDS

Client providers support creating, retrieving, updating and deleting (CRUD) dashboards.

`klt/dash/model/DashboardModel` is the base class which represents a dashboard. After retrieving a dashboard, it can be displayed using `klt/dash/ui/DashboardView` class.

CREATING NEW DASHBOARD

To create a new dashboard, you need to create a `DashboardModel` instance, set its properties and use `createDashboard` method of client provider.

```

require(["klt/dash/provider/Manager",
  "klt/dash/model/DashboardModel",
  "klt/core/when",
  "klt/dash/provider/JsonRest/JsonRestProvider"],
function (Manager, DashboardModel, when) {

  // Create new DashboardModel instance and set its properties.
  var dashboard = new DashboardModel({
    title: "My first dashboard",
    layout: "table"
  });

  // Get instance of default client provider
  var provider = Manager.getInstance();

  // Call createDashboard method of provider. Use when function
  // to handle both asynchronous and synchronous calls.
  when(provider.createDashboard(dashboard),

    // Handler for success
    function (result) {
      console.info("Dashboard created successfully. Assigned id is "+
        result.id);
    },

    // Optional error handler
    function (err) {
      console.error(err);
    }
  );
}

```

```
});
});
```

Above code snippet creates a dashboard using default values.

QUERYING DASHBOARDS

Below table lists key methods of client provider which can be used to query dashboards.

Member	Description
<i>getDashboard</i>	Retrieves the DashboardModel object for a specific id.
<i>getUserDashboards</i>	Returns an array of DashboardModel objects belonging to a specific user.
<i>getAllDashboards</i>	Returns an array of all DashboardModel objects.

```
require(["klt/dash/provider/Manager",
        "klt/dash/model/DashboardModel",
        "klt/core/when", "klt/dash/provider/JsonRest/JsonRestProvider"],
function (Manager, DashboardModel, when) {
    var provider = Manager.getInstance();
    //Get all dashboard instances
    when(provider.getAllDashboards(), function (Dashboards) {
        //your logic here...
    }, function (err) { console.error(err);
    });
});
```

More information about methods can be found at [api documentation](#) of client provider.

UPDATING DASHBOARD

`saveDashboard` method of client provider lets you to save changes for a `DashboardModel` object.

You can use `getDashboard` method of client provider to get a reference to an existing `DashboardModel` object, change its properties and call `saveDashboard`.

```
require(["klt/dash/provider/Manager",
        "klt/core/when",
        "klt/dash/provider/JsonRest/JsonRestProvider"],
function (Manager, dashboard, when) {
    var provider = Manager.getInstance();
    when(provider.getDashboard(224),
        function (dashboard) {
            //update "title" property of dashboard
            dashboard.set("title", "My Dashboard");
            //Save changes
            when(provider.saveDashboard(dashboardToUpdate),
                function () {
                    //logic on success
                },
            ),
        },
    );
});
```

```

        function () {
            //logic on fail
        }
    );
},
function (err) {
    console.error(err);
});
});

```

DELETING DASHBOARD

`deleteDashboard` method of client provider lets you to remove a dashboard.

```

require(["klt/dash/provider/Manager",
    "klt/dash/ui/DashboardView",
    "klt/core/when", "klt/dash/provider/JsonRest/JsonRestProvider"],
function (Manager, dashboard, when) {
    //Create provider instances
    var provider = Manager.getInstance();
    //Delete dashboard by id
    when(provider.deleteDashboard(233),
        //Handle deferred object
        function () {
            //logic on success
        },
        function () {
            //logic on fail
        });
});

```

DISPLAYING DASHBOARD

`klt/dash/ui/DashboardView` class is responsible for displaying a dashboard.

`DashboardView` constructor takes two parameters.

```

new DashboardView(/* Object? */ params, /* DomNode|string */ srcNodeRef)

```

`params` contains the initialization parameters. `srcNodeRef` is the id of the dom node you want to place the view. This parameter also accepts the dom node itself, instead of its id.

If not specified default provider is used by *DashboardView*. You can set a provider using *params* object.

```

require(["klt/dash/provider/Manager",
    "klt/dash/ui/DashboardView",
    "klt/dash/provider/JsonRest/JsonRestProvider",
    "klt/domReady!"],
function (Manager, DashboardView) {
    var jsonRest = Manager.getInstance("JsonRest");
    var view = new DashboardView({ provider: jsonRest }, "node");
});

```



```
});
```

To get a reference to the active provider which `DashboardView` uses, you can use `DashboardView.provider` property.

Below code snippet uses `getAllDashboards` method of default provider to get a reference to a `DashboardModel` object and `Load` method of `klt/dash/ui/DashboardView`.

```
<div id="dashboardContainer"></div>

<script type="text/javascript">
  require(["klt/dash/provider/Manager",
    "klt/dash/model/DashboardModel",
    "klt/core/when",
    "klt/dash/ui/DashboardView",
    "klt/domReady!"],
    function (Manager, DashboardModel, when, DashboardView) {

      // Create a DashboardView instance using initialization
      // parameters and id of container node.
      var viewer = new DashboardView({ id: "myViewer" },
        "dashboardContainer");

      // call startup method.
      viewer.startup();

      var provider = Manager.getInstance();
      when(provider.getAllDashboards(), function (allDashboards) {

        // Get a reference to first dashboard.
        var firstDashboard = allDashboards[1];

        // use DashboardView.Load method to load dashboard model.
        when(viewer.load(firstDashboard), function () {
          console.info("Loaded dashboard id:" + firstDashboard.id);
        }, function (err) {
          console.error(err);
        });
      }, function (err) {
        console.error(err);
      });
    });
</script>
```

`DashboardView.Load` method also accepts id value of an existing dashboard. Provider is used to query dashboard data.

```
require(["klt/dash/provider/Manager",
  "klt/core/when",
  "klt/dash/ui/DashboardView",
  "klt/domReady!"],
  function (Manager, DashboardModel, DashboardView) {
```

```

var viewer = new DashboardView({ id: "myViewer" }, "dashboardContainer");
viewer.startup();
when(viewer.load("224"), function (result) {
    console.warn("Loaded dashboard id:" + result.id);
}, function (err) {
    console.error(err);
});
});
});

```

DASHBOARD DESIGN MODES

Dashboard design modes let you to set if a dashboard can be edited by users.

`klt/dash/core/designMode` object can be used to define design modes. A dashboard can have one of the following design modes.

DesignMode	Description
<i>none</i>	Dashboard is read-only. Users cannot add new dashlets nor re-organize / edit existing dashlets.
<i>dashboard</i>	Users can edit layout of dashboard and re-organize existing dashlets.
<i>dashlet</i>	Users can only edit existing dashlets.
<i>full</i>	Users are able to edit layout of dashboard, add new dashlets, re-organize / edit existing dashlets.

Initial design mode of `DashboardView` object can be set as an initialization parameter. If you don't set, it defaults to `designMode.none`.

```

require(["klt/dash/ui/DashboardView",
    "klt/dash/core/designMode"],
function (DashboardView, designMode) {
    var view = new DashboardView({ designMode: designMode.full }, "node");
    ...
});

```

Use `set` method to set design mode of an existing `DashboardView` object.

```

require(["klt/dash/ui/DashboardView",
    "klt/dash/core/designMode"],
function (DashboardView, designMode) {
    var view = new DashboardView(null, "node");
    view.load("200");
    ...
    view.set("designMode", designMode.dashboard);
});

```

WORKING WITH DASHLET MODULES

Client provider allows developers to register dashlet modules, query existing modules and do CRUD operations on them. In this section, only client provider methods related with dashlet modules will

be explained. More information about dashlet modules can be found inside Dashlet Development section.

REGISTERING NEW DASHLET MODULE

To register a dashlet module, create an instance of `klt/dash/model/DashletModuleModel` and use `createDashletModule` method of client provider to save it.

```
require(["klt/dash/model/DashletModuleModel",
        "klt/dash/provider/Manager",
        "klt/core/when", "klt/dash/provider/JsonRest/JsonRestProvider"],
function (DashletModuleModel, Manager, when) {
    //Create dashlet module model
    var myModule = new DashletModuleModel({
        // AMD module definition path
        path: "myApp/dashlets/MyDashlet/MyDashletModule",
        title: "My Dashlet",
    });

    //Create a provider instance
    var myProvider = Manager.getInstance();

    //Add module to your data source
    when(myProvider.createDashletModule(myModule),
        function () {
            //logic on success
        },
        function () {
            //logic on fail
        }
    );
});
```

`DashletModuleModel` has a *config* object which allows you to set a configuration for a module.

```
require(["klt/dash/model/DashletModuleModel",
        "klt/dash/provider/Manager",
        "klt/core/when", "klt/dash/provider/JsonRest/JsonRestProvider"],
function (DashletModuleModel, Manager, when) {
    var myModule = new DashletModuleModel({
        path: "myApp/dashlets/MyDashlet/MyDashletModule",
        title: "My Dashlet",
        //Define module configuration here
        config: {
            // Only one instance of a module is allowed in a dashboard
            singleInstance: true,
            //About menu of dashlet
            about: {
                url: "http://www.dynamicdashboards.net",
                copyright: "Kalitte"
            }
        },
    });
```

```

    //Define editor module path
    editor: {
        path: "myApp/dashlets/MyDashlet/MyEditorModule"
    }
}
});
});

```

QUERYING DASHLET MODULES

Below table lists key methods of client provider which can be used to query dashlet modules.

Method	Description
<code>getDashletModule</code>	Returns a <code>DashletModuleModel</code> object with a specific id.
<code>getDashletModuleByTitle</code>	Returns a <code>DashletModuleModel</code> object with a specific title.
<code>getDashletModuleByPath</code>	Returns a <code>DashletModuleModel</code> object using its path value.
<code>getUserDashletModules</code>	Returns an array of <code>DashletModuleModel</code> instances for a user.
<code>getAllDashletModules</code>	Returns an array of all <code>DashletModuleModel</code> instances.

```

require(["klt/dash/provider/Manager",
        "klt/core/when",
        "klt/dash/provider/JsonRest/JsonRestProvider"],
function (Manager, when, JsonRest) {
    var provider = Manager.getInstance();
    when(provider.getAllDashletModules(),
        function (modules) {
            //logic on success
        },
        function () {
            //logic on fail
        }
    );
});

```

UPDATING DASHLET MODULES

`saveDashletModule` method of client provider lets you to save changes for a `DashletModuleModel` object.

You can use `getDashletModule` method of client provider to get a reference to an existing `DashletModuleModel` object, change its properties and call `saveDashletModule` method.

```

require(["klt/dash/provider/Manager",
        "klt/core/when",
        "klt/dash/provider/JsonRest/JsonRestProvider"],
function (Manager, when, JsonRest) {
    var provider = Manager.getInstance();
    when(provider.getDashletModule("70"),
        function (module) {
            //Update module title
        }
    );
});

```

```

module.set("title", "Updated Title")
//Get config object
var config = module.get("config");
//Change single instance property of module config
config.set("singleInstance", false);
//Save changes to your data source
when(provider.saveDashletModule(module),
    //Handle deferred object
    function () {
        //logic on success
    },
    function () {
        //logic on fail
    });
},
function () {
    //logic on load fail
}
);
});

```

DELETING DASHLET MODULE

`deleteDashletModule` method of client provider lets you to remove a dashlet module. It's up to the provider what to do when deleting a dashlet module which has dashlet instances.

```

require(["klt/dash/provider/Manager",
    "klt/core/when",
    "klt/dash/provider/JsonRest/JsonRestProvider"],
function (Manager, when, JsonRest) {
    var provider = Manager.getInstance();
    //Delete dashlet module by id
    when(provider.deleteDashletModule("70").then(
        function () {
            //logic on success
        },
        function () {
            //logic on fail
        });
});
});

```

DASHLET DEVELOPMENT

In this section, we will be covering of dashlet development model and common scenarios.

HELLO WORLD DASHLET

SETUP CLIENT ENVIRONMENT

Since DD Dashlet modules use AMD format, we need to use an AMD compliant loader to load modules dynamically. Although you may prefer to use other AMD loaders ([Curl](#), [RequireJs](#) or [bdLoad](#)), we recommend you to use Dojo AMD Loader. More information about Dojo AMD Loader can be found at <http://livedocs.dojotoolkit.org/loader/amd>.

To setup a development environment;

1. Create a web application using your javascript IDE.
2. Create a folder named Scripts inside your web application.
3. Copy DD files inside Scripts folder.
4. Add following script tag.

```
<script src="/scripts/dojo.js"
data-doj-config="async: true, packages:['klt','dashlets']"></script>
```

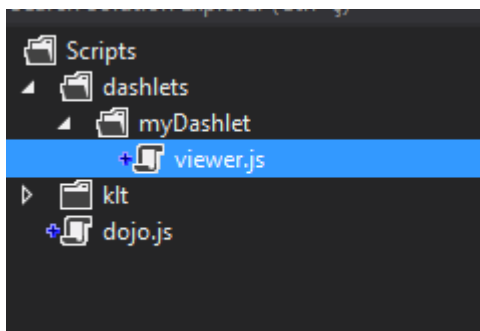
SETUP SERVER ENVIRONMENT

[TODO]

DEVELOP YOUR DASHLET

Inside Scripts folder, create a folder named *dashlets*. Inside dashlets folder, create a folder named *myDashlet*. This folder will contain all files related with your dashlet.

Add a javascript file named viewer.js inside myDashlet. The folder structure should be like below.



