

jRIApp, the RIA application framework – user guide

jRIApp, the application framework

- 1.1 What is the jRIApp framework
- 1.2 The Framework's files and deployment
- 1.3 Licensing
- 1.4 The Future

The framework's types and objects

- 2.1 The BaseObject type
 - Creation and destruction
 - Events
 - Event handlers
 - Property change notifications
- 2.2 The Global object type
 - Its Role in the framework
 - Defaults adjustment
 - Global modules
- 2.3 The Application object type
 - Its Role in the framework
 - Application modules
 - User modules
 - Error handling
- 2.4 The Data Binding type and its usage
 - Binding's properties
 - Converters
 - Converter parameters
- 2.5 The Command type
- 2.6 Element's views
- 2.7 Data templates
- 2.8 Data contents
- 2.9 Controls
 - DataGrid
 - DataPager
 - DataEditDialog
 - DataForm
 - StackPanel
 - ListBox (*aka ComboBox*)

Working with the data on the client side

- 3.1 Working with simple collection's data
 - Collections and CollectionItems
- 3.2 Working with the DataService fed data
 - DbContext
 - DataCache
 - DataQuery and loading data
 - DataService's methods invocation
 - DbSet type

- Entities
- DataView type
- Associations and ChildDataViews
- Data validation

Working with the data on the server side

4.1 Data service

- Data service's public interface
- Exposing the data service through ASP.NET MVC controller
- Query methods
- Entity refresh methods
- Custom validation methods
- Service methods
- Data service's metadata
- Associations (*foreign keys*)
- Authorization
- Tracking changes
- Error logging
- Service disposal (*cleanup*)

jRIApp, the RIA application framework

1.1 What is the jRIApp framework

jRIApp – is application framework for developing rich internet applications - RIA's. It consists of two parts – the client and the server parts. The client part was written in javascript language. The server part was written in C# and the demo application was implemented in ASP.NET MVC (*it can also be written in other languages, for example Ruby or Java, but you have to roll up your sleeves and prepare to write them*).

The Server part resembles Microsoft WCF RIA services, featuring data services which is consumed by the clients.

The Client part resembles Microsoft silverlight client development, only it is based on HTML (*not XAML*), and uses javascript language for coding.

The framework was designed primarily for building data centric Line of business (LOB) applications which will work natively in browsers without the need for plugins .

The framework supports wide range of essential features for creating LOB applications, such as, declarative use of data bindings, integration with the server side service, data templates, client and server side data validation, localization, authorization, and a set of GUI controls, like the datagrid, the stackpanel and a lot of utility code.

Unlike many other existing frameworks, which use MVC design pattern, the framework uses Model View View Model (MVVM) design pattern for creating applications. The use of data bindings and view models resembles Microsoft silverlight data bindings style used in the XAML.

The framework was designed for getting maximum convenience and performance, and for this sake it works in browsers which support ECMA Script 5.1 level of javascript and has features like native property setters and getters, and a new object creation style with Object.create method.

The supported browsers are Internet Explorer 9 and above, Mozilla Firefox 4+, Google Chrome 13+, and Opera 11.6+. Because the framework is primarily designed for

developing LOB applications, the exclusion of antique browsers does not harm the purpose, and improves framework's performance and ease of use.

The framework is distinguished from other frameworks available on the market by its full implementation of the features required for building real world LOB applications. It has implemented server side component - the data service. It has GUI controls that are aware of the events raised by data centric objects like the DbSet and the Entity.

For the creation of data centric applications the framework has GUI controls for working with the server originated data, with editing support, and submitting changes to the server with the data passing through the data validation and authorization stages of data processing, returning autogenerated field values - such as primary keys, timestamp values. The framework includes the ability to track changes (*auditing*) and do the error logging.

For this very purpose the framework contains a set of user controls such as:

DataGrid – the control for displaying and editing of the data in tabular style. It supports databinding, row selection with keyboard keys, column sorting, data paging, detail row, data templates, different column's types (*expander column, row selector column, actions column*). For editing it can use the builtin inline editor, and has the support for a popup editor which can be designed with the help of a data template.

StackPanel - the control for displaying and editing of the data as a horizontal or vertical list with the help of a data template and the support for item selections with keyboard keys.

ListBox - the control integrates HTML select tag with the collection's data for displaying options.

DataForm - the control for displaying and editing of the data item with the help of a data template.

DbContext – the control used as a data manager to store data collections (*DbSets*) and to cache changes on the client for submitting them to the server backend.

This is just an overview of the main features, they and the other ones will be discussed in more details later in this user guide.

1.2 The Framework's files and deployment

The Client part of the framework is deployed in one folder, **jriapp**, which is usually located under **Scripts** web application folder (*it can be renamed if desired*).

Inside **jriapp** folder is **jriapp.css** (*the styles for frameworks's GUI controls*) and **img** folder – which contains the image files.

For the localization purposes the framework's javascript code is splitted in two parts. One part is the main file **jriapp.js**, the other part is the localized version of the strings **jriapp_(lang).js**.

for example:

jriapp_en.js – is localization code for the english version of the strings.

jriapp_ru.js – is localization code for the russian version of the strings.

The framework is dependent on several third party javascript libraries: JQuery (*1.7 and above*), JQuery UI (*for calendars, tabs and the other UI controls*), datejs (*for dates formatting*), and qtip (*for tooltips*). Thus, these javascript libraries files should be referenced on the html page for the framework's code to work properly.

P.S. - I recommend to use JetBrains WebStorm javascript editor to look in the framework's source code to better understand its type organization. (it can collapse type definitions which is very convenient).

```
1 use strict;
2 if (!window['RIAPP']) {...}
5
6 RIAPP.utils = (function () {...})();
130
131 RIAPP.global = null;
132 RIAPP.Application = null;
133
134 RIAPP.BaseObject = {...};
399
400 RIAPP._glob_modules = ['array_ext', 'consts', 'errors', 'defaults', 'utils', 'converter', 'riapp'];
401 RIAPP._app_modules = [...];
403
404 RIAPP.Global = RIAPP.BaseObject.extend({"version": "1.0.0..."},
591   {...});
643
644 RIAPP.Global._coreModules.errors = function (global) {...};
790
791 RIAPP.Global._coreModules.defaults = function (global) {...};
963
964 RIAPP.Global._coreModules.array_ext = function (global) {...};
987
988 RIAPP.Global._coreModules.consts = function (global) {...};
1037
1038 RIAPP.Global._coreModules.utils = function (global) {...};
1770
1771 RIAPP.Global._coreModules.converter = function (global) {...};
1930
1931 RIAPP.Global._coreModules.riapp = function (global) {...};
2478
2479 //...
2481 RIAPP.global = RIAPP.Global.create(window, jQuery, []);
2482
2483 RIAPP.Application._coreModules.parser = function (app) {...};
2741
2742 RIAPP.Application._coreModules.collection = function (app) {...};
4021
4022 RIAPP.Application._coreModules.db = function (app) {...};
7708
7709 RIAPP.Application._coreModules.datadialog = function (app) {...};
8089
8090 RIAPP.Application._coreModules.datagrid = function (app) {...};
10122
10123 RIAPP.Application._coreModules.pager = function (app) {...};
10569
10570 RIAPP.Application._coreModules.stackpanel = function (app) {...};
10896
10897 RIAPP.Application._coreModules.listbox = function (app) {...};
11336
11337 RIAPP.Application._coreModules.dataform = function (app) {...};
11628
```

1.3 Licensing

The MIT License

Copyright (c) 2013 Maxim V. Tsapov

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions: The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

1.4 The Future

The client part framework is currently written javascript language, and now in stable form. I don't think that there are reasons to change its public interface. But I think that it would be great to translate the framework to new Microsoft typescript language, because it allows type safety, easy refactoring, and superb intelligence when building applications. So in the near future I will start its translation, but it will be another project on github.

The framework's types and objects

2.1 The BaseObject type

All the types used in the client part of the framework derived from the [RIAPP.BaseObject](#) type. The [RIAPP.BaseObject](#) type includes the logic for the type extension, calling super (*inherited*) methods, adds common logic for object's creation and destruction and adds events support for all objects derived from it.

BaseObject's methods:

Property	Description
extend	used to extend an object type with new properties and methods
_super	used to call a base method implementation which had been overridden. Optionally you can provide arguments for the base method.
addHandler	adds an event handler for an event (<i>with optional support for event namespaces like in jQuery</i>)
removeHandler	removes a handler for an event (<i>can be used to remove all handlers registered within a namespace</i>)
removeNSHandlers	removes all handlers for events registered with an event namespace
addOnPropertyChange	adds a handler for a property change notification
removeOnPropertyChange	removes a handler for a property change notification
raisePropertyChanged	triggers registered events handlers for a property change notification
raiseEvent	triggers registered events handlers for an event
destroy	the method is invoked when the object needs to be destroyed for cleaning up resources such as registered event handlers and other resources. It is usually overridden in descendants (<i>but don't forget to always invoke super method!</i>)
_getEventNames	defines event names supported by the object type.

	BaseObject type supports 'error', and 'destroyed' events. Descendants of the BaseObject can override this method to add their own events.
<code>_onError</code>	typically it is invoked by descendants of the BaseObject on error conditions. It triggers the error event and returns boolean value of handled the error or not. Many frameworks object types override this method, for example, in BaseElView type it invokes <code>_onError</code> inherited from BaseObject, and if the error was not handled it invokes application's <code>_onError</code> method.
<code>_create</code>	used as a constructor for the object. It has object initialization code, but it is <u>not</u> invoked directly by the user's code. When a user creates object instance it instead invokes <code>create</code> method. (<i><code>_create</code> method is invoked internally</i>).
<code>create</code>	used directly by the users to instantiate the object. It is not overridden and used as is.

BaseObject's events:

Property	Description
<code>error</code>	event is raised from the BaseObject's <code>_onError</code> method.
<code>destroyed</code>	event is raised when the object's destroy method completed

an example of a new object definition (derived from RIAPP.BaseObject) :

```

var NewObject = RIAPP.BaseObject.extend(
{
    //constructor's code
    _create:function () {
        this._super();
        //initialize instance variables here
        this._radioValue = 'radioValue1';
    },
    //overriding base method and define new event
    _getEventNames:function () {
        var base_events = this._super();
        return ['radio_value_changed'].concat(base_events);
    },
    //trigger the event in our custom method
    _onRadioValueChanged: function(){
        this.raiseEvent('radio_value_changed',{value: this.radioValue})
    }
},
{
    //defining a new property
    radioValue:{
        set:function (v) {
            if (this._radioValue !== v){
                this._radioValue = v;
                this.raisePropertyChanged('radioValue');
                this._onRadioValueChanged();
            }
        },
        get:function () {
            return this._radioValue;
        }
    }
}
);

```

```
function (obj) {
    //register the new type with the application
    //app here is accessed by an outer scope variable by closure
    app.registerType('custom.NewObject', obj);
};
```

Example of creating an instance of the new object:

```
//direct object creation (used locally where NewObject variable is visible)
var instance1 = NewObject.create();
//or we can get our registered type elsewhere in the application's code, and then create an instance of it
var Instance2 = app.getType('custom.NewObject').create();
```

BaseObject's instance can trigger two predefined events 'error', 'destroy'.

The **error** event is raised from the BaseObject's **_onError** method

```
_onError:function (error, source) {
    if (!!RIAPP.global && RIAPP.global._checkIsDummy(error)) {
        return true;
    }
    if (!error.message){
        error =new Error(""+error);
    }
    var args = { error:error, source:source, isHandled:false };
    this._raiseEvent('error', args);
    return args.isHandled;
}
```

The Error event handler can set isHandled to true , and then the error handling is successfully finished.

The **destroyed** event is raised from the BaseObject's **destroy** method. It is used to notify about object destruction and can be used by subscribers to remove references for the destroyed object.

The BaseObject type allows the descendants to override **_getEventNames** method in order to define new custom events as in the example:

```
_getEventNames:function () {
    var base_events = this._super();
    return ['open','close', 'error', 'message', 'status_changed'].concat(base_events);
}
```

The users can subscribe to the events by using **addHandler** method, as in the example:

```
ourCustomObject.addHandler('status_changed', function (sender, args) {
    if (args.item._isDeleted){
        self.dbContext.submitChanges();
    }
}, self.uniqueID);
```

The last argument of the **addHandler** method is the event's namespace, which is an optional parameter.

The event's namespace can be used to remove subscriptions to the object's events in one swoop by using **removeNSHandlers** method. For removing the event's subscriptions we can also use **removeHandler** method, supplying to it the event's name and optionally the event's namespace.

```
//remove subscription by the event name, plus the event namespace is an optional parameter.
ourCustomObject.removeHandler('status_changed', this.uniqueID);
```


The events can be triggered by using [raiseEvent](#) method as in the example:

```
_onMsg:function (event) {  
    this.raiseEvent('message', { message: event.data, data: JSON.parse(event.data) });  
}
```

For the subscription to the property change notification we can use [addOnPropertyChange](#) method as in the example:

```
ourCustomObject.addOnPropertyChange('currentItem', function (sender, data) {  
    self._onCurrentChanged();  
    },  
//the event namespace is an optional parameter.  
self.uniqueID);
```

For the subscription for all properties change notifications using only one event handler we can supply `''` instead of the property name. We can get the property name inside the handler by the use of `args.property` value, as in the example:

```
ourCustomObject.addOnPropertyChange('',function(s,a){alert('property that has been changed: ' + a.property);},  
self.uniqueID);
```

To remove a property change notification we can use [removeOnPropertyChange](#) method, or by using [removeNSHandlers](#) method, as in the example:

```
//remove a subscription by the property name, plus the event namespace is an optional parameter.  
obj.removeOnPropertyChange('currentItem', this.uniqueID);  
//remove subscriptions by the event namespace  
obj.removeNSHandlers(this.uniqueID);
```

2.2 The Global object type

All the code in the client part of framework is structured into javascript modules. The [Global](#) object instance (*namely, instance of [RIAPP.Global](#)*) is created by the framework at the time of loading of the framework's script file. It is a singleton object. The instance of this object is accessed in a code via [RIAPP.global](#). It can be also accessed by the [global](#) property on an application instance.

[RIAPP.global](#) - is used to hold references for registered types, application instances, global modules, and manages subscriptions to some [window.document](#)'s events, it also subscribes to `window.onerror` event to handle uncaught errors. It dispatches DOM document keydown events to a currently selected DataGrid or StackPanel, so that the only instance of a user control can handle keyboard events at a time (*for example, it is used for selecting a row in the DataGrid with the help of up or down keyboard keys*).

The global object on its creation initializes a set of global modules: `'array_ext', 'consts', 'errors', 'defaults', 'utils', 'converter', 'riapp'`.

For the convenience, the global object holds and exposes references to the [window](#) HTML DOM object and [jQuery](#) function. The global object also holds references for registered converters (*some converters are already registered in the [converter](#) module's function*).

The global object has [defaults](#) property, which exposes an instance of the [Defaults](#) object type (*it is instantiated in [defaults](#) global module's function*). Using this property

we can set or change the default values used in the framework, such as: default date format, time format, decimal point, thousand's separator, datepicker defaults, path to the images, as in the example:

```
global.defaults.dateFormat = 'dd.mm.yy'; //russian style date format
global.defaults.imagesPath = '/Scripts/jriapp/img/';
```

Global object's methods:

Property	Description
findApp	Finds application instance by the name.
registerType	Registers type by its name. The type can be later retrieved anywhere in the code by using getType method.
getType	Retrieves registered type by its name.
getImagePath	Get common images paths used in the framework by their names. It just appends provided image name to the default path for the images.

Global object properties:

Property	ReadOnly	Description
\$	Yes	JQuery function for easier access in the code.
types	--	the namespace (<i>object instance</i>) for registering the javascript objects types in the framework. It is not used directly by users.
window	Yes	DOM Window object instance
document	Yes	DOM Document object instance
currentSelectable	No	The currently selected control DataGrid or StackPanel which accepts keyboard input like Up or Down keys for scrolling them by keyboard.
defaults	Yes	Instance of the Default object type, to access or change the default values. This property is initialized by the global defaults module.
UC	--	the namespace (<i>empty object instance</i>) for attaching any custom user code. We can attach any code to it which can be used globally.
utils	Yes	Object which contains common utility methods. This property is attached by the global utils module.
modules	Yes	the namespace (<i>object instance</i>) for obtaining global modules' instances, and through the modules instances you can get access to the types. But it is rarely used because global object has all modules exposed through its respective properties, like: <i>utils</i> , <i>consts</i> , <i>defaults</i> .
consts	Yes	Object namespace which contains public globally accessed constants, like: <code>global.consts.KEYS</code> .

Global object events:

Property	Description
unload	Raised when the browser's window unloads

2.3 The Application object type

RIAPP.Application - object type which is used for the creation of application instances. On the creation of an application instance we can supply an array of the custom module names, which need to be initialized (*modules are just functions, in which we define our objects in a sandboxed manner*). On its creation an application instance initializes predefined modules 'parser', 'collection', 'db', 'listbox', 'datadialog', 'datagrid', 'pager', 'stackpanel', 'dataform', 'binding', 'template', 'content', 'mvvm', 'elview'. The types defined in these modules, in contrast to global modules, are application specific and they all have property **app** for the access to the application instance.

The main method of the application is **startUp**, which is used to trigger execution of the callback function (*which is provided as part of the parameter*). The callback function used as the sandbox environment, in which can be created the view model instances (*which is usually defined in custom modules*) and any other user defined objects.

Every HTML page which uses the framework, can create one or several instances of **RIAPP.Application** type (*and typically one is enough for a page*).

Example of a typical code for the application initialization on a page:

```
$(document).ready(function ($) {
    //global error handler - the last resort (typically display message to the user)
    RIAPP.global.addHandler('error', function (sender, args) {
        alert(args.error.message);
    });

    var fn_Main = function (app) {
        //initialize images folder path
        app.global.defaults.imagesPath = '@Url.Content("~/Scripts/jriapp/img/")';

        //create view models which are in our custom modules
        app.VM.errorVM = app.getType('custom.ErrorViewModel').create();
        app.VM.headerVM = app.getType('custom.HeaderVM').create();
        app.VM.productVM = app.getType('custom.ProductViewModel').create();
        app.VM.uploadVM = app.getType('custom.UploadThumbnailVM').create('@Url.RouteUrl("Default", new
        {controller="Upload", action="ThumbnailUpload"}));
        //adding event handler for our custom event
        app.VM.uploadVM.addHandler('files_uploaded', function (s, a) {
            a.product.refresh(); //need to update ThumbnailPhotoFileName
        });

        //lookups data can be embedded in the page code
        app.VM.productVM.filter.modelData = @Html.Action("ProductModelData", "RIAppDemoService");
        app.VM.productVM.filter.categoryData = @Html.Action("ProductCategoryData", "RIAppDemoService");
        app.VM.productVM.load().pipe(function (extrainfo) { alert(extrinfo.test); }, function () { alert('load failed'); });

    }; //end fn_Main

    //create an application instance
    var app = RIAPP.Application.create({
        service_url: '@Url.RouteUrl("Default", new {controller="RIAppDemoService", action=""})',
        metadata: @Html.Action("Metadata", "RIAppDemoService"),
        createDbContext: true, //we want the application to create default DbContext's instance
        moduleNames: ['common', 'header', 'gridDemo'] //custom modules which will be initialized
    });

    //here we could process application's errors
    app.addHandler('error', function (sender, data) {
        //debugger;
        data.isHandled = true;
        if (!app.VM.errorVM)
        {
```

```

        alert(data.error.message);
        return;
    }
    app.VM.errorVM.error = data.error;
    app.VM.errorVM.showDialog();
});

//define calculated fields for entities in the application's event handler
app.addHandler('define_calc', function (sender, data) {
    if (data.dbSetName == 'Product' && data.fieldName == 'IsActive') {
        data.getFunc = function () {
            return !this.SellEndDate;
        };
    }
});

//start application - fn_Main will init all view models
//at the end (after executing fn_Main) this method performs databinding
app.startUp(fn_Main);
});

```

The second important method `setUpBindings` (*yet it is not used directly by the user's code*), it is invoked inside `startUp` method (*after executing our callback function*). This method creates data bindings instances which are declared on the page.

To handle errors we can subscribe to the application's `'error'` event. This event is raised when some object instance inside of the application catches an error and invokes the `_onError` method (*typically, data bindings and user defined view models do this*). If the error is not handled in the application's error handler, the error is passed on to the global object, where it can be handled in the global's error event handler.

Application's methods:

Method name	Description
<code>registerElView</code>	Registers element views in the application type system
<code>getElementView</code>	Returns the element view which is already attached to the element or creates new element view and attaches it to the DOM element and then returns this new instance.
<code>_getElementViewType</code>	Returns the element view type by its registered name. Typically it is used internally by the framework.
<code>findGrid</code>	Returns a DataGrid's instance by the name.
<code>bind</code>	Common method for creation of the data bindings. Used by the application as a data bindings factory.
<code>registerType</code>	Registers an object type by its name. The type can be later retrieved by <code>getType</code> method.
<code>getType</code>	Returns the registered type by its name.
<code>registerConverter</code>	Registers converter type by the name.
<code>getConverter</code>	Returns registered converter type by the name.
<code>startUp</code>	Starts the application's instance. It accepts a function which is executed when application is started.

Applications's properties:

Property	ReadOnly	Description
<code>instanceNum</code>	Yes	the unique number of the application's instance

appName	Yes	the application's unique name. If it is not provided on the creation, it will have default value 'default'. The name must be unique. Different applications instances can not share the same name.
modules	--	the namespace (<i>object instance</i>) for obtaining application modules' instances, and through the modules instances you can get access to the types defined in them. Examples: <i>app.modules.template.Template,</i> <i>app.modules.db.DbContext,</i> <i>app.modules.datadialog.DataEditDialog.</i>
global	Yes	exposes an instance of RIAPP.Global object for easier access to the global object.
dbContext	Yes	exposes an instance of DbContext object if it is created when the application's instance is created.
UC	---	the namespace (<i>empty object instance</i>) for attaching any custom user code. We can attach any code to it.
VM	---	the namespace (<i>empty object instance</i>) for attaching user defined view models. We can attach view model's instances to this namespace.
localizable	No	exposes RIAPP.localizable (<i>common localization's strings</i>). It can be used to databind some text properties on UI elements to text values which is needed to be localized.

Application's events:

Event name	Description
define_calc	Raised when an application initializes calculated fields

an example of registering of a custom module

```

RIAPP.Application.registerModule('bindingsDemoMod', function (app) {
    var global = app.global, utils = global.utils, consts = global.consts;
    var TEXT = RIAPP.localizable.TEXT;

    var UppercaseConverter = global.getType('BaseConverter').extend({
        convertToSource: function (val, param, dataContext) {
            if (utils.check_is.String(val))
                return val.toLowerCase();
            else
                return val;
        },
        convertToTarget: function (val, param, dataContext) {
            if (utils.check_is.String(val))
                return val.toUpperCase();
            else
                return val;
        }
    }), null, function (obj) {
        app.registerConverter('uppercaseConverter', obj);
    });

    /*
     * Simple object with some properties, which can be bounded to UI elements on HTML page.
     * P.S. - for validation error display support, this object must be descendant of BaseViewModel
     * usage example: var TestObject = app.getType('BaseViewModel').extend(...)
     */
    var TestObject = RIAPP.BaseObject.extend({
        _create: function (initPropValue) {
            this._super();
            var self = this;

```

```

this._testProperty = initPropValue;
this._testProperty2 = null;
this._testCommand = app.getType('Command').create(function (sender, args) {
    self._onTestCommandExecuted();
}, self,
function (sender, args) {
    //if this function return false, then command is disabled
    return utils.check_is.String(self.testProperty) && self.testProperty.length > 3;
});

this._month = new Date().getMonth()+1;
this._months = global.getType('Dictionary').create('MonthType',{key:0,val:''},'key');
this._months.fillItems([{key:1,val:'January'}, {key:2,val:'February'}, {key:3,val:'March'},
    {key:4,val:'April'}, {key:5,val:'May'}, {key:6,val:'June'},
    {key:7,val:'July'}, {key:8,val:'August'}, {key:9,val:'September'}, {key:10,val:'October'},
    {key:11,val:'November'}, {key:12,val:'December'}], true);

this._format = 'PDF';
this._formats = global.getType('Dictionary').create('format',{key:0,val:''},'key');
this._formats.fillItems([{key:'PDF',val:'Acrobat Reader PDF'}, {key:'WORD',val:'MS Word DOC'}, {key:'EXCEL',val:'MS Excel
XLS'}], true);
},
_onTestCommandExecuted: function(){
    alert(String.format("testProperty:{0}, format:{1}, month: {2}",this.testProperty,this.format,this.month));
}
},
{
    testProperty: {
        get: function () {
            return this._testProperty;
        },
        set: function (v) {
            if (this._testProperty != v){
                this._testProperty = v;
                this.raisePropertyChanged('testProperty');
                //let command evaluate again its availability
                this._testCommand.raiseCanExecuteChanged();
            }
        }
    },
    testProperty2: {
        get: function () {
            return this._testProperty2;
        },
        set: function (v) {
            if (this._testProperty2 != v){
                this._testProperty2 = v;
                this.raisePropertyChanged('testProperty2');
            }
        }
    },
    testCommand:{
        get:function () {
            return this._testCommand;
        }
    },
    testToolTip: {
        get:function () {
            return "Click the button to execute command.<br/>" +
                "P.S. <b>command is active when testProperty length > 3</b>";
        }
    },
    format:{
        set:function (v) {
            if (this._format != v){
                this._format = v;
                this.raisePropertyChanged('format');
            }
        },
        get:function () {
            return this._format;
        }
    },
    formats:{
        get:function () {
            return this._formats;
        }
    },
    months:{

```

```

        get:function () {
            return this._months;
        }
    },
    month:{
        set:function (v) {
            if (v !== this._month){
                this._month = v;
                this.raisePropertyChanged('month');
            }
        },
        get:function () {
            return this._month;
        }
    }
}, function (obj) {
    app.registerType('custom.TestObject', obj);
});
});

```

2.4 The Data Binding type and its usage

The framework's **Binding** type has 7 properties:

Binding's properties:

Property	Description
targetPath	the path for the property which is updated when source property changes
sourcePath	the path for the property which provides value to the target property
mode	the mode of binding ('OneTime', 'OneWay', 'TwoWay')
source	the source of the data (<i>must be a descendant of the BaseObject</i>)
target	the target of the data (<i>must be a descendant of the BaseObject</i>)
converter	converts the data between source property and target property (<i>for example, object value to string value and backward</i>)
converterParam	the converter can use this parameter to adjust data conversion (<i>for example, formatting or use another code branch</i>)

The target of the data binding can be explicitly set when the **Binding's** instance is created in the code.

Data Binding creation in the code (javascript code):

```

app.getType('Binding').create({sourcePath:'selectedSendListID',
    targetPath:'sendListID',
    source:app.VM.sendListVM,
    mode:'OneWay',
    target:app.VM.attachVM,
    converter:null,
    converterParam:null});

```

When data bindings are created by an application from the binding expressions (*which are defined declaratively*), the application evaluates all the paths used in the expression to get the real object instances, then it creates the instances of the Binding's type and sets its properties.

When the declarative binding expression is parsed, HTML DOM control is wrapped with a javascript class derived from **BaseElView** type (*which defined in application's **elview** module*) for exposing properties which can be databound. Element views serve for the purpose of the databindings targets. The selection of which BaseElView's descendant

to create is determined by html element tag or it can be determined by specifying the name of element view in the custom `data-view` attribute.

For example, we can specify to create our custom Expander element view for the span tag (*instead of the default element view for the span html element*):

```
<span id="expander" style="margin:2px;"
data-bind="{this.command,to=expanderCommand,mode=OneWay,source=VM.headerVM}"
data-view="name=expander"></span>
```

We can also supply some optional parameters to an element view with using the options in data-view attribute values, as in the example:

```
<table data-app='default' data-name="tblItems"
data-bind="{this.dataSource,to=dbSet,mode=OneWay,source=VM.sendListVM.itemsVM}
{this.gridEventCommand,to=gridEventCommand,mode=OneWay,source=VM.sendListVM.itemsVM}"
data-view="options={details:{templateID:item_details}}">
```

or in

```
<select size="1" style="width:220px"
data-bind="{this.dataSource,to=mailDocsVM.dbSet,mode=OneWay,source=VM.sendListVM}
{this.selectedValue,to=selectedDocID,mode=TwoWay,source=VM.sendListVM}
{this.toolTip,to=currentItem.DESCRPTION,mode=OneWay,source=VM.sendListVM.mailDocsVM}"
data-view="options:{valuePath=MailDocID,textPath=NAME}">
```

Databinding expressions are contained inside the custom `data-bind` attribute value. A databinding's attribute value can contain multiple binding expressions. Each of the binding expressions is enclosed in figure braces `{ }` (*you can omit them if you have only one expression, but it is not recommended*). The target of the data binding in these expression is always the element view which is created when the framework's application code parses these expressions. The data binding can be done only to the properties exposed by the element view (*the wrapper of HTML DOM element*), and not directly to the HTML DOM element.

For example, the expression:

```
{this.dataSource,to=mailDocsVM.dbSet,mode=OneWay,source=VM.sendListVM}
```

instructs to bind `datasource` property on an element view to the property path `mailDocsVM.dbSet` on the data binding's source (*the view model instance*) in the OneWay mode (*which is default value*).

When databindings are used outside templates, without explicitly providing the source, then the source is assumed to be the application's instance, but when they are used inside templates, the implicit source is a template's current datacontext (*templates have the notion of the datacontexts*).

When the `source` is explicitly provided by the data binding expression, the data binding path is always evaluated starting from the application's instance. Previous expanded expression path can be represented as `app.VM.sendListVM.mailDocsVM.dbSet`.

Very often we can omit source attribute in the binding's expression (*using implicit source*), and could write previous binding expression as:

```
{this.dataSource,to=VM.sendListVM.mailDocsVM.dbSet,mode=OneWay}
```

and even shorter (*omitting the mode attribute, because OneWay is default*):


```
{this.dataSource,to=VM.sendListVM.mailDocsVM.dbSet}
```

If we used this binding expression inside a data template, then the path evaluation would start from the data context which is assigned to a data template (*data templates have datacontext property*), and not from an application instance.