Now you are ready to implement your dashlet module. Copy the following code snippet inside viewer.js file.

```
define(function () {
    return function () {
        this.domNode = document.createTextNode("Hello world!");
    }
});
```

REGISTER YOUR DASHLET

To register your dashlet module use client provider.

```
require(["klt/dash/model/DashletModuleModel",
```

```

        "klt/dash/provider/Manager",
        "klt/core/when", "klt/dash/provider/JsonRest/JsonRestProvider"],
function (DashletModuleModel, Manager, when) {
    // Create a DashletModuleModel instance.
    var module = new DashletModuleModel({
        path: "dashlets/myDashlet/viewer",
        title: "Hello World!"
    });

    // Use createDashletModule method of default client provider.
    when(Manager.getInstance().createDashletModule(module),
        function (result) {
            console.info("Created module with id" + result.id);
        },
        function (err) {
            console.error(err);
        });
});

```

Now, DD knows that there is a dashlet module which has its module definition inside `dashlet/myDashlet/viewer` and able to dynamically load your dashlet.

CREATING A TEST DASHBOARD

More information about creating a new dashboard can be found at [Creating new dashboard](#) section.

ADDING DASHLET INSTANCES TO DASHBOARD

`DashboardView.createDashlet` method is used to create a dashlet instance and add it to the dashboard.

```
createDashlet: function (model) { }
```

`createDashlet` method takes a `DashletModel` object and uses this object to instantiate the dashlet instance and put it inside dashboard. Before adding a dashlet to the dashboard, a dashboard instance should be loaded using `DashboardView.load` method.

```

require(["klt/dash/ui/DashboardView",
        "klt/dash/model/DashletModel",
        "klt/dash/core/designMode",
        "klt/core/when",
        "klt/domReady!"],
function (DashboardView, DashletModel, designMode, when) {

    // set designMode to designMode.dashboard or designMode.full
    var view = new DashboardView({designMode: designMode.full}, "node");
    view.startup();

    // load dashboard with id 206
    when(view.load("206"),
        function () {
            // as an alternative to uniquely identify a module you can use

```

```

// getDashletModuleByPath (use module path)
// getDashletModule (use id)
var query = view.provider.getDashletModuleByTitle("Rss Reader");

when(query, function(module) {
    var instance = new DashletModel({ module: module });
    when(view.createDashlet(instance),
        function () {
            console.info("Instance added to dashboard");
        },
        function (err) {
            console.error(err);
        });
    }, function(err) {
        console.error(err);
    })
}, function (err) {
    console.error(err);
});
})

```

DEVELOPING DASHLETS

Previous section demonstrated development of a simple dashlet. Although for basic scenarios that will be enough to have a module file and return a function, complex scenarios will generally require detailed solutions.

DASHLET DOM STRUCTURE AND LIFECYCLE

When a new dashlet is about to be created by DD, following steps are performed.

1. Dashlet module file is loaded by AMD loader and return value is retrieved. An AMD formatted dashlet module should return a function as a result and return value of the module should be the constructor function.
2. A new dashlet instance is created passing initialization parameters to that function.
3. `domNode` property of dashlet instance is examined and if exists it's placed to the body node of dashlet pane.
4. After dom hierarchy is created and dashlet is displayed to the user, `startup` method is examined. If dashlet instance defines a function named `startup`, it's called within the context of dashlet instance.

When user wants to remove the dashlet instance from dashboard, DD looks for a function named `destroyRecursive`. If you need to do some type of garbage collection for your dashlet instance, this is the best place.

Below code snippet shows `startup` and `destroyRecursive` functions for Hello World! dashlet.

```

define(function () {
    return function () {
        this.domNode = document.createTextNode("Hello world!");
    };
});

```



```

    this.startup = function () {
        console.info("Dashlet is started");
    },
    this.destroyRecursive = function () {
        console.info("Dashlet is destroyed");
    }
}
});

```

WORKING WITH DASHLETCONTEXT

When a new dashlet instance is about to be created by DD, initialization parameters including context is passed to the constructor of your dashlet.

`context` is an instance of `DashletContext` class and is the bridge between your dashlet and DD framework. It contains references to the dashboard view and pane object, also has helper methods and properties.

```

define(function() {
    return function (params) {
        for (var key in params) {
            // Assign initialization parameters to the dashlet instance
            this[key] = params[key];
            console.info("Assigned " + key + " as " + this[key]);
        }

        this.domNode = document.createTextNode("Hello World!");
        this.startup = function () {
            // Get title property from context.
            console.info("Started dashlet " + this.context.title);
        }
    }
});

```

Above code snippet illustrates assigning initialization parameter values to the dashlet object and getting title property of dashlet from context object.

Output is

```

Assigned context as [object Object]
Started dashlet Test Dashlet

```

Although you can directly assign values to properties of context, we recommend you to use `set` function of `Stateful` class since `DashletContext` class inherits from `Stateful`. Any side effects will be handled by this function.

```

define(function() {
    return function (params) {
        for (var key in params) {
            this[key] = params[key];
            console.info("Assigned " + key + " as " + this[key]);
        }
    }
});

```

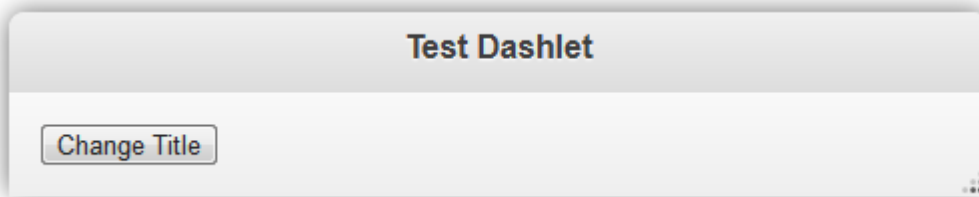
```

// Create a button
this.domNode = document.createElement("button");
this.domNode.innerHTML = "Change Title";

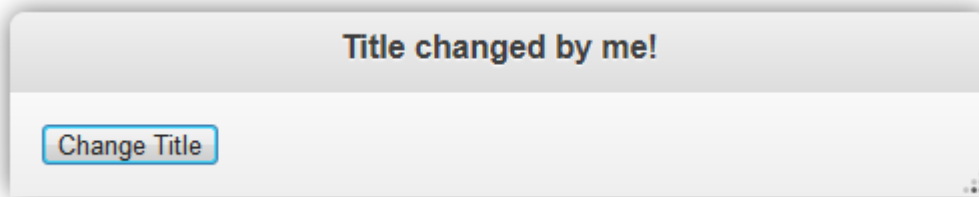
var self = this;
this.domNode.onclick = function () {
    var oldTitle = self.context.title;

    // use set function to change a property value
    self.context.set("title", "Title changed by me!");
    console.info("Changed title from " + oldTitle + " to " +
        self.context.title);
}
}
});











```



After clicking Change Title button, `title` property is updated but also as a side effect, title of dashlet is changed.



Below table lists key members of `DashletContext` class.

Member	Description
 <code>dashboard</code>	Reference to <code>DashboardView</code> object in which dashlet is hosted.
 <code>module</code>	Reference to the <code>DashletModuleModel</code> object.
 <code>model</code>	Reference to the <code>DashletModel</code> object.
 <code>pane</code>	Reference to the <code>DashletPane</code> object in which dashlet is located.
 <code>title</code>	title of dashlet
 <code>id</code>	Unique id of dashlet. Note this is the unique identifier used by client provider, not the dom id.
 <code>paneConfig</code>	Reference to the <code>ConfigModel</code> object which holds pane configuration values.
 <code>config</code>	Reference to the <code>ConfigModel</code> object which holds dashlet configuration values.
 <code>publish</code>	Publishes a topic. Wrapper for <code>messageBus.publish</code> .
 <code>subscribe</code>	Uses <code>messageBus.Subscribe</code> function to subscribe to a topic. When

	context is destroyed (user removed the dashlet or dashboard unloaded) automatically removes the subscription.
F <i>subscribeToDashlet</i>	Prepends <i>klt/dash/dashlet/</i> to the topic parameter and immediately calls subscribe. Instead of using <code>subscribe("klt/dash/dashlet/visualStateChanging")</code> you can use <code>subscribeToDashlet("visualStateChanging")</code>
F <i>subscribeToEditor</i>	Prepends <i>klt/dash/dashlet/editor/</i> to the topic parameter and immediately calls subscribe.
F <i>getProvider</i>	Returns the active provider instance dashboard is using.
F <i>save</i>	Saves dashlet model using provider. Same as calling <code>this.context.getProvider.saveDashlet(this.context.model);</code>
F <i>remove</i>	Removes the dashlet from dashboard and deletes the instance using provider.
F <i>addPaneCommand</i>	Adds a command to the pane. See XXX for more information about pane commands.

WORKING WITH DASHLET EDITORS

Optionally but generally dashlets need configuration. For an RSS dashlet, you may want to get RSS Url from user. For a chart dashlet, you will probably want your users to customize the appearance of your charts.

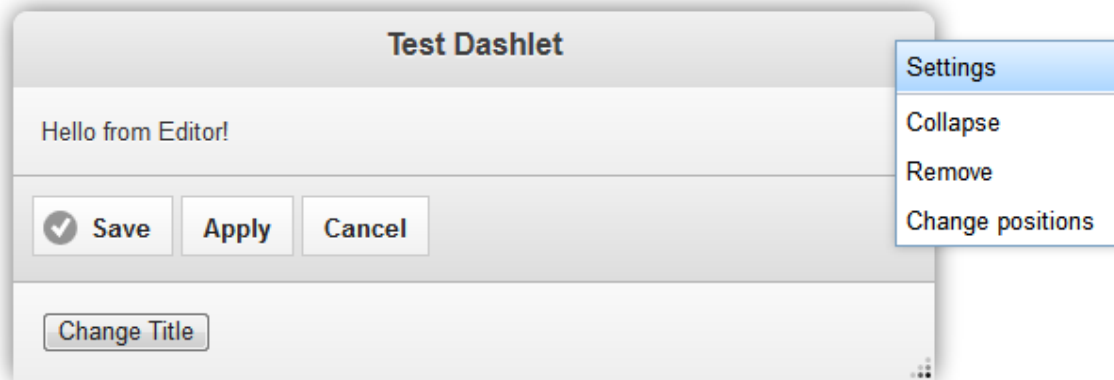
DD provides you to set an editor for your dashlet. When user wants to configure your dashlet, editor is automatically created and displayed to the user. First step to have an editor for a dashlet is adding a configuration value to dashlet module which defines the editor path.

```
require(["klt/dash/model/DashletModuleModel",
        "klt/dash/provider/Manager",
        "klt/core/when", "klt/dash/provider/JsonRest/JsonRestProvider"],
function (DashletModuleModel, Manager, when) {
    // Create a DashletModuleModel instance.
    var module = new DashletModuleModel({
        path: "dashlets/myDashlet/viewer",
        title: "Hello World!",
        // Add a configuration object including editor path
        config: {
            editor: { path: "dashlets/myDashlet/editor" }
        }
    });

    // Use createDashletModule method of default client provider.
    when(Manager.getInstance().createDashletModule(module),
        function (result) {
            console.info("Created module");
        },
        function (err) {
            console.error(err);
        }
    );
});
```

Same rules for dashlets are also valid for dashlet editors. They are AMD formatted modules and loaded by DD. Below is a sample editor.js file content.

```
define(function () {  
    return function (params) {  
        this.domNode = document.createTextNode("Hello from Editor!")  
    }  
});
```



When a new dashlet editor instance is about to be created by DD, initialization parameters including context is passed to the constructor of your dashlet.

`context` is an instance of `DashletEditorContext` class and is the bridge between your dashlet editor and DD framework.

```
define(function () {  
    return function (params) {  
        for (var key in params) {  
            // Assign initialization parameters to the dashlet editor instance  
            this[key] = params[key];  
            console.info("Assigned " + key + " as " + this[key]);  
        }  
  
        this.domNode = document.createTextNode("Hello from Editor!");  
  
        this.startup = function () {  
            console.info("Started editor for " +  
                this.context.dashletContext.title);  
        }  
    }  
});
```

Output is

```
Assigned context as [object Object]  
Started editor for Test Dashlet
```

Below table lists key members of `DashletEditorContext` class.

Member	Description
<i>dashletContext</i>	Reference to the context of dashlet.
<i>config</i>	Reference to the <code>config</code> object of dashlet context. Shortcut for <code>dashletContext.config</code>
<i>editView</i>	Reference to the <code>DashletEditView</code> object. This is the container object for dashlet editor.

MANAGING DASHLET CONFIGURATION

DD lets you to manage configuration values for your dashlets. You can set a configuration value and retrieve it later.

`DashletContext` and `DashletEditorContext` objects both have a property named `config`, which refers to a `ConfigModel` object. Since `ConfigModel` class inherits from `klt/core/Stateful`, you can use this property to set, get and watch property changes.