Template's datacontext property is assigned when an instance of the template is created and can be later reassigned with a new object or be set to null value (*effectively changing the binding's implicit source property value*). If we explicitly provided the source in the binding's expression, then even if it was used inside the template, the source would have been fixed (*explicit*), and would not change if the template's datacontext property is changed (*and the path evaluation would always start from an application's instance*).

Above, there were the examples of shortcut style of writing of the data binding expression, but we can write previous binding expression in another way:

```
{targetPath =dataSource,sourcePath=VM.sendListVM.mailDocsVM.dbSet}
```

because `this.dataSource` is semantically equivalent to the `targetPath =dataSource` and `to=VM.sendListVM.mailDocsVM.dbSet` is equivalent to the `sourcePath=VM.sendListVM.mailDocsVM.dbSet`.

In the data binding expressions, instead of `=` sign (*which separates the name and the value*), can be equally used `:` sign, such as:

```
{targetPath:dataSource,sourcePath:VM.sendListVM.mailDocsVM.dbSet}
```

It is just a matter of personal preference which of the signs to use, `=` or `:`.

The data binding instances are created not only on the startup of the application, but they can be created later in time. It can happen when some controls create templates during their life cycle. The templates can include data binding expressions, and they are evaluated only when the templates are created.

For example, when a `DataGrid` is data bound to a datasource (*or the dataSource is refreshed*), the grid creates data cells for each grid's data row. The grid's `DataCell` can have the templated data content (*cells in which content is defined by data templates*), and when the template instances are created then the data bindings on the template elements are evaluated. Later, when the created template instances are destroyed (*when a row in DataGrid is removed*), the data bindings are also destroyed with the template's instance.

The Data Bindings can use converters to convert values from the source to the target and vice versa.

an example of a converter definition (javascript code):

```
var UppercaseConverter = global.getType("BaseConverter").extend({
    convertToSource:function (val, param, dataContext) {
        if (utils._is.String(val))
            return val.toLowerCase();
        else
            return val;
    },
});
```

```

convertToTarget:function (val, param, dataContext) {
    if (utils._is.String(val))
        return val.toUpperCase();
    else
        return val;
}
}, null, function (obj) {
    app.registerConverter('uppercaseConverter', obj);
});

```

an example of using a converter (javascript code):

```

app.getType('Binding').create({ sourcePath: 'testProperty',
targetPath: 'testProperty2',
source: app.VM.testObject1,
mode: 'OneWay',
target: app.VM.testObject1,
converter: app.getConverter('uppercaseConverter'),
converterParam: null
});

```

an example of using a converter declaratively:

```

<input type="radio" name="radioItem"
data-bind="{this.checked,to=radioValue,mode=TwoWay,converter=radioValueConverter,
converterParam='radioValue3',source=VM.demoVM}" />

```

A special case is when a converter's methods (*any of it*) return **undefined** value. In this case the data binding ignores the value returned by the converter and does not updates a source or a target,

an example of the definition of a radioValue converter

```

var radioValueConverter = global.getType('BaseConverter').extend({
    convertToSource:function (val, param, dataContext) {
        //returning undefined here means, we don't want to set the value on the source
        return !!val?param:undefined;
    },
    convertToTarget:function (val, param, dataContext) {
        return (val == param)?true:false;
    }
}, null, function (obj) {
    app.registerConverter('radioValueConverter', obj);
});

```

In the above example, the radioValue converter returns **undefined** value when a value from the target is **false**. This prevents it from updating the value on the source in this case. The scenario is very helpful when binding several radio buttons or checkboxes to one source's property (*see collections demo, only one radiobutton updates the source - which one is checked*).

2.5 The Command type

Commands provide the means for declarative execution of view model's methods. Some of the element views provide the means to bind their comands to the implementations of the command in view models (*for example, button's element view*).

an example of binding a command to the button's element view:

```

<input type='button' value=' Upload file ' data-bind="{this.command,to=uploadCommand,mode=OneWay}"/>

```

In the above example, a button's element view exposes command property which is data bound to a view model's command implementation. When the button is clicked, it triggers an execution of the action (*typically, a method on a view model*) for the uploadCommand.

an example of the command's definition (javascript code):

```
this._uploadCommand = app.getType('Command').create(function (sender, param) {
    try {
        self.uploadFiles(self._fileEl.files);
    } catch (ex) {
        self._onError(ex, this);
    }
}, self, function (sender, param) {
    return self._canUpload();
});
```

The first command's constructor parameter is a callback function (*an action*), which is invoked when a command is executed by the UI element.

The second parameter is an object which defines the `this` context of the command's action (*inside the action, it will be `this` property value*).

The third parameter is callback function which returns bool value, which determines if the command is currently in an enabled or in a disabled state.

When we want to reevaluate the condition of when the command is disabled or enabled, we invoke the command's `raiseCanExecuteChanged` method, as in the example:

```
this._uploadCommand.raiseCanExecuteChanged();
```

The command's action function accepts two arguments – the first is the sender, which is the invoker of the command (*typically element view instance*), the second argument is command parameter which can be assigned by the data binding.

an example of a HTML markup inside the template (javascript code):

```
<!--bind the commandParameter to current datacontext, which is here the product entity-->
<span data-name="upload"
data-bind="{this.command,to=dialogCommand,source=VM.uploadVM}{this.commandParam}"
data-view="name='link-button',options={text: Upload Thumbnail,tip='click me to upload product thumbnail photo'}"></span>
```

P.S.- `{this.commandParam}` expression binds `commandParam` property on the element view to the current template's datacontext .

Using a command parameter in the command's action (javascript code):

```
this._dialogCommand = app.getType('Command').create(function (sender, param) {
    try {
        //using command parameter to get a product item
        self._product = param;
        self.id = self._product.ProductID;
        self.showUploadDialog('uploadTemplate',450, 200, 'Upload product thumbnail');
    } catch (ex) {
        self._onError(ex, this);
    }
}, self, function (sender, param) {
    return true;
});
```

2.6 Element views

Element views are the wrappers over HTML DOM element's and can also wrap other controls which you want to use in declarative style. They expose properties which can be data bound declaratively in the HTML markup.

P.S. - You can not directly databind a HTML DOM element's property, because you can only databind properties of an object derived from the framework's BaseObject. Element views are objects which all are derived from the BaseObject, so they can expose properties which can be databound.

When an element view is created, its constructor accepts the HTML DOM element, and the options. Without options an element view uses its default values.

For example, the `StackPanelElView` uses the options to determine how it can be displayed - the horizontal or vertical display of the view. `TextBoxElView` uses the `updateOnKeyUp` option, to decide when to update binding's source – when a textbox loses the focus (*the default value*) or when a keyup event occurs.

```
<!--without the updateOnKeyUp option, the value is updated only when the textbox loses the focus-->
<input type="text" data-bind="{this.value,to=testProperty,mode=TwoWay,source=VM.testObject1}"
  data-view="options:{updateOnKeyUp=true}" />
```

The above `data-view` attribute expression provides only the options, but we can also explicitly provide view name and therefore to select which type of the element view will be created for a DOM element, as in the example:

a HTML markup which uses data-view attribute to provide the view name:

```
<span data-bind="{this.command,to=expanderCommand,mode=OneWay,source=VM.headerVM}"
  data-view="name=expander"></span>
```

When data binding expressions are evaluated by the application's code, the code obtains element view using `getElementView` application's method. This method checks if the element view has been already been created for this HTML DOM element. If there's no element view already created, then the code checks for `data-view` attribute on the DOM element, and if it exists, the method tries to get the name of element view and create it explicitly by the name. If the name of element view is not provided explicitly, the method tries to find the default element view for the DOM element tag name. For example, for `<input type='text'/>` tag, the default element view is the `TextBoxElView`, but if you provided explicit name you could override that selection.

Registration of the element views (javascript code):

```
var ExpanderElView = CommandLinkElView.extend({
    ... irrelevant code here
},
{
    ... irrelevant code here
}, function (obj) {
    //registration of element's view type
    app.registerType('ExpanderElView', obj);
    //registering element's view by unique name
    app.registerElView('expander', obj);
});

var ButtonElView = CommandElView.extend({
    ... irrelevant code here
},
```

```

{
    ... irrelevant code here
},
function (obj) {
    app.registerType('ButtonElView', obj);
    //registering element's view by tag's name, so for the button it will be a default element view
    app.registerElView('input:button', obj);
    app.registerElView('button', obj);
});

```

An element view can be simple, only exposing several properties of a DOM element (*like [TextBoxElView](#)*) and can be also complex (*like [GridElView](#)*).

[jRIApp](#) framework has [elview](#) module which registers several predefined ready to use element views:

[BaseElView](#) – base element view type, which provides support for the display of validation errors, and also defines several properties for all descendants of this type. The BaseElView and all its descendants can accept [tip](#) and [css](#) options. The first sets a tooltip to the wrapped DOM element, the second adds a css class to the element at the moment when the element view is created.

BaseElView's properties:

Property	IsRead Only	Description
app	Yes	application instance, which created this element's view
document	Yes	exposes the window's document property, for easier access in view's code
\$el	Yes	jQuery wrapper of the DOM element
el	Yes	DOM element
uniqueID	Yes	unique id which can be used as namespace, for event's subscription inside element's view code (<i>in constructor and methods</i>)
isVisible	No	bool value, determines DOM element visibility on the page
propChangedCommand	No	command (<i>typically, defined in view model</i>) for property change notification. For example, GridElView invokes this command when element view's grid property changes. So, the view model can obtain an instance of the element view through this mechanism.
toolTip	No	valid HTML string which can be provided for display over element view's DOM element.
css	No	css class which can be provided for element's view DOM element. It is usefull to change display style of the element based on the data bound data.
validationErrors	No	internally used by the framework. The data bindings can set this property, for the element view's validation error display.

EditableEiView – a descendant of **BaseEiView**. It is not used directly, but is used as the base class for several element views (*TextBoxEiView*, *TextAreaEiView*, *CheckBoxEiView*, *RadioEiView*, *SelectEiView*). It adds a property **isEnabled** to all of its descendants.

TextBoxEiView – it is a wrapper around `<input type="text"/>` element. Besides inherited properties, it exposes a **value** property, which exposes text value of the DOM element. The default behaviour of this view is to update the value when the textbox loses the focus. It can be tweaked by using the **updateOnKeyUp** option, so the value is changed on every keyup event.

```
<input type="text" data-bind="{this.value,to=testProperty,mode=TwoWay,source=VM.testObject1}" data-view="options:{updateOnKeyUp=true}" style="width:250px"/>
```

TextAreaEiView – it is a wrapper around `<textarea />` element. It is a descendant of the **TextBoxEiView**, so it also has the **value** property (*to get or set text*). Besides it also has **rows**, **cols**, **wrap** properties.

```
<textarea data-bind="{this.value,to=testProperty,mode=TwoWay,source=VM.testObject1}" rows="10" cols="40" wrap="soft"></textarea>
```

CheckBoxEiView – it is a wrapper around `<input type="checkbox"/>` element. It exposes **checked** property of the checkbox DOM element.

```
<input type="checkbox" data-bind="{this.checked,to=boolProperty,mode=TwoWay,source=VM.testObject1}" />
```

RadioEiView – it is a wrapper around `<input type="radio"/>` element. It exposes **checked** property of the radio DOM element. Typically the databinding expression for radio element uses converter, so only one radio button (*which is checked*) updates the source.

```
<input type="radio" name="radioItem" data-bind="{this.checked,to=radioValue,mode=TwoWay,converter=radioValueConverter,converterParam=radioValue2,source=VM.dimoVM}" />
```

CommandEiView - a descendant of the **BaseEiView**. It is not used directly, but is used as a base class for several element views (*ButtonEiView*, *CommandLinkEiView*). It adds two properties **command** and **commandParam** to all of its descendants. The descendants internally use `invokeCommand` method to invoke command (*typically, when the button or the link is clicked*).

ButtonEiView - is a descendant of the **CommandEiView**. It is a wrapper around `<button/>` or `<input type="button" />` element. It exposes **value**, **text**, **isEnabled** properties of a button element.

```
<button data-name="btnCancel" data-bind="{this.text,to=txtCancel,source=localizable.TEXT}"></button>
```

P.S.- **data-name** attribute is used to find element's view in template instance by the name. It can be used in user code, to select only the needed elements. It is an alternative to the `name` attribute, because in HTML5 some elements can not have the `name` attribute, but **data-name** can be used universally.

AncorButtonEiView - is a descendant of the **CommandEiView**. It is the wrapper around the anchor element. Clicking on the link invokes the command. The link can display a text or an image (*which can be determined by the options*). It exposes **imageSrc**, **html**, **text**, **href**, **isEnabled** properties of the anchor element.

```
<a data-name="btn-load" class="btn btn-info btn-small" data-bind="{this.command,to=uploadCommand}" data-view="options={tip='Click to upload a file'}">Upload</a>
```

ExpanderElView - is a descendant of the **AncorButtonElView**. It is the wrapper around the anchor element. It adds an image to the anchor, which changes its appearance depending on the expanded or the collapsed state. When the image is clicked it invokes the command to propagate the event to the view model. The element view is registered by the name **'expander'**.

```
<a href="#" id="expander" style="margin-left:2px;" data-bind="{this.command,to=expanderCommand,mode=OneWay,source=VM.headerVM}" data-view="name=expander"></a>
```

```
//the definition of the command in the view model
this._expanderCommand = app.getType('Command').create(function (sender, param) {
    if (sender.isExpanded) {
        self.expand();
    }
    else
        self.collapse();
}, self, null);
```

TemplateElView - is a descendant of the **CommandElView**. It is a special case element view. It has no other properties besides the inherited from the base type. One property which is important for this element view is the **command** property. This view is used only inside the data templates to get notifications in user defined view models about when the data template's instance is created or is going to be destroyed. The command property must be bound only to the fixed source (*the source should be provided in the data binding expression explicitly*). The command is triggered when the data template is loaded or is starting to unload. This can be used, to access the template's html elements (*you can assign some event handlers to them or add some attributes*). This element view is registered by the name **'template'**.

an example of a template which uses **TemplateElView** (see *dataGrid demo*):

```
@upload thumbnail dialog template*@
<div id="uploadTemplate" data-bind="{this.command,to=templateCommand,source=VM.uploadVM}" data-view="name=template">
<!--dummy form action to satisfy HTML5 specification-->
<form data-name="uploadForm" action="#">
    <div data-name="uploadBlock">
        <input data-name="files-to-upload" type="file" style="visibility:hidden;" />
        <div class="input-append">
            <input data-name="files-input" class="span4" type="text">
            <a data-name="btn-input" class="btn btn-info btn-small"><i class="icon-folder-open"></i></a>
            <a data-name="btn-load" class="btn btn-info btn-small" data-bind="{this.command,to=uploadCommand}" data-view="options={tip='Click to upload a file'}">Upload</a>
        </div>
        <span>File info:</span><text>&nbsp;</text><div style="display:inline-block" data-bind="{this.value,to=fileInfo}"></div>
        <div data-name="progressDiv">
            <progress data-name="progressBar" class="span4" value="0" max="100"></progress><span data-name="percentageCalc"></span>
        </div>
    </div>
</form>
</div>
```

an example of the command definition bound to the **TemplateElView**'s command property (see *UploadThumbnailVM in gridDemo.js*):


```

this._templateCommand = app.getType('Command').create(function (sender, param) {
    try {
        var t = param.template, templEl = t.el, $ = global.$,
            fileEl = $('input[data-name="files-to-upload"]', templEl);
        if (fileEl.length == 0)
            return;
        //when the template is loaded find the elements inside the template and add the event handlers to them
        if (param.isLoaded){
            fileEl.change(function (e) {
                $('input[data-name="files-input"]', templEl).val($(this).val());
            });
            $('*[data-name="btn-input"]', templEl).click(function(e){
                e.preventDefault();
                e.stopPropagation();
                fileEl.click();
            });
        }
        else{
            //when the template is unloaded remove the event handlers
            fileEl.off('change');
            $('*[data-name="btn-input"]', templEl).off('click');
        }
    } catch (ex) {
        self._onError(ex, this);
    }
}, self, function (sender, param) {
    return true;
});
});

```

SpanElView - It is a wrapper around `` element. It is used to data bind some text or html content to a span element's content. It exposes **value**, **text**, **html**, **color**, **fontSize** properties which can be data bound. The **value** property is semantically the same as the **text** property.