Below code snippet shows source code of sample editor module.

```
// editor.js
define(function () {
  return function (params) {

    // set context
    this.context = params.context;

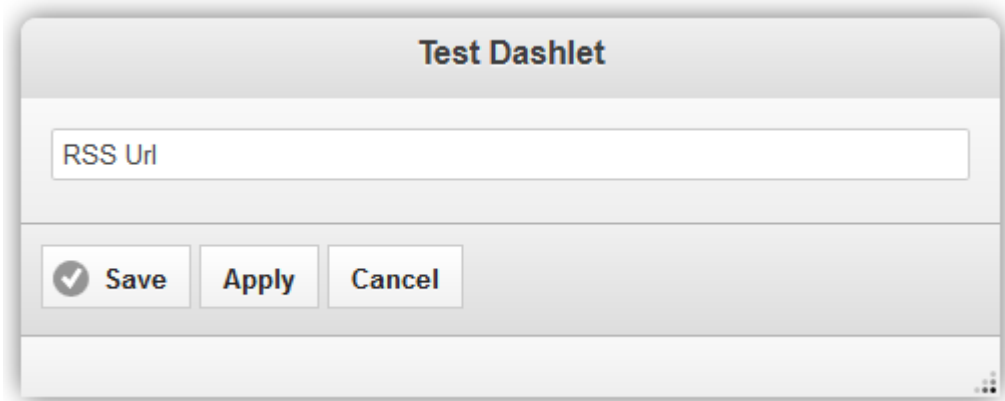
    // create dom elements
    var form = document.createElement("form");
    var row = document.createElement("p");
    this.rssUrlTextbox = document.createElement("input");
    this.rssUrlTextbox.type = "text";
    this.rssUrlTextbox.placeholder = "RSS Url";
    row.appendChild(this.rssUrlTextbox);
    form.appendChild(row);
    this.domNode = form;

    // get rssUrl configuration value from config object.
    // If there exists no configuration value return ""
    this.rssUrlTextbox.value = this.context.config.getDefault("rssUrl", "");

    var self = this;

    // subscribe to editor validating event.
    this.context.subscribe("klt/dash/dashlet/editor/validating",
    function (event) {
      // this refers to self.context in this scope. Check if the
      // context sending validation topic is my context.
      if (event.sender == this) {
        this.config.set("rssUrl", self.rssUrlTextbox.value);
      }
    })
  }
});
```

```
});  
});
```



```
// viewer.js  
define(function () {  
  return function (params) {  
  
    // set context  
    this.context = params.context;  
    var self = this;  
    this.domNode = document.createElement("span");  
  
    this.setRssUrl = function (value) {  
      this.domNode.innerHTML = value;  
    }  
  
    // watch for changes  
    this.context.config.watch("rssUrl", function (name, oldVal, newVal) {  
      self.setRssUrl(newVal);  
    });  
  
    this.startup = function () {  
      // get rssUrl value  
      var rssUrl = this.context.config.get("rssUrl");  
  
      // if user didn't set a value for rssUrl  
      // open editor automatically  
      if (!rssUrl)  
        return this.context.openEditor();  
      else this.setRssUrl(rssUrl);  
    }  
  }  
});
```

Above code snippet illustrates getting *rssUrl* configuration value and watching for changes.

When a configuration value is changed, DD looks up for a *set* function on your dashlet. If there is a *set* function it's called automatically with configuration key and value parameters.

Same feature, instead of watching `context.config`, can be implemented as below.

```
...  
  
this.set = function (name, value) {  
  if (name == "rssUrl")  
    this.setRssUrl(value);  
}  
...
```

SETTING INITIAL CONFIGURATION

An initial configuration may be set before a dashlet instance is created. You can set initial configuration

- Manually, passing to the `DashletModel` constructor
- Automatic, using dashlet modules

USING DASHLETMODEL CONSTRUCTOR

Below code snippet shows assigning default configuration to `DashletModel` instance.

```
var instance = new DashletModel({  
  module: module,  
  config: {  
    rssUrl: "http://rss.cnn.com/rss/edition.rss" }  
});
```

USING DASHLET MODULES

Another way to set initial dashlet configuration is using dashlet modules and using `DashletModuleModel.dashletConfig` property.

```
var pathToRssModule = "klt/dash/dashlet/RssReader/RssReader";  
  
var cnnRss = new DashletModuleModel({  
  path: pathToRssModule,  
  title: "CNN Rss",  
  dashletConfig: {  
    rssUrl: "http://rss.cnn.com/rss/edition.rss"  
  },  
  metaData: {  
    description: "Rss data from Cnn"  
  }  
});  
  
var googleRss = new DashletModuleModel({  
  path: pathToRssModule,  
  title: "Google Code Rss",  
  dashletConfig: {  
    rssUrl: "http://feeds.feedburner.com/GDBcode?format=xml"  
  }  
});
```

```

    },
    metaData: {
        description: "Rss data from google code blog"
    }
});

```

As above code snippet shows, two dashlet modules use the same AMD formatted module file but have different configurations. This can be very helpful if you want to provide your users pre-configured dashlets.

ADVANCED TOPICS

This section focuses on advanced topics about dashlet development and customization techniques.

USING JAVASCRIPT TEMPLATE ENGINES

Previous section demonstrated developing dashlets, dashlet lifecycle and setting configuration values. Although creating dom structure using `document.createElement` or other helper functions may be a solution, JavaScript templates will provide a better separation of user interface and data.

Templating is a good solution in a few scenarios:

- Loading all data from the server especially in rich list displays
- Adding or updating new items in lists
- Anywhere you need to add new complex content to the page
- Anything that requires client side HTML rendering

Rest of this section will show basic principles you can use with different JavaScript frameworks. DD has no dependency to a particular templating engine and does not force you to use one of them.

USING DIJIT WIDGETS

Dijit is a widget system layered on top of Dojo. You can use `dijit._WidgetBase` and `dijit._TemplatedMixin` classes to easily develop you dashlets.

DECLARING WIDGET AND LOADING TEMPLATE

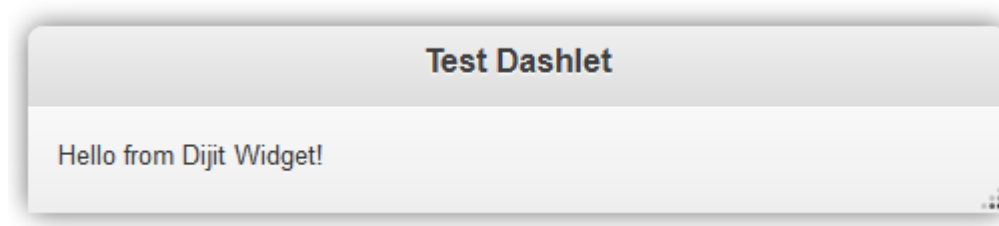
First step to create a *templated dijit widget* is defining the widget inside an AMD module.

```

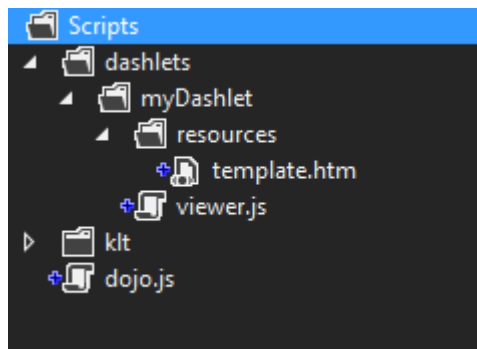
// viewer.js
define(["dijit/_WidgetBase",
    "dijit/_TemplatedMixin",
    "klt/core/declare"],
    function (_WidgetBase, _TemplatedMixin, declare) {
        return declare([_WidgetBase, _TemplatedMixin], {
            templateString: "<div><p>Hello from Dijit Widget!</p></div>"
        });
    });

```

And the output is;



Instead of setting `templateString` property inside JavaScript file, you can also load it using dojo text plugin from an external file.



Assume template file is located inside resources folder; you can load it as below.

```
// viewer.js
define(["dijit/_WidgetBase",
        "dijit/_TemplatedMixin",
        "klt/core/declare",
        "dojo/text!./resources/template.htm"],
function (_WidgetBase, _TemplatedMixin, declare, template) {
    return declare([_WidgetBase, _TemplatedMixin], {
        templateString: template
    });
});
```

Below is a simple template file content. Please note a template should have only one root node.

```
<div>
  <p>Hello from Dijit Widget!</p>
</div>
```

REFERENCING TO DOM NODES

You can use `domNode` property inside your widget to reference to the root dom node of template. Since DD also uses `domNode` property to place your dashlet to a pane, dijit widgets automatically work without additional coding.

You can use `data-dojo-attach-point` attribute to reference a specific dom node inside your template.

```
<div>
  <p>Hello from Dijit Widget!</p>
```

```

<ul data-dojo-attach-point="listNode"></ul>
<span>Loaded from </span>
<span data-dojo-attach-point="urlNode">
</span>
</div>

```

Below is a sample usage of referencing template node named `listNode`.

```

...
renderFeeds: function (feeds) {
    // delete child nodes
    while (this.listNode.firstChild)
        this.listNode.removeChild(this.listNode.firstChild);
    // for each rss feed entry create a node inside listNode
    for (var i = 0; i < feeds.feed.entries.length; i++) {
        var entry = feeds.feed.entries[i];
        var li = document.createElement("li");
        li.appendChild(document.createTextNode(entry.title));
        this.listNode.appendChild(li);
    }
}

```

MONITORING CONFIGURATION CHANGES

Dijit widgets are similar to stateful objects. You can use `get` and `set` functions to get and set a property value.

As explained before, when DD is about to create a dashlet instance, configuration settings are mixed with dashlet context and passed as an argument to the constructor function. If you are using dijit widgets, these property values are automatically mixed up with the dashlet instance.

Before mixing, dijit calls `set` function implemented inside `_WidgetBase` class. This function looks up a function on widget instance in the form of `_setXxxAttr`, where `Xxx` is the name of the property. For example if a dashlet has a configuration value named `rssUrl`, dijit looks up for a function `_setRssUrlAttr`. If found, it's called instead of directly assigning the value to the dashlet instance.

```

// called automatically during instance creation and
// after calling .set("rssUrl", "somevalue")
_setRssUrlAttr: function (value) {
    // if instance started, load feeds
    if (this._started) {
        this.loadFeeds(value);
    }
}

```

FINALIZING RSS DASHLET

Now we are ready to implement a complete Rss dashlet.

```

// viewer.js
define(["dijit/_WidgetBase",

```

```

        "dijit/_TemplatedMixin",
        "klt/core/declare",
        "dojo/text!./resources/template.htm"],

function (_WidgetBase, _TemplatedMixin, declare, template) {
    return declare([_WidgetBase, _TemplatedMixin], {
        templateString: template,

        renderFeeds: function (feeds) {
            // ...
        },

        _setRssUrlAttr: function (value) {
            // ...
        },

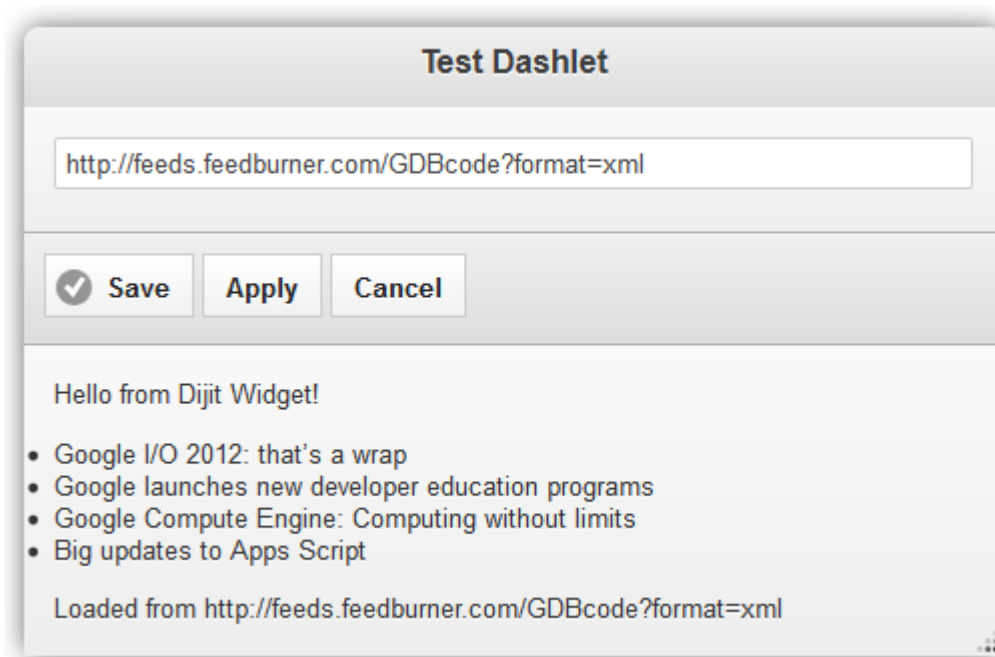
        loadFeeds: function (feedUrl) {
            var api="https://ajax.googleapis.com/ajax/services/feed/load?" +
                "v=1.0&callback=?&q=" +
                encodeURIComponent(feedUrl) + "&output=json_xml"
            var self = this;

            // Display busy indicator
            self.context.setBusy("Retreiving feeds ...");

            $.getJSON(api, function (data) {
                self.context.clearBusy();
                if (data.responseStatus == 200) {
                    self.urlNode.innerHTML = feedUrl;
                    self.renderFeeds(data.responseData);
                } else {
                    console.error("Error loading feeds using " + feedUrl);
                }
            });
        },

        startup: function () {
            this.inherited(arguments);
            var rssUrl = this.context.config.get("rssUrl");
            if (!rssUrl)
                return this.context.openEditor();
            else this.loadFeeds(rssUrl);
        }
    });
});

```



More information about dijit can be found at;

- <http://livedocs.dojotoolkit.org/dijit/index>.
- <http://livedocs.dojotoolkit.org/quickstart/writingWidgets>

USING BACKBONE VIEWS

As an alternative you can use Backbone views to create dashlets. Below is an example of a dashlet using Backbone views.

```
define(["require"], function ( require) {
    return function (params) {

        //Create an initial dom node
        this.domNode = document.createElement("div");
        //Define a template
        var template = "<h1>" +
            "<%= title %>" +
            "</h1>" +
            "<br />" +
            "<i><%= content %></i>";

        //Get resource paths
        var backboneScriptPath = require.toUrl("./Backbone.js");
        var underscoreScriptPath = require.toUrl("./Underscore.js");

        var self = this;

        //Load resources and create backbone view
        //Backbone needs underscore.js so load underscore.js first
        require([underscoreScriptPath], function () {
```

```

require([backboneScriptPath], function () {

    //Define a backbone model
    var simpleModel = Backbone.Model.extend({
        defaults: {
            title: 'Backbone Dashlet',
            content: 'I am a backbone view'
        }
    });

    //Define a backbone view
    var BackboneView = Backbone.View.extend({

        //Set dashlet dom node as "el" object
        el: self.domNode,
        //Create instance of model
        model: new simpleModel(),
        //Set template source as an underscore template
        template: _.template(template),
        //Init view
        initialize: function () {
            _.bindAll(this, 'render');
            this.render();
        },
        //Render view
        render: function () {
            this.el.innerHTML = (this.template(this.model.toJSON()));
        }
    });

    //Create backbone view instance
    var myView = new BackboneView();
});
}
});

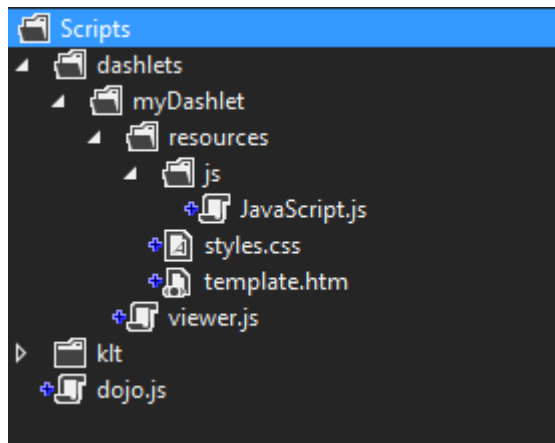
```

LOADING RESOURCES ON DEMAND

One of the common requirements while you develop your widgets is using external resources, i.e. Style sheet files and JavaScript files inside your dashlets.