*The **html** property should be used carefully, because it inserts a HTML content inside an element. It should not be used to display the user input without first checking the content to prevent XSS attacks.*

```

<span data-bind="{this.value,to=testProperty1,mode=OneWay,source=VM.testObject1}"></span>
<span data-bind="{this.text,to=testProperty2,mode=OneWay,source=VM.testObject1}"></span>
<span data-bind="{this.html,to=testProperty3,mode=OneWay,source=VM.testObject1}"></span>

```

DivElView - a descendant of the **SpanElView**. It is a wrapper around `<div/>` element. Besides the properties inherited from the **SpanElView**, it adds **borderColor**, **borderStyle**, **width**, **height** properties, which can be data bound.

```

<div data-bind="{this.html,to=testProperty2,mode=OneWay,source=VM.testObject1}"></div>

```

ImgElView - It is a wrapper around `` element. It is used to data bind image's **src** property.

```

<img data-bind="{this.src,to=srcProperty,source=VM.testObject1}" />

```

BusyElView - It is a wrapper around a html element (*typically, div element*). It exposes the **isBusy** property.

It is used to display an animated loader gif image over a html element to which it is attached. The element view is registered by the name '**busy_indicator**'.

```

<div data-bind="{this.isBusy,to=isLoading}" data-view="name=busy_indicator">
... some html content
</div>

```

GridView - It is a wrapper around `<table/>` element. It is used to attach to a table DOM element the logic and the markup of the **DataGrid** control. It exposes the **dataSource** and the **grid** properties. It is used to display the datagrid with the data obtained from the data service. The DataGrid control has a lot of features which can be better understood from the demo application.

```
<table data-app='default' data-name="gridProducts"
data-bind="{this.dataSource,to=dbSet,source=VM.productVM}{this.propChangedCommand,to=propChangeCommand,
source=VM.productVM}"
data-view="options={wrapCss:productTableWrap,containerCss:productTableContainer,headerCss:productTableHeader,
rowStateField:IsActive,isHandleAddNew:true,isCanEdit:true,editor:{templateID:productEditTemplate,width:550,height:650,
submitOnOK:true,title:'Product editing'},details:{templateID:productDetailsTemplate}}">
  <thead>
    <tr>
      <th data-column="width:35px,type:row_expander"/>
      <th data-column="width:50px,type:row_actions"/>
      <th data-column="width:40px,type:row_selector,rowCellCss:selectorCell,colCellCss:selectorCol"/>
      <th data-column="width:100px,sortable:true,title:ProductNumber" data-
content="fieldName:ProductNumber,css:{displayCss:'number-display',editCss:'number-edit'}"/>
      <th data-column="width:25%,sortable:true,title:Name" data-content="fieldName:Name"/>
      <th data-column="width:90px,title:Weight,sortable:true" data-content="fieldName:Weight"/>
      <th data-column="width:15%,title=CategoryId,sortable:true,sortMemberName=ProductCategoryId" data-content=
"fieldName=ProductCategoryId,lookup:{dataSource=dbContext.dbSets.ProductCategory,valuePath=ProductCategoryId,textPath=N
ame}"/>
      <th data-column="width:100px,sortable:true,title='SellStartDate'" data-content="fieldName=SellStartDate"/>
      <th data-column="width:100px,sortable:true,title='SellEndDate'" data-content="fieldName=SellEndDate"/>
      <th data-column="width:90px,sortable:true,title='IsActive'" data-content="fieldName=IsActive"/>
      <th data-column="width:10%,title=Size,sortable:true,sortMemberName=Size" data-
content="template={displayID=sizeDisplayTemplate,editID=sizeEditTemplate}"/>
    </tr>
  </thead>
  <tbody></tbody>
</table>
```

PagerView - is a wrapper around `<div/>` element. It is used to attach to the div element the logic and the markup of the **Pager** control. It exposes the **dataSource** and the **pager** properties. The element view is registered by the name 'pager'.

```
<!--pager-->
<div data-bind="{this.dataSource,to=dbSet,source=VM.productVM}"
data-view="name=pager,options={sliderSize:20,hideOnSinglePage=false}"></div>
```

StackPanelView - is a wrapper around `<div/>` element. It is used to attach to the div element the logic and the markup of the **StackPanel** control. It exposes the **dataSource** and the **panel** properties. It can be used to show horizontally or vertically stacked panels (*which use template*) on a page. The element view is registered by the name 'stackpanel'.

```
<!--using stackpanel for horizontal list view-->
<div style="width:100%; overflow:auto;"
data-bind="{this.dataSource,to=custAdressView,source=VM.customerVM.custAddrVM}"
data-view="name=stackpanel,options={templateID:stackPanelItemTemplate,orientation:horizontal}">
</div>
```

SelectView - It is a wrapper around `<select/>` element. It is used to attach to the select element the logic of the **ListBox** control. It exposes **dataSource**, **selectedValue**, **selectedItem**, and **listBox** properties. It is used for the data binding of a collection of items to the select DOM element's options.

```
<select size="1" style="width:220px"
data-bind="{this.dataSource,to=ParentCategories,source=VM.productVM.filter}
{this.selectedValue,to=parentCategoryId,mode=TwoWay,source=VM.productVM.filter}"
data-view="options:{valuePath=ProductCategoryId,textPath=Name}"></select>

<select id="prodSCat" size="1" class="span2"
data-bind="{this.dataSource,to=filter.ChildCategories}{this.selectedValue,to=filter.childCategoryId,mode=TwoWay}
```

```
{this.selectedItem,to=filter.selectedCategory,mode=TwoWay}{this.toolTip,to=filter.selectedCategory.Name}"
data-view="options:{valuePath=ProductCategoryID,textPath=Name}"></select>
```

DataFormElView - It is a wrapper around `<div/>` or `<form/>` element. It attaches to them the logic of the **DataForm** control for managing the datacontext (*see more info in DataForm control section of the guide*). It exposes the **dataSource** and the **form** properties. It is used to show the data for an editing and a display purposes. The element view is registered by the name '**dataform**'.

```
<div id="productEditTemplate">
//data form - could use form tag here instead of div. it does not matter because we don't use form submit.
//data form - used here only for data bindings - using data content's attribute , and it attaches dataContext's scope
<div style="width: 470px; font-size: 12px;" data-bind="{this.dataContext,mode=OneWay}" data-view="name=dataform">
  <table style="width: 95%">
    <thead>
      <tr><th>Field NAME</th><th>Field VALUE</th></tr>
    </thead>
    <tbody>
      <tr>
        <td>
          ID:
        </td>
        <td>
          <span data-content="fieldName:ProductID"></span>
        </td>
      </tr>
      <tr>
        <td>
          Name:
        </td>
        <td>
          <span data-content="fieldName:Name,css:{displayCss:'name-display',editCss:'name-
edit'},multiline:{rows:3,cols:20,wrap:hard}">
            </span>
        </td>
      </tr>
      <tr>
        <td>
          ProductNumber:
        </td>
        <td>
          <span data-content="fieldName:ProductNumber"></span>
        </td>
      </tr>
      <tr>
        <td>
          Category:
        </td>
        <td>
          <span data-
content="fieldName=ProductCategoryID,lookup:{dataSource=dbContext.dbSets.ProductCategory,valuePath=ProductCategoryID,te
xtPath=Name},css:{editCss:'listbox-edit'}">
            </span>
        </td>
      </tr>
      <tr>
        <td>
          Model:
        </td>
        <td>
          <span data-
content="fieldName=ProductModelID,lookup:{dataSource=dbContext.dbSets.ProductModel,valuePath=ProductModelID,textPath=N
ame},css:{editCss:'listbox-edit'}">
            </span>
        </td>
      </tr>
      <tr>
        ... the REST of the MARKUP HERE OMMITED
      </tr>
    </tbody>
  </table>
</div>
</div>
```

TabsEIVView - It is a wrapper around `<div/>` element. It attaches to it the JQuery UI tabs plugin's logic for managing the tabs content. It exposes **tabsEventCommand** property. It is used to display the tabs in JQuery UI's style. The element view is registered by the name **'tabs'**.

The **tabsEventCommand** is used to get notifications on tabs events (*like when a tab was selected, added, showed, enabled, disabled, removed, loaded*) and handle them in our viewmodel (*but you should bind this command even if you don't need to handle events, because element views are created when data binding expression is evaluated, and if there's no bindings to the element view then its instance will not be created. You should bind at least one property, in order that element view's instance is created!*).

// an example of tabs markup on the html page

```
<div id="productDetailsTemplate">
  <!--using tab's element view inside the template-->
  <div data-name="tabs" style="margin: 5px; padding: 5px; width: 95%;"
    data-bind="this.tabsEventCommand,to=tabsEventCommand,source=VM.productVM"
    data-view="name=tabs">
    <div id="myTabs">
      <ul>
        <li><a href="#a">Tab 1</a></li>
        <li><a href="#b">Tab 2</a></li>
      </ul>
      <div id="a">
        <span>Product Name: </span>
        <input type="text" style="color: Green; width: 220px; margin: 5px;" data-bind="{this.value,to=Name,mode=TwoWay}" />
        <br />
        <a class="btn btn-info btn-small" data-
bind="{this.command,to=testInvokeCommand,source=VM.productVM}{this.commandParam}"
  data-view="options={tip='Invokes method on the server and displays result'}">
          Click me to invoke service method</a>
      </div>
      <div id="b">
        <img style="float:left" data-bind="{this.id,to=ProductID}{this.fileName,to=ThumbnailPhotoFileName}" alt="Product
Image" src="" data-view="name=fileImage,options={baseUri:@Url.RouteUrl("Default", new { controller = "Download", action =
"ThumbnailDownload" } )}" /><br />
        <div style="float: left; margin-left: 8px;">
          click to download the image: <a class="btn btn-info btn-small" data-
bind="{this.text,to=ThumbnailPhotoFileName}{this.id,to=ProductID}" data-
view="name=fileLink,options={baseUri:@Url.RouteUrl("Default", new { controller = "Download", action =
"ThumbnailDownload" } )}">
            </a>
        </div>
        <div style="clear: both; padding: 5px 0px 5px 0px;">
          <!--bind commandParameter to current datacontext, that is product entity-->
          <a class="btn btn-info btn-small" data-name="upload" data-
bind="{this.command,to=dialogCommand,source=VM.uploadVM}{this.commandParam}"
  data-view="options={tip='click me to upload product thumbnail photo'}">Upload product
            thumbnail</a>
        </div>
      </div>
    </div>
  <!--myTabs-->
</div>
```

// tabs events command handler in our view model

```
this._tabsEventCommand = app.getType('Command').create(function (sender, param) {
  var index = param.args.index, tab = param.args.tab, panel = param.args.panel;
  // the next line is commented in the demo code, you can uncomment it to see it in action
  alert('event: ' + param.eventName + ' was triggered on tab: ' + index);
}, self, null);
```

Custom built element views - Besides the predefined element views, it is easy to add custom element views.

For example, in the demo application there has been added several custom element views, such as **AutoCompleteEIVView**, **DownloadLinkEIVView**, **FileImageEIVView** – they all have been defined in custom modules.

```

<!--using a custom element view for the display of the product image-->
<img data-bind="{this.id,to=ProductID}{this.fileName,to=ThumbnailPhotoFileName}"
alt="Product Image"
src=""
data-view="name=fileImage,options={baseUri:'@Url.RouteUrl("Default", new { controller = "Download", action =
"ThumbnailDownload" })}"/>

```

P.S - One property *propChangedCommand* introduced in the *BaseElView* needs more explanations.

You can use this command to get instances of element views in your code (typically view models), if you get element view instance then you can get any property value on the element view.

It is invoked when a property on the element view had been changed. So in the handler for this command we can get reference to the element view instance and to the changed property name.

For example, it can be used, to obtain references to the datagrid's instance when the datagrid is created by the element view (and other properties of the element view). And when we have grid's instance we can attach event handlers to it.

an example of binding expression (used in datagrid's demo) to bind *propChangedCommand* to the handler in our viewmodel's instance:

```

{this.propChangedCommand,to=propChangeCommand,source=VM.productVM}

//our product view model defines handler for the command
//we can obtain grid instance from the sender - (the sender is the element view - GridElView)
this._propChangeCommand = app.getType('Command').create(function (sender, data) {
    if (data.property==='*' || data.property==='grid'){
        if (self._dataGrid === sender.grid)
            return;
        self._dataGrid = sender.grid;
    }
    //example of binding to dataGrid events
    if (!self._dataGrid){
        self._dataGrid.addHandler('page_changed', function(s,a){
            self._onGridPageChanged();
        }, self.uniqueID);
        self._dataGrid.addHandler('row_selected', function(s,a){
            self._onGridRowSelected(a.row);
        }, self.uniqueID);
        self._dataGrid.addHandler('row_expanded', function(s,a){
            self._onGridRowExpanded(a.old_expandedRow, a.expandedRow, a.isExpanded);
        }, self.uniqueID);
        self._dataGrid.addHandler('row_state_changed', function(s,a){
            if (!a.val){
                a.css = 'rowInactive';
            }
        }, self.uniqueID);
    }
}, self, null);

```

2.7 Data templates

The data templates are pieces of the html markup which can be used in RIA applications by cloning their structure and adding dynamic data to them by data binding. They are typically used by data template aware controls. The framework contains several built-in controls, which are data template aware: [DataEditDialog](#), [DataGrid](#), [StackPanel](#).