A solution maybe to add all resources to the head section of HTML page. Although this works as expected, loading resources dynamically when an instance of a dashlet created can be a better solution. You can load a resource when user adds your dashlet to the dashboard and unload it when user removes your dashlet.

Assume your dashlet uses *resources/styles.css* and *resources/js/JavaScript.js* files.



LOADING CSS FILES DYNAMICALLY

`DashletContext.loadResource` and `DashletContext.unloadResource` functions can be used to load external css files.

```
define(["require"], function (require) {
    return function (params) {
        this.context = params.context;
        this.domNode = document.createTextNode("Hello world!");

        this.startup = function () {

            // use require.toUrl function to get file path
            this.cssFile = require.toUrl("./resources/styles.css");
            this.context.loadResource(this.cssFile, "css");
        },
        this.destroyRecursive = function () {

            // unload resource when user removes dashlet instance
            this.context.unloadResource(this.cssFile, "css");
        }
    };
});
```

When startup function is called, style sheet file is loaded dynamically and when `destroyRecursive` is called resource is unloaded.

```
loadResource: function (href, type, callback) {
},

unloadResource: function (href, type) {
}
```

You can optionally set a callback function when resource is loaded.

The magic behind `loadResource` and `unloadResource` functions is that, they both count how many times they are called for a specific resource file. When user adds a second instance of a dashlet to the dashboard,

`loadResource` increments the counter of the requested resource. If an `unloadResource` is called for that resource file, DD simply ignores and does not unload the resource because it assumes there is a dependency to that resource.

LOADING JAVASCRIPT FILES DYNAMICALLY

Although you can use `loadResource` and `unloadResource` functions to load / unload external JavaScript files, we recommend you using `require`.

If the JavaScript file is AMD formatted, you can load it as usual.

```
// resources/js/JavaScript.js, AMD formatted
define(function () {
    return function () {
        var res = 1;
        for (var i = 0; i < arguments.length; i++) {
            res = res * arguments[i];
        }
        return res;
    }
});
```

Module can be loaded dynamically using `require` as below.

```
define(["require"], function (require) {
    return function (params) {
        this.context = params.context;
        this.domNode = document.createTextNode("Hello world!");

        this.startup = function () {
            require(["./resources/js/JavaScript"], function (multiply) {
                console.info(multiply(12, 5, 3));
            });
        }
    }
});
```

Note that, when `startup` function completes, module may not be loaded. Second alternative is setting a dependency to the module as below.

```
define(["./resources/js/JavaScript"], function (multiply) {
    return function (params) {
        this.context = params.context;
        this.domNode = document.createTextNode("Hello world!");

        this.startup = function () {
            console.info(multiply(12, 5, 3));
        }
    }
});
```

If the requirement is load the module conditionally, you can use the first solution. If the module is a part of your dashlet and should be loaded always, second way is the best.

AMD loader can also be used to load non-AMD code by passing an identifier that is actually a path to a JavaScript file. The loader identifies these special identifiers in one of three ways:

- The identifier starts with a `“/”`
- The identifier starts with a protocol (e.g. `“http:”`, `“https:”`)
- The identifier ends with `“.js”`

When arbitrary code is loaded as a module, the module’s resolved value is undefined; you will need to directly access whatever code was defined globally by the script.

Below is a sample non AMD formatted JavaScript file.

```
var MySimpleMath = (function () {
    return {
        multiply: function () {
            var res = 1;
            for (var i = 0; i < arguments.length; i++) {
                res = res * arguments[i];
            }
            return res;
        }
    }
})();
```

You can load the file as

























```
define(["require"], function (require) {
    return function (params) {
        this.context = params.context;
        this.domNode = document.createTextNode("Hello world!");

        this.startup = function () {
            var path = require.toUrl("./resources/js/JavaScript.js");
            require([path], function () {
                console.info(MySimpleMath.multiply(12, 5, 3));
            })
        }
    }
});
```

CUSTOMIZING DASHLET PANE

Every dashlet is placed inside a pane in a dashboard. Pane is responsible to hold the dashlet instance and provide a container to the dashlet. You can use `context.pane` property inside your dashlet to get a reference to the pane object.

`DashletPaneBase` class is the base class for panes and inherits from `dijit._widgetBase`, which means it has all functionality of dijit widgets. Below table lists key members of `DashletPaneBase` class.

Member	Description
 <code>iconClass</code>	Specifies the css class name of pane icon.
 <code>iconBoxHidden</code>	Specifies if the iconbox of pane is hidden.
 <code>iconBusyClass</code>	Specifies the css class to be set for the icon when dashlet is busy. Defaults to <i>ui-icon-busy</i>
 <code>iconBoxCommandName</code>	
 <code>bodyBusyClass</code>	Specifies the css class to be set for the pane body when dashlet is busy.
 <code>autoHideHeader</code>	If set true, specifies if the header of dashlet is only displayed when mouse is over the pane.
 <code>noHeader</code>	If set true, specifies header node is hidden.
 <code>editState</code>	Specifies current edit state of the dashlet. If set to “edit”, dashlet editor is displayed.
 <code>visualState</code>	Specifies current visual state of the dashlet. Possible values are maximize, expand and collapse. Set to a known value to change the visual state of the dashlet.
 <code>readonly</code>	Specifies if pane is read-only. A read-only dashlet is not editable nor draggable.
 <code>disableEdit</code>	Specifies if editing of dashlet is disabled. If editing is disabled, edit dashlet command is hidden automatically.
 <code>disableMaximize</code>	Specifies if maximization of dashlet is disabled. If maximization is disabled, maximize dashlet command is hidden automatically.
 <code>disableRemove</code>	Specifies if removing dashlet is disabled. If removing is disabled, remove dashlet command is hidden automatically.
 <code>disableMove</code>	<i>Specifies if moving of dashlet is disabled.</i>
 <code>disableCollapse</code>	Specifies if collapsing of dashlet is disabled. If collapsing is disabled, collapse command is hidden automatically.
 <code>showHeader</code>	Shows pane header if it's hidden.
 <code>hideHeader</code>	Hides pane header.
 <code>toggleCollapse</code>	Toogles collapse.
 <code>getBodySize</code>	Returns an object which has <i>w</i> and <i>h</i> properties for width and hight of body node.
 <code>getBodyNode</code>	Returns domNode object for pane body container.
 <code>getHeaderNode</code>	Returns domNode object for pane header container.
 <code>getTitleContainerNode</code>	Returns domNode object for pane title container.
 <code>getTitleNode</code>	Returns domNode object for pane title.
 <code>getEditorContainerNode</code>	Returns domNode object for dashlet editor container.

SETTING PANE PROPERTIES

Pane properties can be set using;

- `klt.dash.defaults.dashlet.paneConfig` global object.
- `defaults.dashlet.paneConfig` property of `DashboardView` object.
- `DashletModuleModel.paneConfig` property.
- `DashletContext.paneConfig` property.

When a new pane is about to be created, DD starts with an empty configuration object (`{}`). Then it looks for the global object `klt.dash.defaults.dashlet.paneConfig` and if exists mixes it to the configuration object. After, DD looks `defaults.dashlet.paneConfig` property on dashboard object and if exists mixes it to the configuration object. Next, `paneConfig` property of dashlet module is checked and if exists mixes its value too. Finally `paneConfig` value of dashlet instance is mixed, context is set and pane object is created.

This lets you to globally set default pane properties, able to customize them on dashboard, module and finally dashlet level.

```
klt = klt || {};  
klt.dash = klt.dash || {};  
klt.dash.defaults = klt.dash.defaults || {};  
klt.dash.defaults.dashlet = klt.dash.defaults.dashlet || {};  
klt.dash.defaults.dashlet.paneConfig = {  
    iconBoxHidden: true  
}
```

Above code snippet sets a default value for `iconBoxHidden` property of dashlet pane. This lets you to globally hide icon box node of your dashlets automatically. Please note default values are only applied before pane instance creation.

```
require(["klt/dash/provider/Manager",  
        "klt/dash/module/DashletModuleModel",  
        "klt/core/when",  
        "klt/dash/provider/JsonRest/JsonRestProvider"  
],  
function (Manager, DashletModuleModel, when) {  
    var provider = Manager.getInstance();  
  
    // get DashboardModuleModel object  
    var moduleQuery = provider.getDashletModuleByTitle("Rss Reader");  
    when(moduleQuery,  
        function (module) {  
  
            // set paneConfig  
            module.paneConfig.set({  
                iconClass: "ui-icon-alert",  
                disableMaximize: true  
            });  
  
            // save using provider  
            when(provider.saveDashletModule(module),  
                function () {  
  
                },  
                function (err) {  
                    console.error(err);  
                });  
        },  
        function (err) {  
            console.error(err);  
        });  
    },  
    function (err) {  
        console.error(err);  
    });  
});
```

```
});
});
```

Above code snippet shows how to set default pane configuration using `DashletModuleModel.paneConfig` property. New instances of Rss Reader will have default pane config properties of *iconClass* and *disableMaximize*.

After a dashlet is created it is also possible to set pane properties for that dashlet instance only using `DashletContext.paneConfig`. Below code shows a sample test dashlet module.

```
define(function () {
    return function (params) {
        this.context = params.context;

        var self = this;

        this.domNode = document.createElement("button");
        this.domNode.innerHTML = "Test";
        this.domNode.onclick = function () {

            // set config
            self.context.paneConfig.set("iconClass", "ui-icon-alert");

            // save using client provider
            self.context.save();

        }
    }
});
```

WORKING WITH PANE COMMANDS

ABOUT COMMAND OBJECTS

A command is either a button or menu item displayed on pane. DD allows developers to set initial commands for their dashlets or manage existing commands.

Command object is a simple JavaScript object which has the following key properties.

Method Name	Description
<i>name</i>	Specifies name of the command. This is generally a string value which uniquely identifies the command.
<i>label</i>	Specifies label of command.
<i>type</i>	Specifies type of command. A value of <i>builtin</i> , <i>visualState</i> and <i>editState</i> is automatically handled by DD.
<i>location</i>	Specifies user interface location of command. If set to <i>menu</i> , command is created inside dashlet menu. If set to <i>header</i> , command is displayed on header of pane.
<i>float</i>	If location is set to <i>header</i> , specifies if command is created on left or right side of header. A value of <i>l</i> means left, <i>r</i> means right.
<i>ui</i>	Specifies an object which is applied to the user control of the control.

Below is an example of a sample command.

```
{
  name: "testCommand",
  label: "Hello World!",
  type: "custom",
  location: "header", float: "r",
  ui: {
    iconClass: "d-icon-arrow-u",
    title: "Click to see the result"
  }
}
```

CREATING COMMAND OBJECTS

Although you can directly create a javascript object for a command, using create method of `klt/dash/core/DashletCommand` singleton is recommended. Create method checks the name of your command with the existing commands and assigns default properties for the new command.

```
require(["klt/dash/core/DashletCommand"], function (DashletCommand) {
  var myCommand = DashletCommand.create("myCommand",
    {
      label: "Hello World!",
      type: "custom",
      ui: {
        iconClass: "d-icon-refresh",
        hideLabel: false
      }
    }
  );
});
```

You can use `klt/dash/core/CommandName` singleton to get built in command name constants.

```
{
  none: 'none',
  maximize: 'maximize',
  collapse: 'collapse',
  restore: 'restore',
  expand: 'expand',
  drop: 'drop',
  remove: 'remove',
  share: 'share',
  about: 'about',
  edit: 'edit',
  menu: 'menu',
  refresh: 'refresh',
  visualState: 'visualState'
}
```

Use above command constants to get a direct reference to known commands used by DD.

```
require(["klt/dash/core/DashletCommand"], function (DashletCommand) {
  var removeCommand = DashletCommand.remove;
});
```

SETTING INITIAL PANE COMMANDS

Before DD prepares initial commands for a dashlet, `klt/dash/dashlet/commands/prepare` topic is published.

You can use `event.args.commands` value to get a list of commands created by default for dashlet and modify this list.

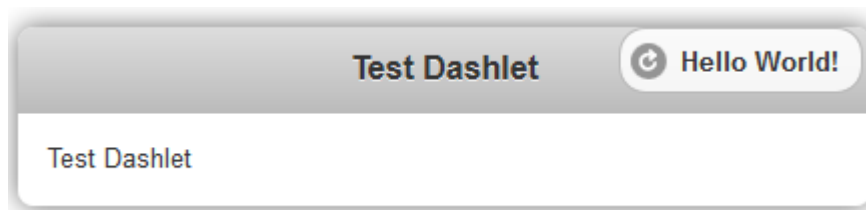
Below is a sample dashlet demonstrating how to set initial commands.

```
define(["require",
        "klt/dash/core/DashletCommand",
        "klt/dash/core/CommandName"],
function (require, DashletCommand, CommandName) {
    return function (params) {
        this.context = params.context;
        var self = this;

        this.domNode = document.createElement("div");
        this.domNode.innerHTML = "Test Dashlet";

        this.context.subscribe("klt/dash/dashlet/command/prepare",
        function (event) {
            if (event.sender == this) {
                var commands = event.args.commands;
                // reset commands array
                commands.length = 0;
                var removeCommand= DashletCommand.create(CommandName.remove);
                removeCommand.ui.hideLabel = false;
                var myCommand = DashletCommand.create("myCommand",
                {
                    label: "Hello World!",
                    type: "custom",
                    ui: {
                        iconClass: "d-icon-refresh",
                        hideLabel: false,
                        corners:"all"
                    }
                });

                commands.push(removeCommand);
                commands.push(myCommand);
            }
        });
    };
});
```



MANAGING COMMANDS

Use *DashletContext.getPaneCommand* method to get a reference to an existing command.

Below code snippet shows how to get a command and a reference to the user interface control object.

```
this.startup = function () {  
    var cmd = this.context.getPaneCommand("myCommand");  
    cmd.uiControl.set("label", "Hi again");  
}
```

You can use *DashletContext.removePaneCommand* to remove a command, *DashletContext.paneCommands* to get a list of commands.

RESPONDING TO COMMAND EVENTS

When a command (built in or custom) is about to be executed *klt/dash/dashlet/command/executing* topic is published. You can subscribe this topic and if necessary cancel the execution of command.

```
this.context.subscribe("klt/dash/dashlet/command/executing",  
function (event) {  
    if (event.sender == this) {  
        var cmd = event.args.command;  
        if (cmd.name == "maximize")  
            event.cancel = true;  
    }  
});
```

After a command is executed successfully, *klt/dash/dashlet/command/executed* topic is published. You can use *event.args.command* to get a reference to the executed command object and *event.args.result* to the result of execution.

WORKING WITH LAYOUTS

ABOUT LAYOUTS AND DASHLET POSITION

Every dashboard has a layout object which organizes dashboard layout and positions of dashlets. A layout should have at least one section and a section should have at least one zone.

Section can be considered as row of basic html table. It contains zones which can be considered as column of basic html table. Please note dashlets are placed inside zones and layouts are responsible for updating positions of dashlets.

USING READY TO USE LAYOUTS

Dashboard supports three different layout types. Each layout has different HTML structure.

Layout Type	Description
<i>TableLayout</i>	Uses a HTML table dom object. <i>tr</i> nodes are sections and <i>td</i> nodes are zones of the layout. You can use any table dom properties such as <i>colspan</i> or <i>rowspan</i> .
<i>RowColumnLayout</i>	Sections and zones are HTML div nodes.
<i>AbsoluteLayout</i>	Allows dashlets to move freely in dashboard. Dashlets can overlap each other and move pixel by pixel.
<i>MainLayout</i>	Like <i>TableLayout</i> but uses div tags inside of <i>tr</i> and <i>td</i> . Use both for desktop and mobile.

Here is an example:

```
require(["klt/dash/provider/Manager",
        "klt/dash/model/DashboardModel",
        "klt/core/when",
        "klt/dash/layout/TableLayout",
        "klt/dash/provider/JsonRest/JsonRestProvider"],
        function (Manager, DashboardModel, when, TableLayout) {
    //Define a table layout
    var myLayout = new TableLayout({
        //Define your sections
        sections: {
            mySection1: {
                //Define your zones
                zones: {
                    myZone1: {
                        //Style property for zone dom
                        style: {
                            "background-color": "red"
                        }
                        //Custom attributes
                        ,attr:{
                            "id":"myZone"
                        }
                    },
                    //An other zone
                    myZone2: {
                        style: {
                            "background-color": "blue"
                        }
                    }
                }
            }
        }
    })
}
```

```

    }
  });

  //Create a new dashboard model using "myLayout"
  var myDashboard = new DashboardModel({
    title: "My Dashboard",
    layout: myLayout
  });

  //Save to data source
  var myProvider = Manager.getInstance();
  when(myProvider.createDashboard(myDashboard),
    function (result) {
      //logic on success
    },
    function (error) {
      //logic on fail
    });
  });
};

```

A row-column layout can be created similar way.

```

require(["klt/dash/provider/Manager",
  "klt/dash/model/DashboardModel",
  "klt/core/when",
  "klt/dash/layout/RowColumnLayout",
  "klt/dash/provider/JsonRest/JsonRestProvider"],
function (Manager, DashboardModel, when, RowColumnLayout) {
  //Define a row-column layout
  var myLayout = new RowColumnLayout({
    sections: {
      mySection1: {
        zones: {
          myZone1: {

          }
        }
      }
    }
  });

  var myDashboard = new DashboardModel({
    title: "My Dashboard",
    layout: myLayout
  });

  //Save to data source
  var myProvider = Manager.getInstance();

```



```

        when(myProvider.createDashboard(myDashboard),
            function (result) {
                //logic on success
            },
            function (error) {
                //logic on fail
            }
        );
    });

```

Below table lists methods of layout objects to organize sections and zones.

Method Name	Description
<i>addSection</i>	Adds a section
<i>removeSection</i>	Removes a section
<i>addZone</i>	Adds a zone
<i>removeZone</i>	Removes a zone

Here is a basic usage:

```

require(["klt/core/when",
    "klt/dash/ui/DashboardView"],
    function (when, DashboardView) {
        //Initialize a dashboard view
        var viewer = new DashboardView({ id: "myViewer" }, "dashboard");
        viewer.startup();
        //Load a dashboard
        when(viewer.load("269"), function (dashboard) {
            //Get layout
            var layout = viewer.layout;
            //Add new section.
            var myNewSection = layout.addSection("myNewSection");
            //Now add a new zone
            var myNewZone = layout.addZone(myNewSection.section, "myNewZone",
                //Add custom zone properties
                {
                    style: {
                        "background-color": "yellow"
                    }
                }
            );
            //Save to your data source
            viewer.save();
        }, function (err) {
            console.error("Can't load dashboard!");
        });
    });

```

DEVELOPING NEW LAYOUTS

Developers can create their own layouts. You can use existing layout classes to extend them or use `klt/dash/layout/_Layout` class to start from scratch.

WORKING WITH THEMES

With the help of themes, you can change user interface of dashboard, layout and panes. DD also supports custom themes.

A theme is defined by

- An id value which uniquely identifies the theme.
- Path to a cascading style sheet (.css) file.
- An optional list of styles which theme supports.
- An optional default style identifier.

Below is a sample theme definition.

```
{
  css: require.resolve("../resources/themes/classic/main.css"),
  styles: {
    c: {
      title: "Default",
      color: "#E3E3E3",
    },

    b: {
      title: "Blue",
      color: "#5B92C1"
    },

    d: {
      title: "Gray",
      color: "#C8C8C8"
    },

    a: {
      title: "Black",
      color: "#242424"
    },

    e: {
      title: "Yellow",
      color: "#FBEE8E"
    }
  },
  defaultStyle: "b"
}
```

`klt/dash/ui/ThemeManager` class is responsible from managing themes.

RETRIEVING THEMES AND STYLES

`ThemeManager.getThemes` method can be used to retrieve a list of registered themes.

`ThemeManager.currentThemeId` property returns current theme id. By default, theme with id *classic* is loaded.

```
require(["klt/dash/ui/ThemeManager"],

function (ThemeManager) {
    var themes = ThemeManager.getThemes();
    for (var tid in themes) {
        var theme = themes[tid];

        // id of theme
        console.info(ThemeManager.currentThemeId == tid ?
            tid + " is current" : tid);
        console.info("Theme supports following styles");

        // get styles
        var styles = theme.styles;
        for (var sid in styles) {
            var style = styles[sid];
            console.info("style id " + sid + ", title " + style.title);
        }
    }
});
```

Below is a sample output.

```
classic is current
Theme supports following styles
style id c, title Default
style id b, title Blue
style id d, title Gray
style id a, title Black
style id e, title Yellow
```

CHANGING THEME

`ThemeManager.select` method can be used to change a theme and style.

```
select: function (/* String */ id, /*String?*/ style)
```

if `style` parameter is not specified, default style of theme (if exists) is loaded.

```
ThemeManager.select("classic", "a");
```

Above code snippet loads theme with id *classic* and style with id *a* (black).

PERSISTING SELECTED THEME AND STYLE

Two helper functions, `ThemeManager.saveToCookie` and `ThemeManager.loadFromCookie`, can be used to save / load current theme and style id to / from cookie.

RESPONDING TO THEME AND STYLE CHANGES

To be notified about current theme or style changes use `klt/core/messageBus` to subscribe `klt/dash/theme/changed` topic.

```
require(["klt/dash/ui/ThemeManager",
        "klt/core/messageBus"],
function (ThemeManager, bus) {
    bus.subscribe("klt/dash/theme/changed", function (event) {
        if (event.oldTheme) {
            console.info("Old theme was " + event.oldTheme);
        }

        if (event.newTheme) {
            console.info("New theme is " + event.newTheme);
        }

        if (event.oldStyle) {
            console.info("Old style was " + event.oldStyle);
        }

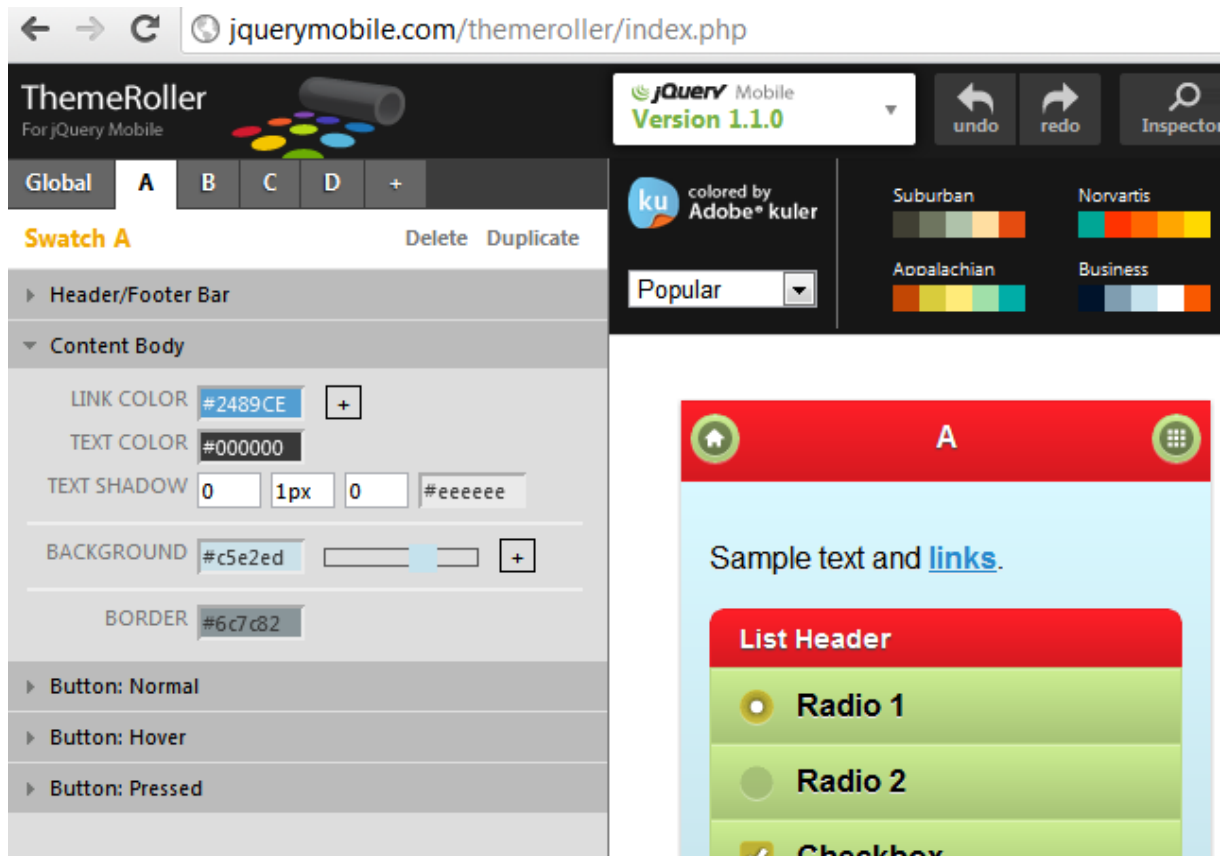
        if (event.newStyle) {
            console.info("New style is " + event.newStyle);
        }

        // Save current theme to cookie. In your app.init handler
        // you can use themeManager.loadFromCookie method
        ThemeManager.saveToCookie();
    });
});
```

DEVELOPING NEW THEMES

DD supports custom themes and by default uses same classes with JQuery Mobile UI. You can develop a theme by starting from scratch or add new styles to existing themes.