The data templates have the `id` attribute for referencing them in the options of data aware controls. They are defined in a special section on the html page (*but there can be several of such sections*). The section has a special css class `ria-template`, which makes the section invisible on the page and distinguishable from the other non templates containing sections.

an example of a template's section (html markup):

```
@*invisible section to hold data templates*@
<section class="ria-template">
  @*data template – id attribute is mandatory*@
  <div id="stackPanellItemTemplate" class="stackPanellItem" style="width:170px;">
    <fieldset>
      <legend><span data-bind="{this.value,to=radioValue}"></span></legend>
      Time:&nbsp;
      <span data-bind="{this.value,to=time,converter=dateTimeConverter,converterParam='HH:mm:ss'}"></span>
    </fieldset>
  </div>
  @*HERE can be more data templates ...*@
</section>
```

The data template aware controls create instances of the data templates add the newly created data template instance to the HTML DOM tree, and can set template's `datacontext` property. The data template's `datacontext` can be changed at any time on the data template's instance (*for example, bind another data item to the same data template*).

an example of a template creation inside a control's code (javascript code):

```
//example of template creation inside DataDialog method code
_createTemplate:function (dcxt) {
    var T = app.getType('Template'), t = T.create(this._templateID);
    // template's instance created here in a disabled state - (it disables all the databindings inside the template)
    // we don't want the databindings to be enabled until the dialog is shown
    // it conserves resources when the dialog is hidden.
    t.isDisabled = true;
    t.dataContext = dcxt;
    return t;
}
```

2.8 Data contents

The data templates are useful for displaying data in one state, but there are cases when the data can be in different states, such as: the edit and the display state, and for each state we need different controls or templates. For example, when in the edit state then the controls should be editable (*textboxes, textareas, selects*), and in the display state they should be readonly (*spans and divs*).

This problem can be solved by the use of the data contents.

The `Data content` can display data in two different modes - the editing and the display modes.

The data-content attribute can be used only inside the data forms and the datagrids (*to define columns*).

When the data content is bound to an item which has `isEditing` property (*the entities have them*), it starts to monitor (*observe*) this property for changes. When `isEditing` property changes then the appearance of the data content is also changed. Typically, the data contents are bound to collection items.

an example of using data contents inside the data form:

```
<div id="orderDetEditTemplate">
  <div style="width:100%" data-bind="{this.dataContext,mode=OneWay}" data-view="name=dataform">
    <table style="width:95%; border:none; table-layout:fixed; background-color:transparent;">
      <colgroup>
        <col style="width:225px;border:none;text-align:left;" />
        <col style="width:100%;border:none; text-align:left;" />
      </colgroup>
      <tbody>
        <tr><td>ID:</td><td><span data-content="fieldName:SalesOrderDetailID"></span></td></tr>
        <tr><td>OrderQty:</td><td><span data-content="fieldName:OrderQty,css:{editCss:'qtyEdit'}"></span></td></tr>
        <tr><td>Product:</td><td>
          <span data-content="template={displayID=productTemplate1,editID=productTemplate2}"></span>
        </td></tr>
        <tr><td>UnitPrice:</td><td><span data-content="fieldName:UnitPrice"></span></td></tr>
        <tr><td>UnitPriceDiscount:</td><td><span data-content="fieldName:UnitPriceDiscount"></span></td></tr>
        <tr><td>LineTotal:</td><td><span data-content="fieldName:LineTotal,readonly:true"></span></td></tr>
      </tbody>
    </table>
  </div>
</div>
```

The data content can be used in two ways:

- 1) Bind directly to a field name.
- 2) Use the data templates (*one can for the display and the other for the edit mode*).

The first option to bind directly:

is more simple, but the way it is rendered was predefined by the field's data type. For example, when the field's data type is the text, then in the display mode the data content is rendered as the text inside a `` tag, and when in the edit mode it is rendered as an `<input type="text"/>`. All these data content's types derived from `BindingContent` type. Currently, the framework has: `DateContent`, `BoolContent`, `DateTimeContent`, `NumberContent`, `StringContent`, `MultyLineContent`, and `LookupContent` types.

The type for the creation of the instance of the data content is mainly determined by the data type of the field (*Number, String, Bool, Date*) to which the data content is bound. The decision is made by the `BindingContentFactory`, which is used to create the content types in the application. But this decision can be tweaked by specifying the options for the data content.

an example of specifying the multiline option for the data content:

```
<span data-content="fieldName:Comment,multiline:{rows:3,cols:20,wrap:hard}"></span>
```

an example of specifying the lookup option for the data content:

```
<span data-content="fieldName:BillToAddressID,lookup:{dataSource=VM.customerVM.custAdressView,
valuePath=Address.AddressID,textPath=Address.AddressLine1,css:{editCss:'listbox-edit'}"></span>
```

also you can use in a data content the `readOnly` option to ensure that it will only be in display mode.

```
<th data-column="width:110px,sortable:true,title:OrderDate" data-content="fieldName:OrderDate,readonly:true" />
```

and you also can use in a data content any navigation properties like in the example (*Here the datacontext is OrderDetail and Product is a navigation property on OrderDetail entity*).


```
<th data-column="width:100%,title:Product" data-content="fieldName:Product.Name,readonly:true" />
```

The second option is the templated data content:

is more versatile than a field directly data bound to a data content.

The data templates can be created in the exact way the user wish it to be rendered, and the data template can include several fields data bound inside the template. The templated content is defined by the [TemplateContent](#) type.

an example of defining the templated data content:

```
<span data-content="template={displayID=salespersonTemplate1,editID=salespersonTemplate2},  
css:{displayCss:'custInfo',editCss:'custEdit'}"></span>
```

P.S. - *editID* can be ommited, and the data content will only display one template in each state.

2.9 Controls

DataGrid:

The [DataGrid](#) is a control for attaching logic to the table HTML element.

DataGrid has the following features:

- 1) Data binding to collection datasource
- 2) Inline or Dialog data editing mode
- 3) Data paging (*by using the pager control*)
- 4) Sorting the data by column clicking
- 5) Special column types like - row selector, row actions, row expander
- 6) Advanced column definition with templated columns
- 7) Visual row state (*change in row display*) based on the field's value
- 8) Fixed column headers (*always visible when data is scrolled*)
- 9) Row scrolling using the keyboard keys - up, down, left, right keys
- 10) Row selection (*when row selector column is present*) by the keyboard's space key
- 11) Going to the next or previous page display by using keaboard's pageup or pagedown keys.

The DataGrid – is defined in the [datagrid](#) module. In the module there are several types which relevant to the DataGrid functionality (*BaseCell, DataCell, ExpanderCell, ActionsCell, RowSelectorCell, DetailsCell, Row, DetailsRow, BaseColumn, DataColumn, ExpanderColumn, ActionsColumn, RowSelectorColumn, DataGrid*). The DataGrid instantiation is only possible in the code. In order to avoid using the code and get the benefits of the declarative style, the control has the specific element view - [GridElView](#).

an example of the data grid definition on the page:

```
<table data-app='default' data-name="gridCustomers" data-  
bind="{this.dataSource,to=dbSet,source=VM.customerVM}{this.propChangedCommand,to=propChangeCommand,s  
ource=VM.customerVM}"  
data-view="options={wrapCss:customerTableWrap,isHandleAddNew:true,  
editor:{templateID:customerEditTemplate,width:500,height:550,title:'Customer  
editing'},details:{templateID:customerDetailsTemplate}}">  
  <thead>  
  <tr>
```

```

<th data-column="width:35px,type:row_expander"/>
<th data-column="width:50px,type:row_actions"/>
<th data-column="width:90%,sortable:true,title:Title" data-content="fieldName:Title"/>
<th data-column="width:90%,sortable:true,title:Customer
Name,sortMemberName=LastName;MiddleName;FirstName" data-content="fieldName:Name"/>
<th data-column="width:90%,sortable:true,title:CompanyName" data-content="fieldName:CompanyName"/>
<th data-column="width:90%,sortable:true,title:SalesPerson" data-content="fieldName:SalesPerson"/>
</tr>
</thead>
<tbody></tbody>
</table>

```

The columns in the datagrid are defined by adding to the <th /> tag [data-column](#) and [data-content](#) attributes.

The Data-column attribute can have [width](#), [title](#), [sortable](#), [rowCellCss](#), [colCellCss](#), [type](#), [sortMemberName](#) options.

The SortMemberName option can contain several field names separated by semicolons, as in:

```

<th data-column="width:200px,title:'FIO',sortable:true,sortMemberName:MCOD;FAM;IM;OT;DR;INFOTYPE" data-content="fieldName:FIO" />

```

If the [type](#) option is omitted, then it is assumed that the column is the data column, which displays fields data (*there can be also the actions or the expander or the row selector columns*).

The data-content attribute specifies the data which will be displayed in the data cell. It can be the templated data, as in:

```

<th data-column="width:60%,title:'PERIOD'"
data-content="template={displayID=monthsTemplate,editID=monthsTemplate}" />

```

or just the default field data, as in:

```

<th data-column="width:75px,title:'MCOD',sortable:true,sortMemberName:MCOD;FAM;IM;OT;DR;INFOTYPE"
data-content="fieldName:MCOD" />

```

Typically, to display the rows in the data grid, it is needed to bind the source of the data to the grid's [dataSource](#) property. Optionally, if there is a need to handle datagrid's events in the view model's code, we can also bind to [propChangedCommand](#) and then in the command's code we can access the instance of the grid element view, and can add event handlers for the datagrid's events.

example of adding the event handlers to the grid in the view model's code:

```

this._propChangeCommand = app.getType('Command').create(function (sender, data) {
    //getting instance of the dataGrid
    //the sender is an element view which wraps the grid control
    if (data.property==='*' || data.property==='grid'){
        if (self._dataGrid === sender.grid)
            return;
        self._dataGrid = sender.grid;
    }

    //example of binding to dataGrid events
    if (!self._dataGrid){
        self._dataGrid.addHandler('page_changed', function(s,a){
            self._onGridPageChanged();
        }, self.uniqueID);
        self._dataGrid.addHandler('row_selected', function(s,a){
            self._onGridRowSelected(a.row);
        }, self.uniqueID);
    }
});

```

```

        self._dataGrid.addHandler('row_expanded', function(s,a){
            self._onGridRowExpanded(a.old_expandedRow, a.expandedRow, a.isExpanded);
        }, self.uniqueID);
        self._dataGrid.addHandler('row_state_changed', function(s,a){
            if (!a.val){
                a.css = 'rowInactive';
            }
        }, self.uniqueID);
    }
}, self, null);

```

The other important piece of the markup, which is used in grid definition, is the **data-view** attribute:

```

data-view="options={wrapCss:productTableWrap,containerCss:productTableContainer,headerCss:productTableHeader,rowStateField:IsActive,isHandleAddNew:true,editor:{templateID:productEditTemplate,width:550,height:650,submitOnOK:true,title:'Product editing'},details:{templateID:productDetailsTemplate}}"

```

The attribute is optional, but by using the **data-view** attribute, we can define the datagrid's options and can specify the detail's template, and we can also provide the options for datagrid's dialog editor. In the options we can also specify the css styles for the table's container, tables header, and the table's wrap. Among the styles the most important is the table's wrap style.

With the table's wrap style, we can add to the table the vertical scrolling for the table's data (*the table body*), and allow the table's header visible even when the table's body is scrolled down.

an example of the overall table markup structure rendered on the page:

```

<!-- overall table container is added when grid's code is attached to the table -->
<div class="ria-table-container productTableContainer">
    <!-- these columns are always on the top of the table.
         they replace the original table's columns, which are invisible
    -->
    <div class="ria-table-header productTableHeader" style="width: 1207px;">
        <div class="ria-ex-column" style="width: 35px; position: relative; top: 0.0666667px;
            left: 1px;">
            <div class="cell-div row-expander">
            </div>
        </div>
        <div class="ria-ex-column" style="width: 50px; position: relative; top: 0.0666667px;
            left: 1px;"><div class="cell-div row-actions"></div>
        </div>
        <div class="ria-ex-column" style="width: 40px; position: relative; top: 0.0666667px;
            left: 1px;">
            <div class="cell-div row-selector selectorCol selected">
                <input type="checkbox">
            </div>
        </div>
        <div class="ria-ex-column" style="width: 100px; position: relative; top: 0.0666667px;
            left: 1px;">
            <div class="cell-div data-column sortable">ProductNumber</div>
        </div>
    <!-- the other columns markup here -->
</div>

<!-- the table is wrapped in the div tag, for scrolling the data -->
<div class="ria-table-wrap productTableWrap">
    <table class="ria-data-table" data-view=" ... " data-bind=" ... "
        data-name="gridProducts" data-app="default" data-elvwkey0="s_10">
        <thead><!-- the real table's columns are invisible --></thead>
        <tbody><!-- table's rows here --></tbody>
    </table>
</div>

```

</div>

an example of the overall table row markup structure rendered on the page:

```
<tr class="row-highlight">
  <td class="row-collapsed row-expander" style="width: 35px;">
    ...
  </td>
  <td class="row-actions cell-div ria-nobr" style="width: 50px;">
    <!-- cells data here-->
  </td>
  <td class="row-selector" style="width: 40px;">
    <!-- cells data here-->
  </td>
  <div class="cell-div ria-content-field selectorCell" data-scope="51">
    <input type="checkbox" data-elvwkey0="s_125" style="opacity: 1;">
  </div>
  </td>
  <td style="width: 100px;">
    <div class="cell-div ria-content-field number-display" data-scope="52">
      <span data-elvwkey0="s_126">FD-2342</span>
    </div>
  </td>
  <td style="width: 25%;">
    <div class="cell-div ria-content-field" data-scope="53">
      <span data-elvwkey0="s_127">Front Derailleur</span>
    </div>
  </td>
  <!-- the other cells-->
</tr>
```

The other DataGrid's options include:

isUseScrollInto - If true, then when using keyboard keys for scrolling the table's data, the active record is positioned on the screen using HTML DOM scrollInto method. The default is true.

isUseScrollIntoDetails - If true, then when expanding the table's details, the details are positioned on the screen using HTML DOM scrollInto method. The default is true.

rowStateField - a name of the field, on which change the grid's **row_state_changed** event is invoked. In the handling of this event can be inspected the fields value, and based on that can be selected the css style for the whole row.

isCanEdit and **isCanDelete** - if the options values are false, then the editing or deleting row in the grid is not allowed. Usefull to override respective values of permissions obtained from data service metadata, if we want to use some grids in readonly mode.

isHandleAddNew - if set to true, then if the datasource which is bound to the grid add new item, then the grid will show data edit dialog for editing this item. The default is false.