To develop a new theme use *ThemeRoller* (<http://jquerymobile.com/themeroller/index.php>) to create your theme.



After finish designing your theme, download it as a zip file.

Download Theme

Theme Name

This will generate a Zip file that contains both a compressed (for production) and uncompressed (for editing) version of the theme.

To use your theme, add it to the head of your page before the `jquery.mobile.structure` file, like this:

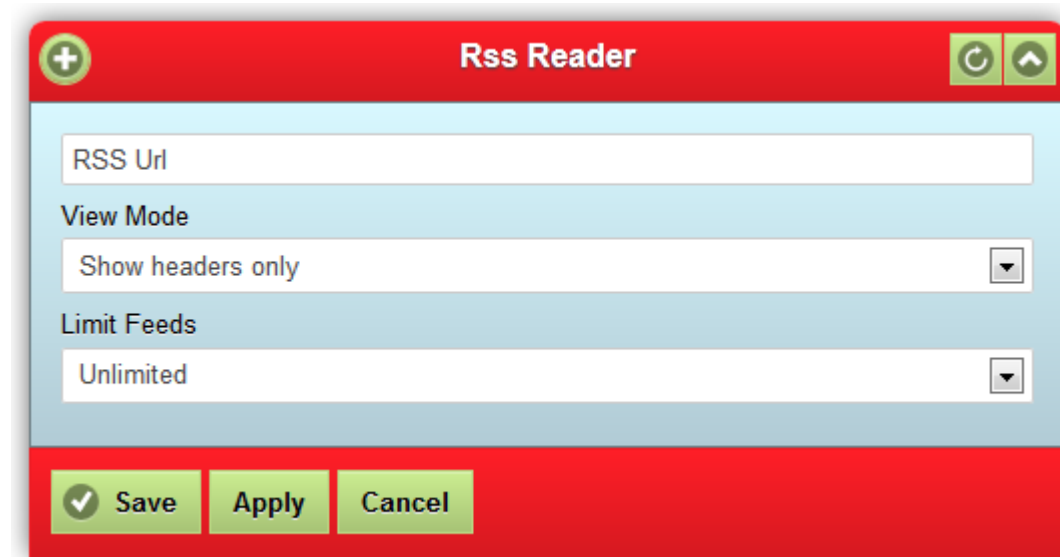
This will produce you a zip file which contains a css file and images folder. Create an empty directory in your project (i.e. *myTheme*) and copy all files and folders inside.

- As a startup point, you can use classic theme. Copy `jquery.mobile.structure-1.1.1.min` and `main.css` file inside `resource/themes/classic/` to the above directory.
- Download <http://code.jquery.com/mobile/1.1.0/jquery.mobile.structure-1.1.0.min.css> file and copy it to the same directory.
- Open `main.css` file inside your folder. Replace file reference.
 - Uncomment `/*@import url("jquery.mobile.structure-1.1.1.min.css");*/`
 - Replace `@import url("classic.min.css")` with your theme, `@import url("myTheme.min.css")`
 - Open `myTheme.min.css` and `myTheme.css`. Replace text `.ui-` with `.d-`

To register your theme use `ThemeManager.register` method.

```
require(["klt/dash/ui/ThemeManager"],
function (ThemeManager) {
    // user your path
    var pathToCss = "/Content/MyTheme/main.css";
    ThemeManager.register("myTheme",
    {
        css: pathToCss,
        styles: {
            a: {
                title: "default"
            }
        },
        defaultStyle: "a"
    });
    ThemeManager.select("myTheme");
});
```

Below is a sample screen shot with new theme.



WORKING WITH SERVER SIDE

HOW TO CONFIGURE ASP.NET MVC APPLICATION

ASP.NET MVC 3 CONFIGURATION

Dashboard requires some configuration to run over Asp.net MVC Framework. To use dashboard in your application first run metadata database script. After created dashboard database successfully, add required references to your project.

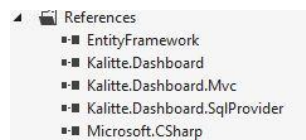
These are reference files:

Reference	Description	Dependency
<i>Kalitte.Dashboard.dll</i>	Main dashboard library	Required
<i>Kalitte.Dashboard.Mvc.dll*</i>	Asp.net MVC Framework implementation library	Required
<i>Kalitte.Dashboard.MySqlProvider.dll</i>	Data provider for MySQL databases	Optional**
<i>Kalitte.Dashboard.OracleProvider.dll</i>	Data provider for Oracle Databases	Optional**
<i>Kalitte.Dashboard.SessionProvider.dll</i>	Data provider for server session data	Optional**
<i>Kalitte.Dashboard.SqlProvider.dll</i>	Data provider for Microsoft SQL Server	Optional**

(*) There are two different versions for MVC 3 and MVC 4 frameworks

(**) At least one data provider reference required

Project references should look like this:



Add following property in “configSections” area in master “Web.config” file of application.

```
<section
type="Kalitte.Dashboard.Configuration.DashboardSettingsSection,Kalitte.Dashboard"
name="KalitteDashboard" />
```

It should look like this:

```
<configuration>
  <configSections>
    <!-- For more information on Entity Framework configuration, visit http://go.microsoft.com/fwlink/?LinkID=237468 -->
    <section name="entityFramework" type="System.Data.Entity.Internal.ConfigFile.EntityFrameworkSection, EntityFramework, Version=4.
    <section type="Kalitte.Dashboard.Configuration.DashboardSettingsSection,Kalitte.Dashboard" name="KalitteDashboard" />
  </configSections>
```

Add following property in “configuration” area in master “Web.config” file of application.

```
<KalitteDashboard defaultProvider="SQLDashboardProvider">
  <providers>
    <clear />
    <add applicationName="DashboardApp" connectionString="SqlConstr"
name="SQLDashboardProvider"
type="Kalitte.Dashboard.SQLProvider.Provider,Kalitte.Dashboard.SQLProvider" />
  </providers>
</KalitteDashboard>
```

Here how it should look:

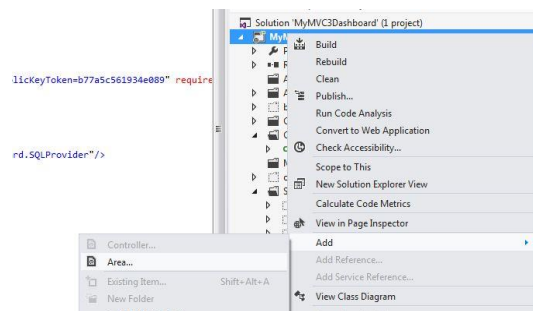
```
<section type="Kalitte.Dashboard.Configuration.DashboardSettingsSection,Kalitte.Dashboard" name="KalitteDashboard" />
</configSections>
<KalitteDashboard defaultProvider="SQLDashboardProvider">
  <providers>
    <clear/>
    <add applicationName="DashboardApp" connectionString="SqlConstr" name="SQLDashboardProvider" type="Kalitte.Dashboard.SQLProvider.Provider,Kalitte.Dashboard.SQLProvider"/>
  </providers>
</KalitteDashboard>
<appSettings>
  <add key="webpages:Version" value="1.0.0.0"/>
  <add key="ClientValidationEnabled" value="true"/>
```

More than one data provider can be defined here as well. For example:

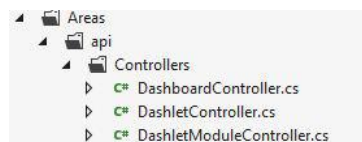
```
<KalitteDashboard defaultProvider="OracleDashboardProvider ">
  <providers>
    <clear />
    <add applicationName="DashboardApp" connectionString="SqlConstr"
name="SQLDashboardProvider"
type="Kalitte.Dashboard.SQLProvider.Provider,Kalitte.Dashboard.SQLProvider" />
    <add applicationName="DashboardApp" connectionString="OracleConstr"
name="OracleDashboardProvider"
type="Kalitte.Dashboard.OracleProvider.Provider,Kalitte.Dashboard.OracleProvider"
/>
  </providers>
</KalitteDashboard>
```

Don't forget to set `defaultProvider` and `connectionString` properties. Dashboard initializes provider which is defined in `defaultProvider` property. That's all configuration for "Web.config" file.

An "api" area is required to handle dashboard requests. Right click your project and select "Add->Area" from the menu. Name it as "api".



An area should be created. Create following classes under "Controllers" folder of "api" area.



These classes handle dashboard requests. They must be inherited from base classes under `Kalitte.Dashboard.Mvc.Controllers` namespace. For example "DashboardController.cs" class must be defined like `public class DashboardController : DashboardControllerBase`. Any method in base class can be override here.

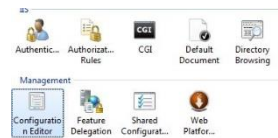
ASP.NET MVC 4 CONFIGURATION

Follow every instruction for MVC 3 configuration. Some extra modifications need to be done. First go "RouteConfig.cs" file under "App_Start" folder of your application. There should be a route setting for "api" area. Modify it like this:

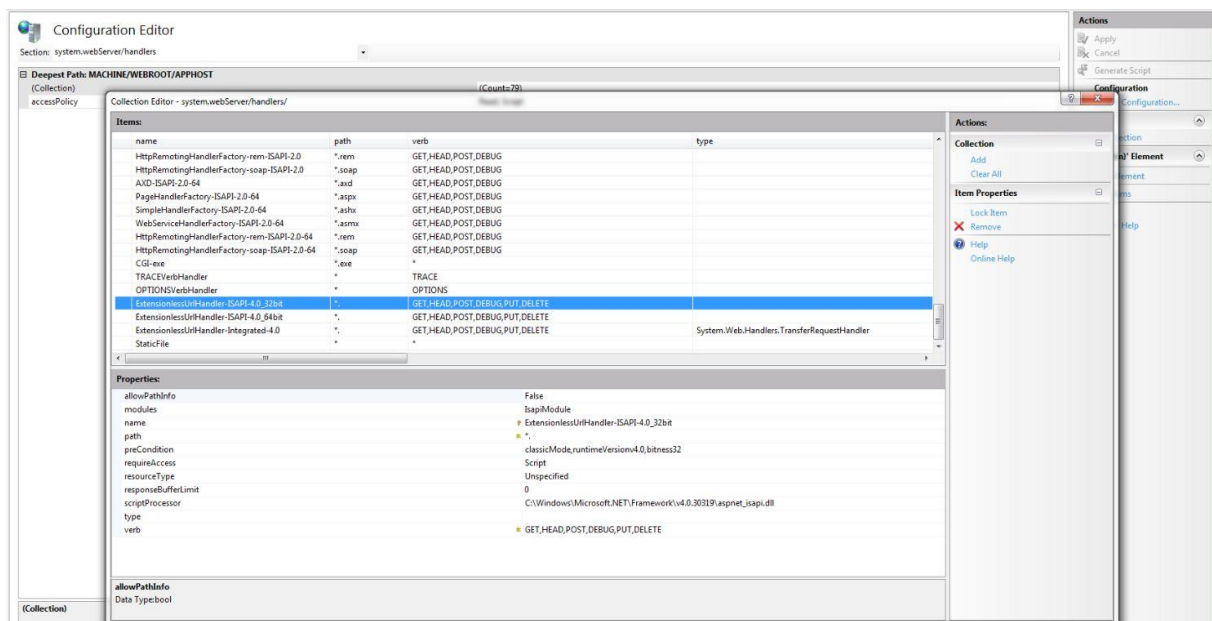

```
routes.MapHttpRoute (
    name: "DefaultApi",
    routeTemplate: "api/{controller}/{action}/{id}",
    defaults: new { id = RouteParameter.Optional } );
```

This should be enough to run dashboard application. If you receive errors in some HTTP verbs like “PUT” or “DELETE” you may need IIS configuration.

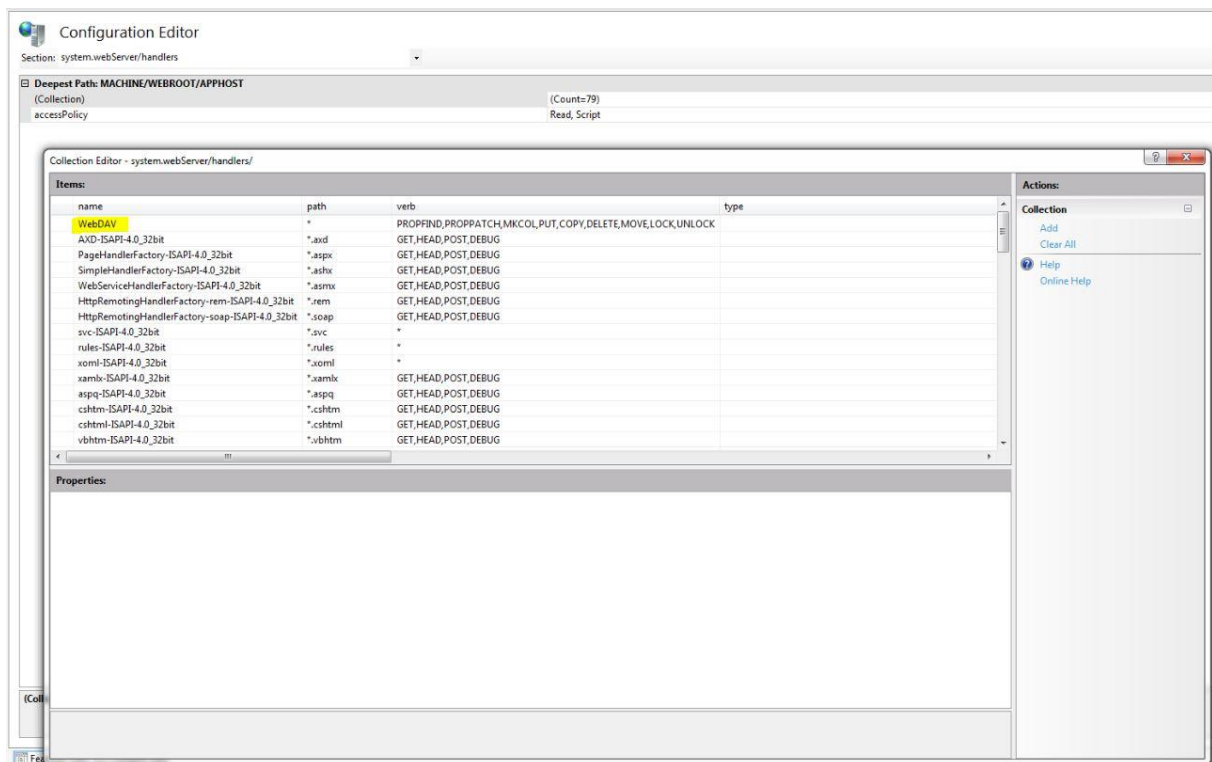
Select “Configuration Editor” in the manage server menu.



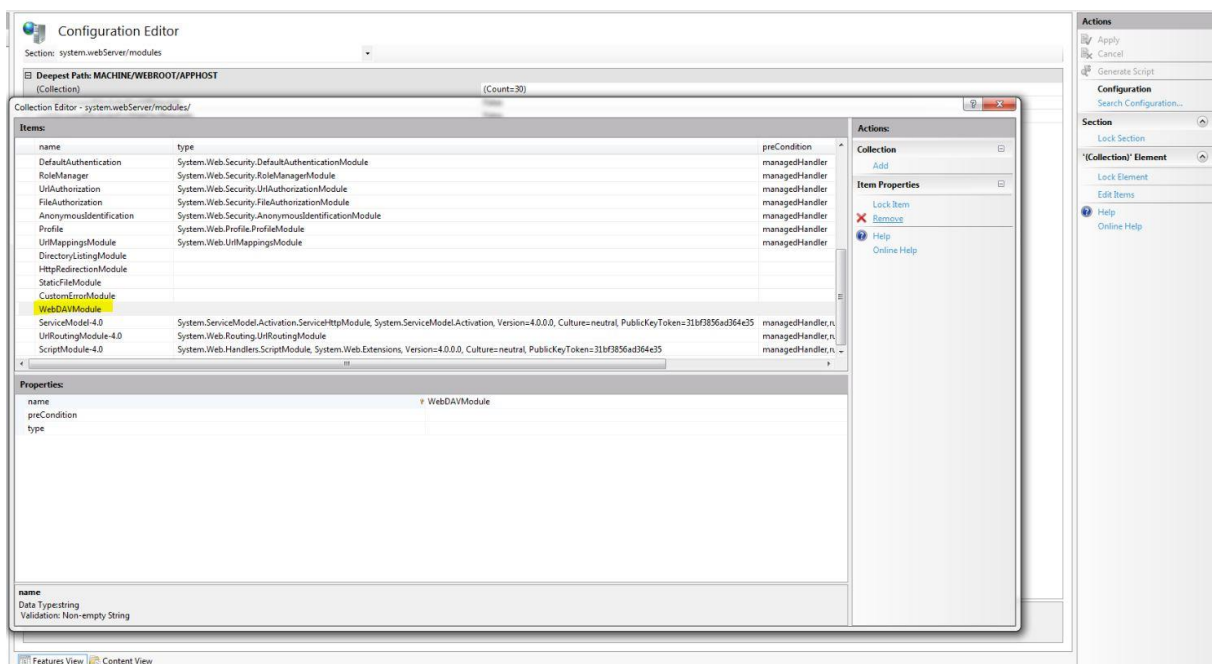
Select “system.webServer/handlers” from sections menu. Click browser button of “(Collection)” row. Add “PUT” and “DELETE” verbs in properties which start with “ExtensionlessUrlHeader”.



If there is a “WebDav” property, delete it.



If there is a “WebDavModule” property under “system.webServer/modules” collection, delete it too.

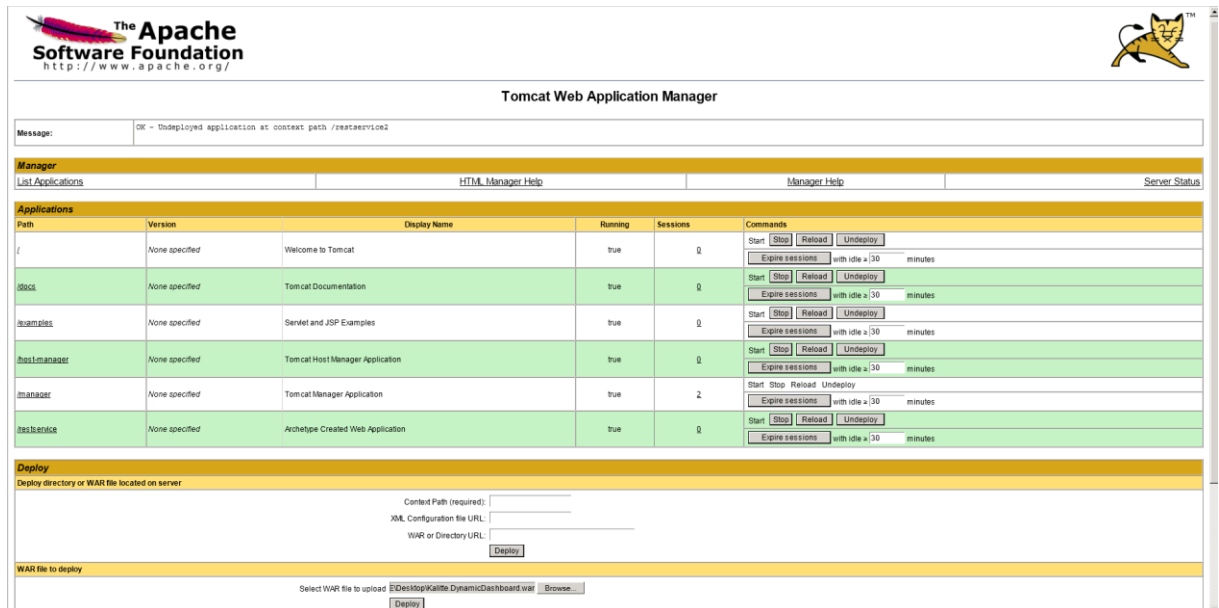


CONFIGURE SERVER SIDE WITH JAVA

You can deploy Dynamic Dashboard Java Rest Service by using Apache Tomcat®

Prerequisites: Apache Tomcat 7®, Java Runtime Environment 6.0+ and MySQL® 5.5+

- Install [Apache Tomcat 7®](#) (32-bit/64-bit Windows Service Installer), [Java Runtime Environment 6.0+](#) and [MySQL® 5.5+](#)
- Open Apache Tomcat Manager Page. You can find this URL in start menu “Apache Tomcat 7.0 Tomcat7” folder.
- Scroll to “War file to deploy” section. Select Dynamic Dashboard Rest Service war file then click deploy.



The screenshot shows the Apache Tomcat Web Application Manager interface. At the top, there's a message box indicating a successful deployment. Below that, there's a navigation bar with links like 'List Applications', 'HTML Manager Help', 'Manager Help', and 'Server Status'. The main content area is divided into two sections: 'Applications' and 'Deploy'. The 'Applications' section contains a table with columns: Path, Version, Display Name, Running, Sessions, and Commands. The 'Deploy' section has fields for 'Context Path (required)', 'XML Configuration file URL', and 'WAR or Directory URL', along with a 'Deploy' button. The 'WAR file to deploy' section at the bottom shows a file upload area with a 'Browse...' button and a 'Deploy' button.

Path	Version	Display Name	Running	Sessions	Commands
/	None specified	Welcome to Tomcat	true	0	Start Stop Reload Undeploy
/docs	None specified	Tomcat Documentation	true	0	Start Stop Reload Undeploy
/examples	None specified	Servlet and JSP Examples	true	0	Start Stop Reload Undeploy
/host-manager	None specified	Tomcat Host Manager Application	true	0	Start Stop Reload Undeploy
/manager	None specified	Tomcat Manager Application	true	2	Start Stop Reload Undeploy
/test-ahc	None specified	Archetype Created Web Application	true	0	Start Stop Reload Undeploy

- If you get “org.apache.tomcat.util.http.fileupload.FileUploadBase\$SizeLimitExceededException” error, go to Apache Tomcat 7.0 installation folder. Now, go to “webapps\manager\WEB-INF\” folder. Open web.xml in notepad. Replace the upload size value with larger value then size of war file and try to deploy war file again.

```
<multipart-config>
  <!-- 50MB max -->
  <max-file-size>524288000</max-file-size>
  <max-request-size>524288000</max-request-size>
  <file-size-threshold>0</file-size-threshold>
</multipart-config>
</servlet>
```

- Now, if you deploy war file successfully, Apache will extract this file to “[Apache Tomcat 7.0 installation folder]\ webapps\[file name of war file]”. Navigate to this folder.
- Navigate to “WEB-INF\classes\” folder. Then open “application.properties” file with notepad. File shown in below.

```
jdbc.url=jdbc:mysql://[ServerIP]:3306/[DatabaseName]
jdbc.driverClassName=com.mysql.jdbc.Driver
jdbc.username=[Username]
jdbc.password=[Password]
```

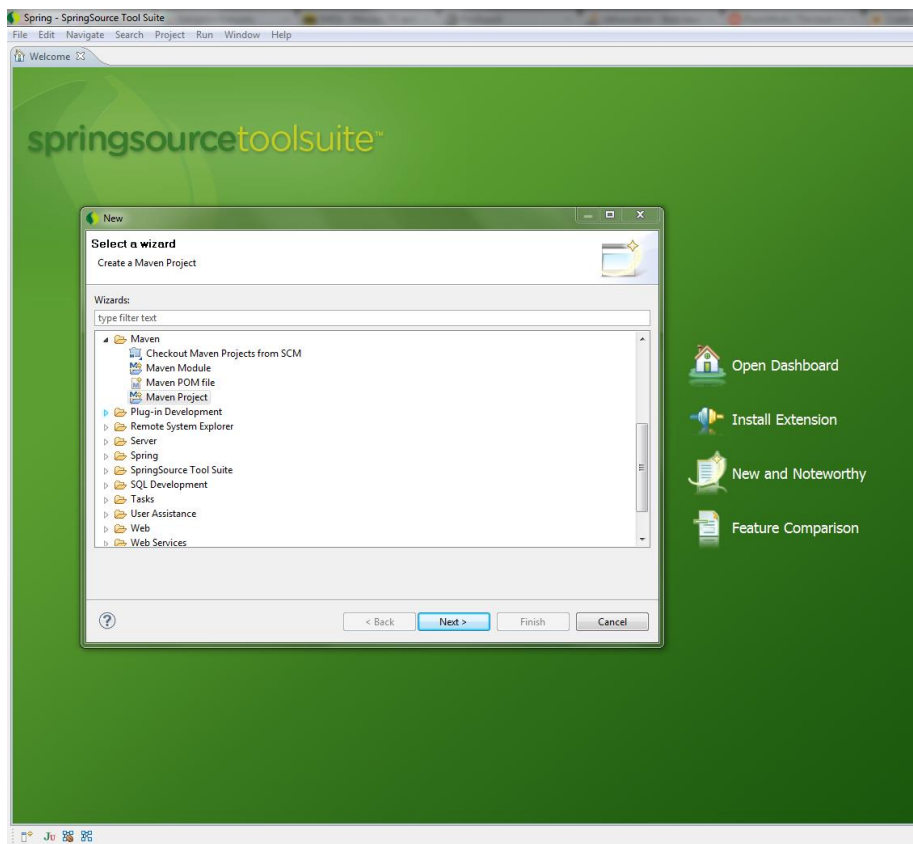
- This file contains MySQL connection parameters. To provide Connection parameters , we will first create metadata.
- Open MySql management tool and run DynamicDashboard MySQL Metadata script
- Now, you created database for Dynamic Dashboard.
- Input Dynamic Dashboard Database information to “application.properties” file opened four steps above. Save the file and close.
- Open Apache Tomcat Manager Page again. Click reload button which is in commands section of your application row.
- Congratulations, you deployed Dynamic Dashboard Java Rest Service successfully.

CREATE CUSTOM DASHBOARD PROVIDER WITH JAVA

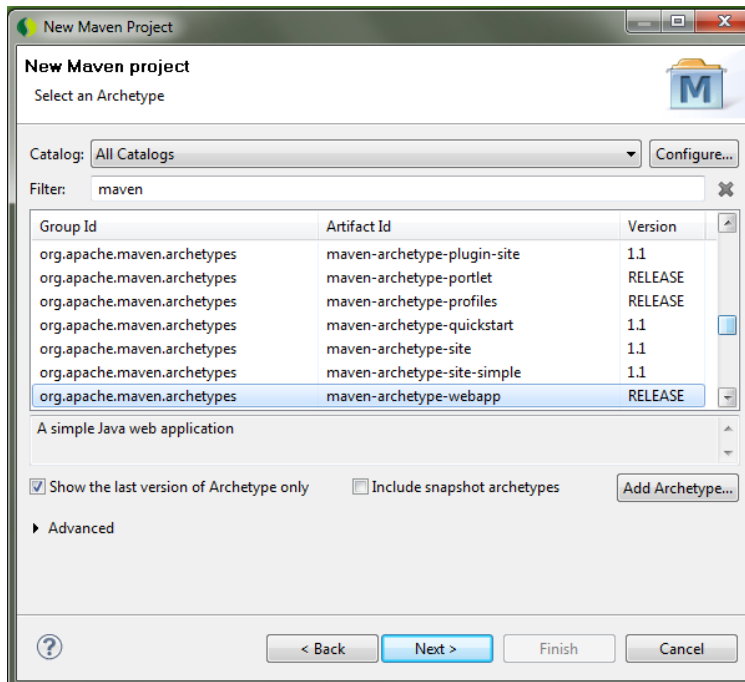
You can create your custom dashboard provider with java by using SpringSource Tool Suite®.