The Grid's Dialog (editor) options include:

templateID - name of the template which will be used for the dialog display.

width - the width of the dialog.

height - the height of the dialog.

submitOnOK - if true, then when clicking OK button of the dialog automatically submits changes to the server, and the dialog is not closed until the response from the server confirms successful commit of the changes.

title - the title used in the dialog's header.

the DataGrid with the products data on the page and expanded row details:


DataGrid Demo

Filter

	ProductNumber	Name	Weight	CategoryID	SellStartDate	SellEndDate	IsActive	Size
	ST-1401	All-Purpose Bike Stand		Bike Stands	01.07.2003		<input checked="" type="checkbox"/>	Size:
	CA-1098	AWC Logo Cap		Caps	01.07.2001		<input checked="" type="checkbox"/>	Size:

Tab 1

Tab 2



click to download the image: [UserImage.ashx.jpg](#)

Upload product thumbnail

	CL-9009	Bike Wash - Dissolver		Cleaners	01.07.2003		<input checked="" type="checkbox"/>	Size:
	LO-C400	Cable-Lock		Locks	01-07-2002	30-06-2003	<input type="checkbox"/>	Size:
	CH-0234	Chainus		Chains	01.07.2003		<input checked="" type="checkbox"/>	Size:
	VE-C304-L	Classic Vest, L3		Jerseys	01.07.2003		<input checked="" type="checkbox"/>	Size: M
	VE-C304-M	Classic Vest, M		Vests	01.07.2003		<input checked="" type="checkbox"/>	Size: M
	VE-C304-S	Classic Vest, S		Vests	01.07.2003		<input checked="" type="checkbox"/>	Size: S
	FE-6654	Fender Set - Mountain2		Fenders	01.07.2003		<input checked="" type="checkbox"/>	Size:
	FB-9873	Front Brakes	317,00	Brakes	01.07.2003		<input checked="" type="checkbox"/>	Size:
	FD-2342	Front Derailleur	88,00	Derailleurs	01.07.2003		<input checked="" type="checkbox"/>	Size:
	GL-F410-M	Full-Finger-Gloves,-M		Gloves	01-07-2002	30-06-2003	<input type="checkbox"/>	Size: M
	GL-F410-S	Full-Finger-Gloves,-S		Gloves	01-07-2002	30-06-2003	<input type="checkbox"/>	Size: S

1 2 3 4 5 6 >> Total: 293, Selected: 0

+ New Product

Many To Many Demo

Customer Name

Abel R. Catherine

Abel R. Catherine2

Abercrombie Kim

Abercrombie Kim

Adams Jay

Adams Jay

Adams B. Frances

Adams B. Frances

Agcaolli N. Samuel

Agcaolli N. Samuel

Ahlerin E. Robert

Ahlering E. Robert

Alan A. Stanley

Alan A. Stanley

Alberts E. Amy

Alberts E. Amy

Alcorn L. Paul

Alcorn L. Paul

Alderson F. Gregory

Alderson F. Gregory

Alexander Mary

Alexander Mary

Alexander Michelle

Alexander Michelle

Allen N. Marvin

Allen N. Marvin

Allison J. Cecil

Allison J. Cecil

Alpuerto L. Oscar

Alpuerto L. Oscar

Amland J. Maxwell

Amland J. Maxwell

Antrim J. Ramona

Antrim J. Ramona

Armstrong B. Thomas

Customer Info

ID:

582

Title:

Ms.

FirstName:

Catherine

MiddleName:

R.

LastName:

Abel

Suffix:

CompanyName:

Professional Sales and Service

SalesPerson:

adventure-works\Linda3

Email:

catherine0@adventure-works.com

Phone:

747-555-0171

+ New Customer

Customer Addresses

Type	Address1	Address2	City	State	Region	Zip
Main Office	8713 Yosemite Ct		Bothell	Washington	United States	98011
Main Office	992 St Clair Ave East		Toronto	Ontario	Canada	M4B 1V7
Main Office	99, Rue Saint-pierre		Prot-Rouge	Quebec	Canada	J1E 2T7
Main Office	4400 March Road		Kanata	Ontario	Canada	K2L 1H5
Main Office	5 place Ville-Marie		Montreal	Quebec	Canada	H1Y 2H7
Main Office	32605 West 252 Mile Road, Suite 250		Aurora	Ontario	Canada	L4G 7N6

Manage Addresses

1 2 3 4 5 6 7 8 9 10 >> Total: 848

an example of the datagrid's edit dialog (it's view defined in the template):

Название	Значение
ID:	712
Name:	AWC Logo Cap
ProductNumber:	CA-1098
Color:	Multi
Cost:	6,93
Price:	8,99
Size:	
Weight:	
Category:	Caps
Model:	Cycling Cap
SellStartDate:	01.07.2001
SellEndDate:	

Refresh Ok Cancel

P.S- The *DataGrid* has the feature of changing visual row display depending on some field value of the entity. For example, the *DataGrid* demo uses this feature to display differently rows with *isActive* field value true or false. It is done by stating in grid's options the name of the field to monitor, as in `rowStateField:isActive`, then add handler to the grid's instance to change css style of the row depending on the value.

```
self._dataGrid.addHandler('row_state_changed', function(s,a){
    if (!a.val){
        a.css = 'rowInactive';
    }
}, self.uniqueID);
```

DataPager:

The *Data Pager* is a separate control. Like all the controls defined in the framework, for their use in declarative style, it has the complementing element view *PagerElView*. In order to display it on the page, we need to add an additional markup, as in the example:

```
<div style="margin-top:40px;text-align:left; border:none;width:100%;height:15%">
  <!--the pager definition-->
  <div style="float:left;" data-bind="{this.dataSource,to=dbSet,source=VM.productVM}"
    data-view="name=pager,options={sliderSize:20,hideOnSinglePage=false}"></div>

  <!--a display of the count of the total and selected rows-->
  <div style="float:left; padding-left:10px;padding-top:10px;"><span>Total:</span>&nbsp;<span data-
bind="{this.value,to=totalCount,source=VM.productVM.dbSet}"></span>&nbsp;&nbsp;<span>Selected:</span>&nbsp;<span data-
bind="{this.value,to=selectedCount,source=VM.productVM}"></span></div>

  <!--a button to add new product-->
  <button style="float:right;" data-
bind="{this.command,to=addNewCommand,mode=OneWay,source=VM.productVM}">+ New Product</button>
</div>
```

DataEditDialog:

The DataEditDialog is a control used to display modal popup dialogs. This control is used by the DataGrid control to display edit dialogs. But in addition this control can be used independently from the DataGrid to display different templated information in applications.

The DataEditDialog uses a template to create the visual display. the Template can use data bindings in its markup to bind html controls to the datacontext's data.

```
var self = this, editorOptions = utils.extend(false, {
    dataContext: null, //could set default template's dataContext here
    dbContext: null, //could set dbContext here, if it is not a default app.dbContext
    templateID: 'someTemplateID',
    width: 500,
    height: 350,
    title: 'data edit dialog',
    submitOnOK: false,
    fn_OnClose: null,
    fn_OnOK: null,
    fn_OnCancel: null,
    fn_OnTemplateCreated: null,
    fn_OnShow: null
}, options);

var dialog = global.getType('DataEditDialog').create(editorOptions);
dialog.dataContext = dataContext; //can set datacontext for dialog's template before its showing
dialog.show();
```

The most interesting function in the options is `fn_OnTemplateCreated` which is used to get access to the template, when it is created. As an example, we can get HTML DOM elements inside the template and add event listeners for them to add some extra behavior.

```
//a template example
<div id="treeTemplate">
    <div data-name="tree" style="height:90%;"></div>
    <span style="position:absolute;left:15px;bottom:5px;font-weight:bold;font-size:10px;color:Blue;"
        data-bind="{this.text,to=selectedItem.fullPath,mode=OneWay}"></span>
</div>

//helper function to extract dom element from template by data-name attribute value
var fn_getTemplateElement = function(template, name){
    var t = template;
    var els = t.findElByDataName(name);
    if (els.length < 1)
        return null;
    return els[0];
};

//example of the function definition
fn_OnTemplateCreated: function(template){
    //executed in the context of the dialog
    var dialog = this, $ = global.$;
    var $tree = global.$(fn_getTemplateElement(template, 'tree'));
    self._folderBrowser = FolderBrowser.create({$tree:$tree, includeFiles:self.includeFiles});
    self._folderBrowser.addHandler('node_selected', function(s,a){
        self.selectedItem = a.item;
    },self.uniqueID)
}
```

`fn_OnClose` - function is usefull to do some processing after the dialog has been closed.

```
fn_OnClose: function(dialog){
    if (dialog.result == 'ok' && !!self._selectedItem){
```



```

        self._onSelected(self._selectedItem, self._selectedItem.fullPath);
    }
}

```

fn_OnOK - function is invoked when the user clicks the dialog's OK button. Depending on the result returning by this function the dialog can not be allowed to close. If this function returns [app.modules.datadialog.consts.DIALOG_ACTION.StayOpen](#), then the dialog is not closed after that (*you can see how it was done in ManyToMany demo's view mode*). This behavior can be usefull to execute some action on clicking the OK button, such as to submit results to the server but to wait for the success before closing the dialog.

submitOnOK - is an option, which is set to true, uses application's DbContext (*it is default*) or other DbContext if it was set in the options to submit the changes to the server, and to wait for the result. If the submit is not successful and the template uses dataForm element view in the markup, then the validations errors are displayed on the dataform automatically.

[an example of a custom dialog \(its view defined in the template\):](#)

add new customer address

Customer: **Abel R. Catherine**

Search existing Address:

AddressType	Address
Main Office	8713 Yosemite Ct.
Main Office	992 St Clair Ave East
Main Office	99, Rue Saint-pierre
Main Office	4400 March Road
Main Office	5 place Ville-Marie
Main Office	32605 West 252 Mile Road, Suite 250

Address	City	CountryRegion
9178 Jumping St.	Dallas	United States
575 Rue St Amable	Quebec	Canada
2521 McPherson Street	Markham	Canada
770 Notre Dame Quest Bureau 800	Montreal	Canada
2550 Middlefield Road	Scarborough	Canada
65 Camelin Street	Hull	Canada
253711 Mayfield Place, Unit 150	Richmond	Canada
5th Floor, 79 Place D'armes	Kingston	Canada
63 W Monroe	Chicago	United States
2500 North Stemmons Freeway	Dallas	United States
220 Mercy Drive	Garland	United States
7760 N. Pan Am Expwy	San Antonio	United States
44025 W. Empire	Denby	United States
23025 S.W. Military Rd.	San Antonio	United States
Lakeline Mall	Cedar Park	United States
Blue Ridge Mall	Kansas City	United States
First Colony Mall	Sugar Land	United States
Management Mall	San Antonio	United States
Ohms Road	Houston	United States
Factory Merchants	Branson	United States

1 2 >> Total: 94

DataForm:

The DataForm can be used in order to display a piece of HTML markup (*typically, inside the <div/> or <form/> tag*) that can have the dataContext attached to it.

The [DataForm](#) can contain the elements with [data-content](#) attribute which is used like in the DataGrid's column definitions. This attribute adds the support for automatic display of validation errors, and automatic switching between the edit and the display states.

P.S- The dataform (like the templates do) manages databindings by itself. The databindings inside daforms are created by the dataforms when the dataforms are created. This resembles creation of the databindings by templates. They also create databingings internally on templates creation and destroy them on the destruction. So the memory can be recycled. The dataform, like temlates can be set to disabled state to prevent all the data binding inside them - by setting its isDisabled property. When you set isDisabled property on the template's instance it sets all databindings and dataforms in it to disabled state. For example, the datadialog (it uses template to define its content) uses this feature to prevent data binding when it is hidden and enables data binding when it is shown.

The DataForm – is defined in the [dataform](#) module.

The DataForm instantiation is only possible in the code. In order to avoid using the code and get the benefits of the declarative style, the control has the specific element view - the [DataFormEView](#).

an example of the DataForm definition on the page:

```
<div style="width:100%" data-bind="{this.dataContext,mode=OneWay}" data-view="name=dataform">
  <table style="width:95%; border:none; table-layout:fixed; background-color:transparent;">
    <colgroup>
      <col style="width:225px;border:none;text-align:left;" />
      <col style="width:100%;border:none; text-align:left;" />
    </colgroup>
    <tbody>
      <tr><td>ID:</td><td><span data-content="fieldName:SalesOrderID"></span></td></tr>
      <tr><td>Order Date:</td><td><span data-content="fieldName:OrderDate"></span></td></tr>
      <tr><td>Due Date:</td><td><span data-content="fieldName:DueDate"></span></td></tr>
      <tr><td>Ship Date:</td><td><span data-content="fieldName:ShipDate"></span></td></tr>
      <tr><td>Order Status:</td><td><span data-
content="fieldName:Status,lookup:{dataSource=VM.customerVM.ordersVM.orderStatuses,valuePath=key,textPath=va
al},css:{editCss:'listbox-edit'}"></span></td></tr>
      <tr><td>Ship to Address:</td><td><span data-
content="fieldName:ShipToAddressID,lookup:{dataSource=VM.customerVM.custAdressView,valuePath=Address.Ad
dressID,textPath=Address.AddressLine1},css:{editCss:'listbox-edit'}"></span></td></tr>
      <tr><td>Bill to Address:</td><td><span data-
content="fieldName:BillToAddressID,lookup:{dataSource=VM.customerVM.custAdressView,valuePath=Address.Addr
essID,textPath=Address.AddressLine1},css:{editCss:'listbox-edit'}"></span></td></tr>
      <tr><td>Is Online Order:</td><td><span data-content="fieldName:OnlineOrderFlag"></span></td></tr>
      <tr><td>SalesOrder Number:</td><td><span data-
content="fieldName:SalesOrderNumber"></span></td></tr>
      <tr><td>Ship Method:</td><td><span data-content="fieldName:ShipMethod"></span></td></tr>
      <tr><td>Credit Card Approval Code:</td><td><span data-
content="fieldName=CreditCardApprovalCode"></span></td></tr>
      <tr><td>SubTotal:</td><td><span data-content="fieldName:SubTotal"></span></td></tr>
      <tr><td>TaxAmt:</td><td><span data-content="fieldName:TaxAmt"></span></td></tr>
      <tr><td>Freight:</td><td><span data-content="fieldName:Freight"></span></td></tr>
      <tr><td>TotalDue:</td><td><span data-content="fieldName:TotalDue"></span></td></tr>
      <tr><td>Comment:</td><td><span data-
content="fieldName:Comment,multyline:{rows:3,cols:20,wrap:hard}"></span></td></tr>
    </tbody>
  </table>
</div>
```

StackPanel:

The [StackPanel](#) - used to attach to a block tag (*typically, the <div/> tag*) the logic and markup for displaying a vertical or horizontal list of objects (*it can be a list of entities retrieved from data service*). The display of each object in the list is templated.

The StackPanel – is defined in the [stackpanel](#) module. In order to use the control declaratively it has the complementing element view [StackPanelElView](#). The control allows to use the keyboard keys (*left, right or up, down*) to move to the previous and next elements in the list.

an example of definitions of the StackPanels on the page:

```
<!--example of using stackpanel for vertical and horizontal list view-->
<div style="border: 1px solid gray;float:left;width:180px; min-height:50px; max-height:250px; overflow:auto;" data-
bind="{this.dataSource,to=historyList,source=VM.demoVM}" data-
view="name=stackpanel,options:{templateID:stackPanelItemTemplate,orientation:vertical}"></div>

<div style="border: 1px solid gray;float:left;min-height:50px; min-width:170px; max-width:650px; overflow:auto;
margin-left:15px;" data-bind="{this.dataSource,to=historyList,source=VM.demoVM}" data-
view="name=stackpanel,options:{templateID:stackPanelItemTemplate,orientation:horizontal}"></div>
```

an example of the display of the StackPanels on the page:



an example of the overall StackPanel's markup structure rendered on the page:

```
<div class="ria-stackpanel" data-
view="name=stackpanel,options:{templateID:stackPanelItemTemplate,orientation:horizontal}"
data-bind="{this.dataSource,to=historyList,source=VM.demoVM}" style="border: 1px solid gray;
float: left; min-height: 50px; min-width: 170px; max-width: 650px; overflow: auto;
margin-left: 15px;" data-elmkey0="s_10">
  <div class="stackpanel-item" style="display: inline-block;" data-key="clkey_0">
    <div class="stackPanelItem" style="width: 170px;">
      <fieldset>
        <legend><span data-elmkey0="s_14">radioValue2</span> </legend>Time:&nbsp;<span
          data-elmkey0="s_15">14:35:55</span>
      </fieldset>
    </div>
  </div>
  <div class="stackpanel-item" style="display: inline-block;" data-key="clkey_1">
    <!-- another template data here-->
  </div>
  <div class="stackpanel-item" style="display: inline-block;" data-key="clkey_2">
    <!-- another template data here-->
  </div>
  <div class="stackpanel-item current-item" style="display: inline-block;" data-key="clkey_3">
    <!-- another template data here-->
  </div>
</div>
```

ListBox:

The [ListBox](#) - is used to attach to a select tag the logic for binding it to the datasource.

It looks like ordinary combobox on the page, only the options in it are created and removed in response to the datasource data changes.

The ListBox – is defined in the [listbox](#) module. In order to use the control declaratively it has the complementing element view [SelectElView](#). This control is also used internally in the [LookupContent](#) used to display lookup fields.

Working with the data on the client side

3.1 Working with the simple collection's data

The framework contains the [Collection](#) type (*defined in the [collection](#) module*), which is the base abstract type for all other specialized collections. The [CollectionItem](#) is the base type for all the item types in these collections.

These base classes are defined in the *collection* module, which has also [ListItem](#), [List](#), and [Dictionary](#) types definitions.

These collections are used as data sources by the DataGrid , the StackPanel, the ListBox controls which are collection aware controls.

Each Collection's descendant instance has the [currentItem](#) property and the events which notify the controls about the changing current position, adding or removing of a collection item, and also about the start and the end of the item's editing.

All the types in the framework, including the [Collection](#) and the [CollectionItem](#) types are descendants of the [BaseObject](#) type, and they have all the properties, the methods and the events of that base type.

Collection's properties:

Property	Is readOnly	Description
permissions	Yes	Exposes permissions in a collection's descendant for updating, inserting and refreshing of the entities in the collection. Can be used to enable or disable some controls depending on the permissions. usage example: permissions.canEditRow, permissions.canAddRow, permissions.canDeleteRow, permissions.canRefreshRow
currentItem	No	Current item
count	Yes	Number of the items in the collection
totalCount	No	Total number of the items. Used for the paging support.
pageSize	No	Size of the data page. Used for the paging support.
pageIndex	No	Current page index. Zero based value.
items	Yes	Array of the collection items.
isPagingEnabled	Yes	If true then the collection descendant supports paging.
isEditing	Yes	True if a collection item is in editing state.
isHasErrors	Yes	True if the collection items have validation errors.
isLoading	No	True if the items are loading.
isUpdating	No	Set to true when the items are bulk updating. Used when it is needed to update large number of items without triggering begin_edit and end_edit events.
pageCount	Yes	The calculated number of the pages (<i>if the paging is supported</i>)
options	Yes	Exposes internal options object, which is created in the

	constructor.
--	--------------

Collection's methods:

Method	Description
getFieldInfo	Returns information about the field by a field name.
getFieldNames	Returns an array of field names.
cancelEdit	Cancels editing
endEdit	Ends editing if no validation errors, otherwise cancels editing.
getItemsWithErrors	Returns an array of the items which have validation errors.
addNew	Adds and returns new item. It is already in editing state.
getItemByPos	Returns item by the position (or null).
getItemByKey	Returns an item by the key (or null). The Key is the CollectionItem property _key which always has string type. When items are returned from the data service layer they have server side key (<i>string values of primary key concantenated by ;</i>). When the item is created on the client side before it is not submitted to the server, the _key property contains client generated value.
findByPK	Returns an item by primary key values (or null). Primary key values supplied as arguments ordered exactly as the fields in the primary key.
moveFirst	Moves current position to the first item in the collection. Returns boolean value indicating if the move was successful. Accepts an optional boolean parameter skipDeleted , which indicates if deleted (<i>and not submitted</i>) items should be taken into account or skipped.
movePrev	Moves current position to the previous item in the collection. Returns boolean value indicating if the move was successful. Accepts an optional boolean parameter skipDeleted , which indicates if deleted (<i>and not submitted</i>) items should be taken into account.
moveNext	Moves current position to the next item in the collection. Returns boolean value indicating if the move was successful. Accepts an optional boolean parameter skipDeleted , which indicates if deleted (<i>and not submitted</i>) items should be taken into account or skipped.
moveLast	Moves current position to the last item in the collection. Returns boolean value indicating if the move was successful. Accepts an optional boolean parameter skipDeleted , which indicates if deleted (<i>and not submitted</i>) items should be taken into account.
goTo	Moves current item position to the provided in the method argument. Returns boolean value indicating if the move was successful.
forEach	The same functionality as in the array forEach method.
removeItem	Immediately removes the item from the collection.
sortLocal	Sorts the collection items locally on the client. First parameter must be the array of field names, the second ' ASC ' or ' DESC '.
sortLocalByFunc	Sorts the collection items locally on the client. Expects sorting function for the items.
clear	Clears items from the collection.
waitForNotLoading	Utility method which allows to wait for callback function executed only when the collection is not loading the data.

Collection's events:

Event	Description
begin_edit	Occurs when an item started editing. It provides argument: { item:item }
end_edit	Occurs when an item ended editing. It provides argument: { item:item, isCanceled:isCanceled }
fill	Occurs when the items are started or ended filling the collection. when fill starting it provides argument: { isBegin:true, rowCount:objArray.length, time:new Date(), isPageChanged:false } when fill is ended it provides argument: { isBegin:false, rowCount:fetchdItems.length, time:new Date(), resetUI:!!clearAll, fetchdItems:fetchdItems, newItems:newItems, isPageChanged:false }
coll_changed	Occurs when the collection has been changed - items added, removed, item has changed _key property, or collection has been reset. argument for added items: { change_type:consts.COLL_CHANGE_TYPE.ADDED, items:[item], pos:[pos] } argument for removed items: { change_type:CH_T.REMOVE, items:[item], pos:[pos] } argument for reset: { change_type:consts.COLL_CHANGE_TYPE.RESET, items:[] } argument for _key changed: { change_type:COLL_CT.REMAP_KEY, items:[item], old_key:key, new_key:item._key }
item_deleting	Occurs before an item was deleted. argument { item:item, isCancel:false }
item_added	Occurs after a NEW item was added. argument { item:item, isAddNewHandled:false }
item_adding	Occurs before a NEW item was added. argument { item:item, isCancel:false }
validate	Occurs on an item's validation. argument { item:item, fieldName:null, errors:[] }
current_changing	Occurs when current item is changing. argument { newCurrent:newCurrent }
page_changing	Occurs when current page is changing. argument { page:this.pageIndex, isCancel:false }
errors_changed	Occurs when collection's item error status is changed. argument { item:item }
status_changed	Occurs when collection's item changeType (<i>deleted, updated, unchanged, added</i>) is changed. argument { item:item, oldChangeType:oldChangeType, key:item._key }
clearing	Occurs before the collection is cleared (<i>emptied</i>)
cleared	Occurs after the collection is cleared (<i>emptied</i>)
commit_changes	Occurs when changes on a collection's item are committed or rejected. argument { item:item, isBegin:isBegin, isRejected:isRejected, changeType:changeType }

CollectionItem's properties:

Property	Is readOnly	Description
_isNew	Yes	Returns true if the item is new (<i>in the Entity it is before the item is successfully submitted after its creation</i>). The item is set to the new state after it was created by the collection's addNew method.
_isDeleted	Yes	Returns true if the item in deleted state.

<code>_key</code>	False	Returns the key (<i>string value</i>) of the item in the collection.
<code>_collection</code>	Yes	Returns parent collection. (<i>DbSet, List, Dictionary</i>)
<code>_isUpdating</code>	Yes	Returns true if the collection in a bulk updating state.
<code>isEditing</code>	Yes	Returns true if the item in the editing state.

CollectionItem's methods:

Property	Description
<code>getFieldInfo</code>	Returns information about the field by its name in the form of the <code>fieldInfo</code> .
<code>getFieldNames</code>	Returns an array of field names.
<code>beginEdit</code>	Starts items's editing
<code>endEdit</code>	Ends (commits) items's editing
<code>cancelEdit</code>	Cancels current items's editing
<code>deleteItem</code>	Deletes the item
<code>addOnItemErrorsChanged</code>	Adds event handler for errors change event
<code>getFieldErrors</code>	Returns an array of errors for the field by field's name. If field's name is asterisk * , then it returns errors for the whole item validation.
<code>getAllErrors</code>	Returns all errors for the item validation.
<code>getErrorString</code>	Returns errors in the string form.
<code>_resetIsNew</code>	Used mainly internally, to reset <code>_isNew</code> property to false.
<code>getIsHasErrors</code>	Returns true if the item has any validation errors.

CollectionItem's events:

Event	Description
<code>errors_changed</code>	Occurs when item's errors collection is changed. Arguments { item:item }

FieldInfo object structure with default values:

```
var fieldInfo = {isPrimaryKey:0,
  isRowTimeStamp:false,
  dataType:consts.DATA_TYPE.NONE,
  isNullable:true,
  maxLength:-1,
  isReadOnly:false,
  isAutoGenerated:false,
  allowClientDefault:false,
  dateConversion:consts.DATE_CONVERSION.NONE,
  isClientOnly:true,
  isCalculated:false,
  isNeedOriginal:false,
  dependentOn:null,
  range:null,
  regex:null,
  isNavigation:false,
  fieldName:fieldName
};
```

The descendants of the collection type:

List – is simple collection of **ListItem** types.

Dictionary – is simple collection of **ListItem** types. It is has also **keyName** parameter in constructor arguments, so the items are indexed by the key and can be retrieved by their keys.

In order to fill the data we can use the collection's **fillItems** method, providing to it the items array in the first parameter, and the bool value in the second, to signal if the list must be cleared before the new items are added.

Example of the creation and filling the dictionary:

```
/* the first parameter is the name for list items's type (any unique name among the app types)

the second parameter is an array of property names for the ListItem
the third parameter is the name for the key property
*/
this._listTypes = global.getType('Dictionary').create('ListType', ['key', 'value', 'comment'], 'key');
//fill collection with the data
this._listTypes.fillItems([ { key: 'fullList', value: 'Full list', comment: 'Full list of items' },
{ key: 'uploadList', value: 'Uploaded list', comment: 'Item's list is uploaded' } ], false);
```

Another way to provide the property names to the List collection is by using an object with its own properties instead of the array. In this case property names are copied from the object.

```
this._listTypes = global.getType('Dictionary').create('ListType', {key:"", value:"", comment:""}, 'key');
```

3.2 Working with the DataService fed data

DbContext:

The **DbContext**'s instance is used to load the data from the data service.

One instance of the DbContext is created when the application is created (*if the default options is not overridden*).

The DbContext stores the data in DbSets (*the DbSet type is derived from the collection type*).

When the DbContext loads the data into the DbSets it first checks (*searching by the items' keys*) if the DbSet already contains the data item (*the Entity*). If the data item (*the Entity*) already contained in the DbSet then the data item is not replaced by the newly fetched one, but instead, it is the item's internal data that is refreshed.

DbContext's properties:

Property	Is readonly	Description
isBusy	Yes	Boolean property, which indicates that the DbContext doing some work (<i>typically asynchronous</i>).
isSubmitting	Yes	Boolean property, which indicates that the DbContext submit updates to the service end.
serverTimezone	Yes	Timezone of the server from which the application was loaded.
dbSets	Yes	A Hash map of the datasets. They are indexed by their names.
arrDbSets	Yes	An array of the datasets (<i>the same datasets which are stored in the hash map, only in the array form</i>).

serviceMethods	Yes	A Hash map (<i>indexed by names</i>) of the service methods (<i>client's proxys</i>), which can be executed from the client. Service method invocation is an asynchronous operation. It returns a promise which will be resolved with the data returned from the method, or rejected if the invocation failed.
hasChanges	Yes	Boolean property which indicates that the datacontext has some pending changes.
app	Yes	A reference to the application object.
service_url	Yes	Property for the string value of the data service url.
isInitialized	Yes	Indicates if the datacontext had been initialized with the metadata.

DbContext's methods:

Property	Description
getDbSet	Returns the DbSet by its name.
submitChanges	Submits the changes to the service. It is an asynchronous operation which returns a promise.
load	Loads the data from the service ends. It is an asynchronous operation which accepts a query object, and returns a promise. If the load is succesfull then the promise is resolved with the object in the format <code>{fetchedItems:array, newItems:array, isPageChanged:bool, outOfBandData: object }</code> .
acceptChanges	Sets all changes to the data as accepted. Typically it is automatically invoked when the datacontext successfully submits changes.
rejectChanges	All the changes that was not committed to the server are rejected, and the values are restored to the original ones.
waitForNotBusy	Used internally to wait for all async operations to complete, before invoking a callback method.
waitForNotSubmitting	Used internally to wait for a submit to complete, before invoking a callback method.
initialize	Accepts options for initializing the datacontext , such as the metadata and service url.
getAssociation	Returns a parent-child association object instance by its name.

DbContext's events:

Event	Description
submit_error	raised when submitting the changes is not sucessful. It allows to handle the submit error without rejecting the changes.
define_calc	raised to get calculated field definitions in response to the event. The event also invokes application's <code>define_calc</code> event. Typically it is the application's version of the event that is mainly used, but the DbContext's version of the event can be used when the DbContext's instance was created separately.

an example of the application creation which contains the default dbContext:

```
//create application instance
var app = RIAPP.Application.create({
  service_url: '@Url.RouteUrl("Default",new {controller="RIAppDemoService", action=""})',
```

```

metadata: @Html.Action("Metadata", "RIAppDemoService"),
createDbContext: true, //we want the application to create default DbContext's instance - it is default
moduleNames:['common','header','gridDemo'] //custom modules which will be initialized
});

```

When the default DbContext is created by the application, it is initialized with the metadata (*what datasets it contains and which fields those datasets have, fields types, fields constraints, which fields in the datasets are primary keys and which fields are read only*). This information is defined on the server side, and can not be altered later on the client. This information is used to check the data validity when the client submits the changes to the server.