Prerequisites : SpringSource Tool Suite® and Java Development Kit

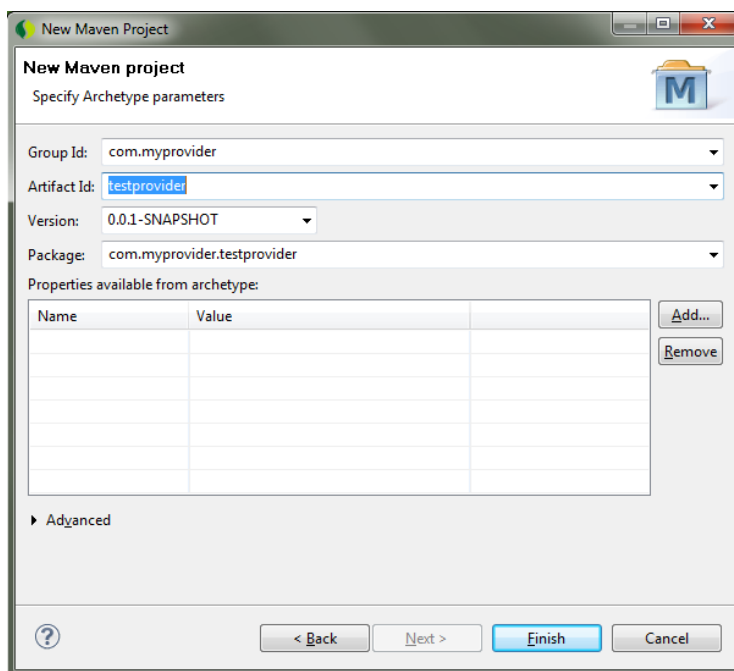
- Install [JDK](#) and [SpringSource Tool Suite](#)®.
- Open SpringSource Tool Suite® and Create a New Maven Project



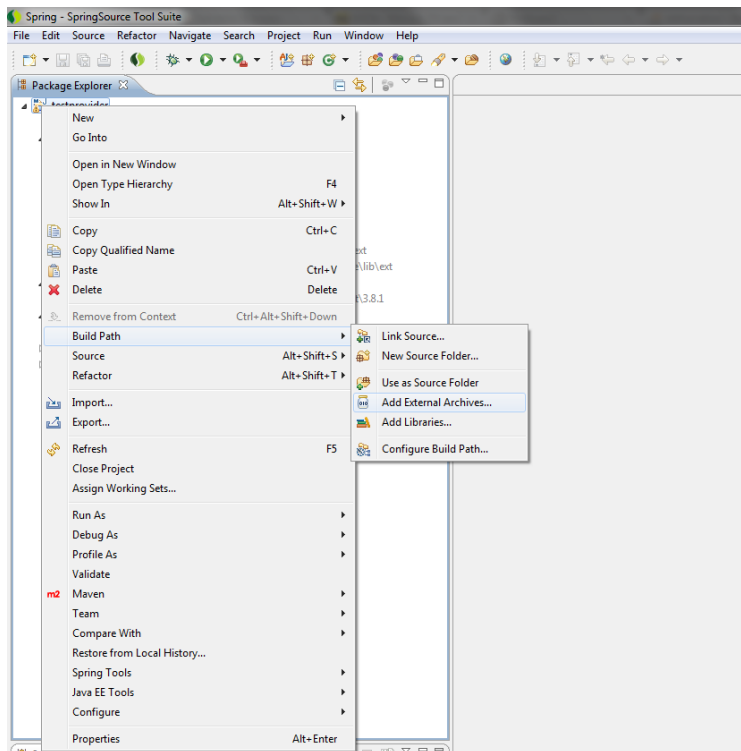
- Select maven-archetype-webapp as an Archetype



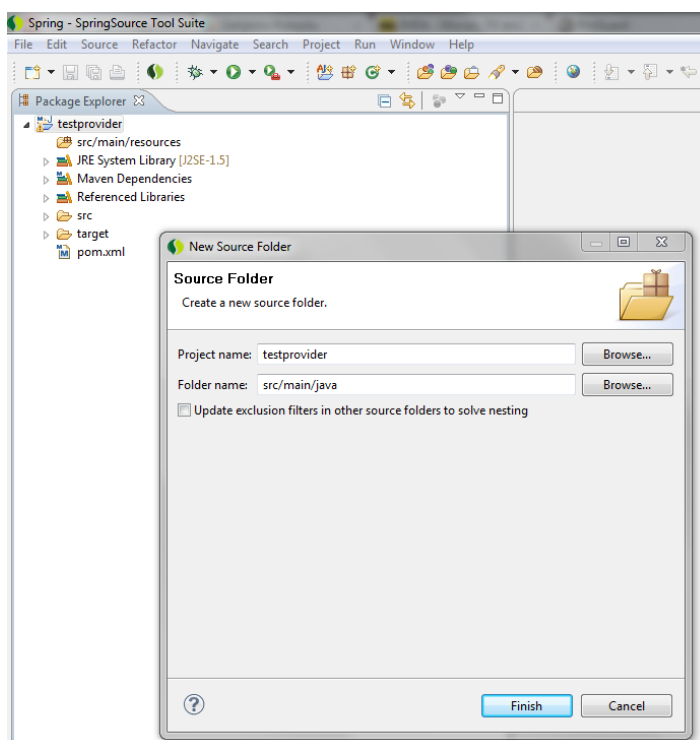
- Specify archetype parameters and click finish. Your maven project will be created.



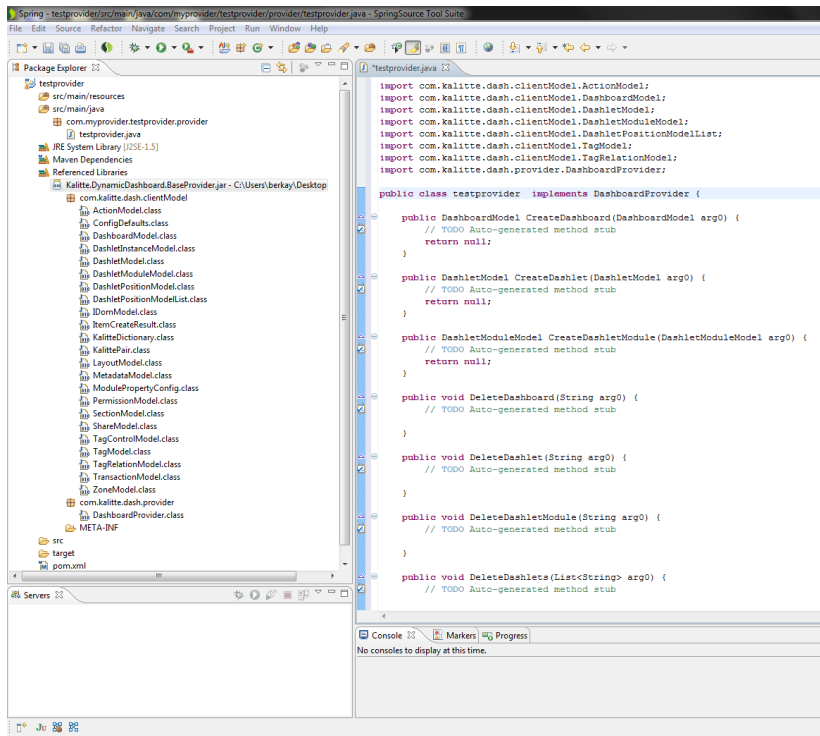
- Right click on project root element and select build path then click Add External Archives.



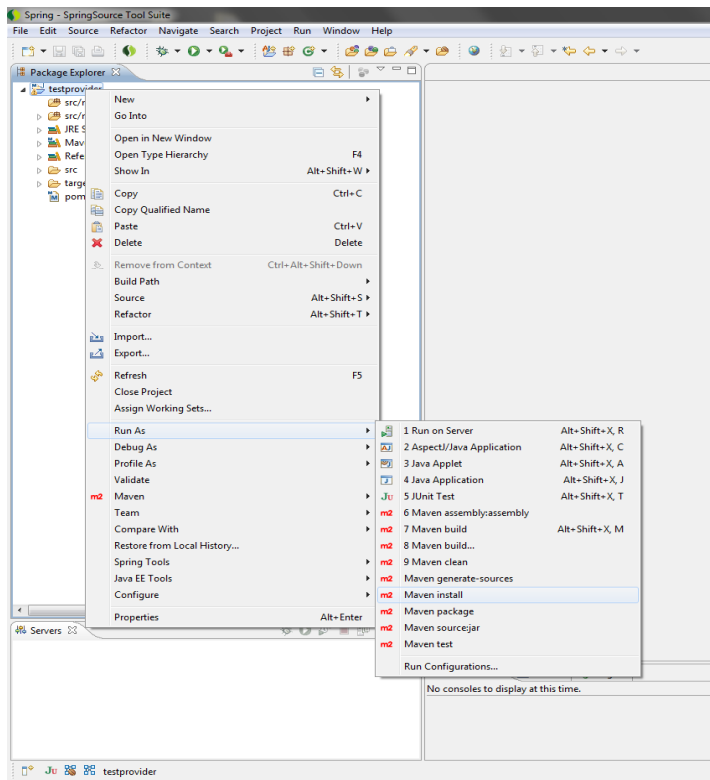
- Add Kalitte.DynamicDashboard.BaseProvider.jar from file system.
- Create a new source source folder and a package. Then you can create your provider class.



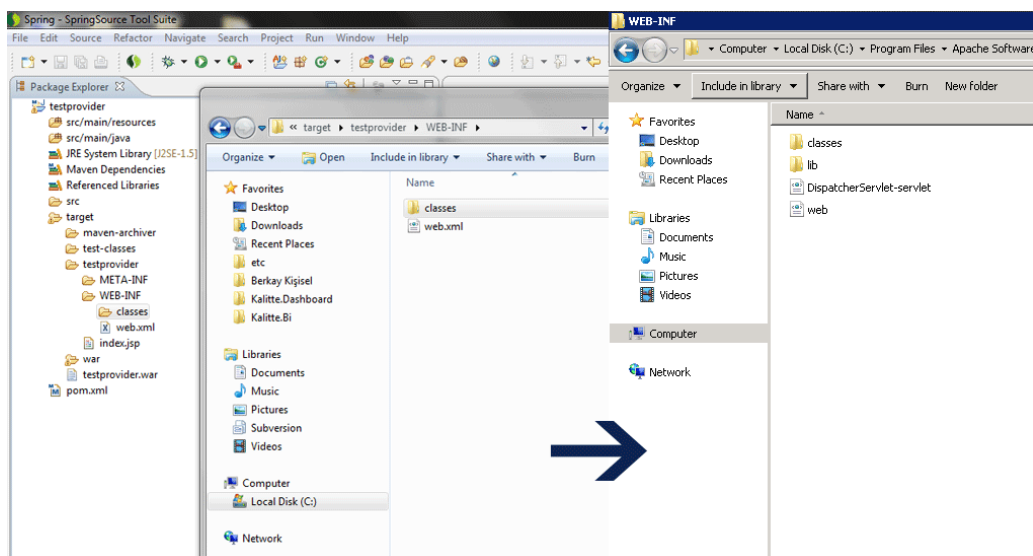
- Implement the DashboardProvider in your provider class. After that, you can develop your dashboard provider using the other data sources.



- Now, you can build your code and then you can deploy it to server. To compile your code, right click on project root element and select run as node, then click maven install.



- If you can build successfully, go to `target\[project name]\web-inf\` folder which is in the project root folder.
- You can copy the classes folder and other resource files (if you have) to “web-inf” folder of restful service.



- Open `DispatcherServlet-servlet.xml` and replace class of `dataProvider` bean with your custom `dataProvider`.


```

        class="org.springframework.http.converter.ByteArrayHttpMessageConverter" />
    <bean
        class="org.springframework.http.converter.StringHttpMessageConverter" />
    <bean
        class="org.springframework.http.converter.ResourceHttpMessageConverter" />
    <bean
        class="org.springframework.http.converter.xml.SourceHttpMessageConverter" />
    <bean
        class="org.springframework.http.converter.xml.XmlAwareFormHttpMessageConverter" />
    <!-- bean class="org.springframework.http.converter.xml.Jaxb2RootElementHttpMessageConverter"
        / -->
    </list>
    </property>
</bean>

<bean id="handlerMapping"
    class="org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping" />

<bean id="transactionManager"
    class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource" />
</bean>

<bean id="jdbcTemplate"
    class="org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate">
    <constructor-arg ref="dataSource"></constructor-arg>
</bean>

<bean
    class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="viewClass"
        value="org.springframework.web.servlet.view.JstlView" />
    <property name="prefix" value="/" />
    <property name="suffix" value=".jsp" />
</bean>

<!-- <bean id="dataProvider" class="com.kalitte.dash.mysqlprovider.MySqlProvider" /> -->
<bean id="dataProvider" class="com.myprovider.testprovider.provider.Testprovider"
    />

```

- Reload you Restful Service application from Apache Manager.
- Congratulations, you developed custom data provider and configured successfully.