The Other instances of the DbContext type can be also created separately in the javascript code.

an example of a separate dbContext creation:

```

var service_url = @Url.RouteUrl("Default",new {controller="FolderBrowserService", action=""});
var metadata= @Html.Action("Metadata", "FolderBrowserService");
var self = this;

var dbContext = global.getType('DbContext').create();
dbContext.addHandler('define_calc',function(s,args){
    if (args.dbSetName == 'FoldersDB' && args.fieldName == 'fullPath') {
        args.getFunc = function () {
            return self.getFullPath(this);
        };
    }
});

dbContext.initialize({
    serviceUrl: service_url,
    metadata: metadata
});

```

The DbContext's instance has references to DbSet instances, which in turn have Entities (*the DbSet is a collection type with Entity items*).

an example of loading the data using the DbContext:

```

var self = this, custAdress=self._custAdressDb, query = custAdress.createQuery('ReadAddressForCustomers');
//customerIDs for all loaded customers entities (for current page only, not which in cache if query.loadPageCount>1)
var custIDs = this._customerVM.dbSet.items.map(function(item){
    return item.CustomerID;
});

//send them to our service method which expects them (we defined our custom parameter, see the service method)
query.params = { custIDs: custIDs };
var promise = this.dbContext.load(query);
//then load related addresses
promise.done(function(){
    var addressIDs = custAdress.items.map(function(item){
        return item.AddressID;
    });

    //in our other viewmodel's method load new addresses and clear all previous addresses
    self._loadAddresses(addressIDs, true);
});

//the server side query method implementation
[Authorize(Roles =new string[] { "Users" })]
[Query]
public QueryResult<CustomerAddress> ReadAddressForCustomers(GetDataInfo getInfo, int[] custIDs)

```

```

{
    int? totalCount = null;
    var res = this.DB.CustomerAddresses.Where(ca => custIDs.Contains(ca.CustomerID));
    return new QueryResult<CustomerAddress>(res, totalCount);
}

```

another example of loading the data using the dbContext:

//a helper function to create text search criteria

```

var addTextQuery = function(query, fldName, val){
    var tmp;
    if (!!val){
        if (RIAPP.utils.startsWith(val,'%') && RIAPP.utils.endsWith(val,'%')){
            tmp = RIAPP.utils.trim(val,'% ');
            query.where(fldName, 'contains', [tmp])
        }
        else if (RIAPP.utils.startsWith(val,'%')){
            tmp = RIAPP.utils.trim(val,'% ');
            query.where(fldName, 'endswith', [tmp])
        }
        else if (RIAPP.utils.endsWith(val,'%')){
            tmp = RIAPP.utils.trim(val,'% ');
            query.where(fldName, 'startswith', [tmp])
        }
        else {
            tmp = RIAPP.utils.trim(val);
            query.where(fldName, '=', [tmp])
        }
    }
    return query;
};

```

//forming query and then loading data

```

function () {
    var query = this.dbSet.createQuery('ReadRegistr', self = this;
    query.pageSize = 100;
    query.loadPageCount = 250;
    query.isClearCacheOnEveryLoad = true;

    addTextQuery(query, 'SN_POL', this._filter.snpol);
    addTextQuery(query, 'FAM', this._filter.fam);
    addTextQuery(query, 'IM', this._filter.im);
    addTextQuery(query, 'OT', this._filter.ot);

    query.params = { d1: this._filter.period1, d2: this._filter.period2, cod: this._filter.cod };
    if (!!this._filter.mcod){
        query.where('MCOD', '=', [this._filter.mcod]);
    }

    if (!!this._filter.dr){
        query.where('DR', '=', [this._filter.dr]);
    }

    if (!utils._is.nt(this._filter.hasErr)){
        if (this._filter.hasErr)
            query.where('ERRS', '>', [0]);
        else
            query.where('ERRS', '=', [0]);
    }

    switch(this._sortID){
        case 'fio':
            query.orderBy('FAM',
'ASC').thenBy('IM','ASC').thenBy('OT','ASC').thenBy('DR','ASC').thenBy('MCOD','ASC').thenBy('INFOTYPE','ASC');
            break;
        case 'sum':
            query.orderBy('S_ALL', 'DESC').thenBy('FAM',
'ASC').thenBy('IM','ASC').thenBy('OT','ASC').thenBy('DR','ASC').thenBy('MCOD','ASC').thenBy('INFOTYPE','ASC');
            break;
    }
}

```

```

        case 'snpol':
            query.orderBy('SN_POL', 'ASC').thenBy('FAM',
'ASC').thenBy('IM','ASC').thenBy('OT','ASC').thenBy('DR','ASC').thenBy('MCOD','ASC').thenBy('INFOTYPE','ASC');
            break;
        case 'dr':
            query.orderBy('DR', 'ASC').thenBy('FAM',
'ASC').thenBy('IM','ASC').thenBy('OT','ASC').thenBy('DR','ASC').thenBy('MCOD','ASC').thenBy('INFOTYPE','ASC');
            break;
        default:
            throw new Error(String.format("Invalid sortID: {0}",this._sortID));
    }

    this.dbContext.load(query).always(function(){
        //after the query is executed, set the datagrid which displays the data to accept keyboard inputs
        global.currentSelectable= self._dataGrid;
    });
}

//the server side query method implementation which uses custom query building (using strings)
//it was done because Oracle has no support stored procedures which return select result like in MS SQL
[Authorize(Roles = new string[] { "Users" })]
[Query]
public QueryResult<VW_PERSON3> ReadRegistr(GetDataInfo getInfo, string d1, string d2, string cod)
{
    var isOk = Regex.IsMatch(d1, @"^d{6}$") && Regex.IsMatch(d2, @"^d{6}$");
    if (!isOk)
        throw new DomainServiceException(string.Format("invalid parameters format: {0}-{1}",d1,d2));

    int? totalCount = null;
    string filter = GetSqlFilter(d1, d2, cod, getInfo.filterInfo, getInfo.dbSetInfo);
    string paging = GetSqlPaging(getInfo);
    string sort = GetSqlSort(getInfo.sortInfo);
    if (String.IsNullOrEmpty(sort))
    {
        sort = "ORDER BY MCODE, FAM, IM, OT, DR, INFOTYPE";
    }

    string sql = @"WITH t AS (SELECT a.INFOMONTH, a.INFOTYPE, a.MCODE, a.REGNUM, SN_POL, SN_ERZ,
SMO, D_TYPE_R, Q, ANS_R, FAM,IM,OT,DR,W,UL,DOM,KV,KOR,STR, SUM(NVL(a.S_ALL,0)/100) AS S_ALL " +
"FROM OMSBILLS.PCOUNT a " +
    filter + @" GROUP BY a.INFOMONTH, a.INFOTYPE, a.MCODE, a.REGNUM, SN_POL, SN_ERZ, SMO, D_TYPE_R,
Q, ANS_R, FAM,IM,OT,DR,W,UL,DOM,KV,KOR,STR " +
    sort + ")", t2 AS (SELECT t.*, ROWNUM RN FROM t) SELECT INFOMONTH, INFOTYPE,
MCOD,REGNUM,SN_POL, SN_ERZ, SMO, D_TYPE_R, Q, ANS_R, FAM,IM,OT,DR,W,UL,DOM,KV,KOR,STR,
S_ALL, NULL as DS FROM t2 " + paging;

    //there or performance reasons we do total count only when fetching the first page
    if (getInfo.pageIndex == 0)
    {
        string cntsql = @"WITH t AS (SELECT 1 AS V FROM OMSBILLS.PCOUNT a " + filter + @" GROUP BY
a.INFOMONTH, a.INFOTYPE, a.MCODE, a.REGNUM, SN_POL, SN_ERZ, SMO, D_TYPE_R, Q, ANS_R,
FAM,IM,OT,DR,W,UL,DOM,KV,KOR,STR) SELECT COUNT(*) FROM t";
        var cntRes = this.DB.ExecuteStoreQuery<Decimal>(cntsql);
        totalCount = int.Parse(cntRes.FirstOrDefault().ToString());
    }

    var res = this.DB.ExecuteStoreQuery<VW_PERSON3>(sql).AsEnumerable();
    return new QueryResult<VW_PERSON3>(res, totalCount);
}

```

DataCache:

It is mostly internally used type, it is mainly not used by client user code directly, but it is important piece of the infrastructure.

If the query's `loadPageCount` property is set before the loading operation to the value of more than 1, then the loading operation returns several pages of the data (*loadPageCount value*).

Those extra pages of data are cached inside the query's instance with the help of **internally** used [DataCache](#) class. When the DbSet's pageIndex is changed and the DataCache has the cached page data then the data is taken from the local cache, and not from the server.

P.S - local data caching is very usefull for returning more data than can be displayed in the DataGrid at once (i say to display more than 200 rows in the grid is impractical, i usually use page size 50). When the query execution is slow (for complex queries), and when moving from one page to the other takes much time to return one page result from the server, it is better to preload more pages of the data to the client, in order to minify the time from going from one page to the other.

DataQuery:

The DbContext's [createQuery](#) method return an instance of the [DataQuery](#) type, which can be used to alter query options and supply additional query parameters.

DataQuery's properties:

Property	Is readonly	Description
loadPageCount		determines, how much pages to load when loading of the data is occurred. If the pageSize is 100 and loadPageCount is 25 then the loading will try to load 2500 records from the server (<i>if available</i>). For example, if we want to preload several pages from the server, so to cache them on the client, then we increase the loadPageCount property. When the user changes the DbSet's pageIndex by selecting a new page using the Pager control, the DbContext's load method checks if the needed data already cached on the client, and if it is, then the data is just retrieved from the local cache and not from the server. This behavior is very useful when, for example, the sql query which returns the data is very slow and the user must wait for considerable time when the retrieval of a page's data is finished. By way of preloading of several pages to the client, the user needs to wait for the data retrieval less frequently.
isClearCacheOnEveryLoad		determines if the cached data is cleared when the DbContext's load method is called explicitly. If we set it to true then the already cached data would be cleared, and replaced with the new data. If it is set to false, then the new data would be appended to the previous data.
isIncludeTotalCount		determines if the query will try to return the total count of the records. the Pager control

		uses the total count to calculate how many of pages of the data is available.
params		used to set the query parameters if the data service query method expects some explicit additional arguments.

an example of invoking a query method and supplying the parameters:

```
function () {
    //the query method can accept additional parameters that you can supply with query
    var query = this.dbSet.createQuery('ReadProduct');
    query.pageSize = 50;
    addTextQuery(query, 'ProductNumber', this._filter.prodNumber);
    addTextQuery(query, 'Name', this._filter.name);
    if (!utils._is.nt(this._filter.childCategoryId)){
        query.where('ProductCategoryId', '=', [this._filter.childCategoryId]);
    }
    if (!utils._is.nt(this._filter.modelID)){
        query.where('ProductModelID', '=', [this._filter.modelID]);
    }

    query.orderBy('Name', 'ASC').thenBy('SellStartDate', 'DESC');

    //supply parameters to ReadProduct service method
    //they are optional (you can not supply them at all and on the service method they will be null)
    query.params = { param1: [10, 11, 12, 13, 14], param2: 'Test' };

    this.dbContext.load(query).done(function(res){
        alert(res.outOfBandData.test);
    });
}

//the server side query method implementation, it expects two additional parameters - param1, param2
[Authorize()]
[Query]
public QueryResult<Product> ReadProduct(GetDataInfo getInfo, int[] param1, string param2)
{
    int? totalCount = null;
    var res = QueryHelper.PerformQuery(this.DB.Products, getInfo, ref totalCount).AsEnumerable();
    var queryResult = new QueryResult<Product>(res, totalCount);
    //example of returning out of band information and use it on the client (it can be something more useful)
    queryResult.extraInfo = new { test = "ReadProduct Extra Info: " + DateTime.Now.ToString("dd.MM.yyyy
HH:mm:ss") };
    return queryResult;
}
```

Besides loading of the DbSetes using query methods, we can explicitly invoke methods which can be defined on the data service (*which have `[invoke()]` attribute on them*). The access to these methods on the DbContext goes through DbContext's *serviceMethods* property.

an example of invoking the service method:

```
var self = this;

app.dbContext.serviceMethods.GetClassifiers({}).done(
    function (res) {
        //the result is set on our viewmodel's cls property
        //there arrays are converted to hash maps - for the use in calculated fields definitions
        self.cls = res;
    });
```


//our custom property definition on the viewmodel type where we convert arrays to hash maps

```
cls: {  
    set:function (v) {  
        var cls ={};  
        cls.uslDict = {};  
        cls.lpuDict = {};  
        cls.mkbDict = {};  
        cls.tipDict = {};  
        cls.otdDict = {};  
        cls.dtypeRDict = {};  
        cls.dtypeUDict = {};  
        cls.smoDict = {};  
        cls.ulDict = {};  
        cls.errDict = {};  
        v.usl.forEach(function(u){  
            cls.uslDict[u.k] = u;  
        });  
        v.lpu.forEach(function(u){  
            cls.lpuDict[u.k] = u;  
        });  
        v.mkb.forEach(function(u){  
            cls.mkbDict[u.k] = u;  
        });  
        v.tip.forEach(function(u){  
            cls.tipDict[u.k] = u;  
        });  
        v.otd.forEach(function(u){  
            cls.otdDict[u.k] = u;  
        });  
        v.dtypeR.forEach(function(u){  
            cls.dtypeRDict[u.k] = u;  
        });  
        v.dtypeU.forEach(function(u){  
            cls.dtypeUDict[u.k] = u;  
        });  
        v.smo.forEach(function(u){  
            cls.smoDict[u.k] = u;  
        });  
        v.ul.forEach(function(u){  
            cls.ulDict[u.k] = u;  
        });  
        v.err.forEach(function(u){  
            cls.errDict[u.k] = u;  
        });  
        this._cls=cls;  
    },  
    get:function () {  
        return this._cls  
    }  
}
```

//the server side method implementation

[Invoke()]

public Schet.DataService.Models.CIS GetClassifiers()

```
{  
    var lpu = this.DB.vw_LPUs.Select(a => new CLSItem { k = a.MCOD, v = a.FULLNAME }).ToList();  
    var mkb = this.DB.TMKB10s.Where(a => a.IsOSN == true).Select(a => new CLSItem { k = a.DS, v =  
a.NAME_DS }).ToList();  
    var usl = this.DB.vw_USLUGI2s.Select(a => new CLSItemInt { k = a.COD.Value, v = a.NAME }).ToList();  
    var dtypeR = this.DB.TOSOREEs.Select(a => new CLSItem { k = a.D_type.ToString(), v = a.Name }).ToList();  
    var dtypeU = this.DB.TOSOSCHes.Where(a => a.IsOsn == true).Select(a => new CLSItem { k =  
a.D_type.ToString(), v = a.N_type }).ToList();  
    var tip = this.DB.TipPrers.Select(a => new CLSItem { k = a.TIP, v = a.NAME_TIP }).ToList();  
    var otd = this.DB.TPROFOTs.Where(a => a.IsOSN == true).Select(a => new CLSItem { k = a.OTD, v =  
a.FULL_NAME }).ToList();  
    var smo = this.DB.vw_SMOs.Select(a => new CLSItem { k = a.QQ, v = a.NAME}).ToList();  
    var ul = this.DB.TSPR_ULs.Where(a=>a.PRIZNAK=='a').Select(a => new CLSItemInt { k = a.UL_ID, v =  
a.NAME }).ToList();  
}
```

```

        var err = this.DB.SOOKODs.Select(a => new CLSItem { k = a.ER_C, v = a.COMMENT }).ToList();

        Schet.DataService.Models.CIS res = new CIS { lpu = lpu, ul=ul, usl = usl, mkb = mkb, otd = otd, tip = tip, dtypeR =
dtypeR, dtypeU = dtypeU, smo = smo, err=err };

        return res;
    }

```

another example of invoking a service method with parameters:

```

dbContext.serviceMethods.TestInvoke({ param1: [10, 11, 12, 13, 14], param2: param.Name }).done(
function (res) {
    alert('TestInvoke result:' + res);
});

//the server side method implementation
[Invoke()]
public string TestInvoke(int[] param1, string param2)
{
    return string.Format("TestInvoke method invoked with param1: {0} param2: {1}", param1, param2);
}

```

DbSet:

the **DbSet** - is the items' collection. The items (*entities*) are typically loaded by executing a query through the DbContext's **load** method. But it can be loaded directly using DbSet's **fillItems** method as in:

```

//obtain data to load on the server side and imbed it into the HTML page
var json1 = @Html.Action("ProductModelData", "RIAppDemoService");
//load DbSet with obtained data when HTML is loaded from the web server
app.dbContext.dbSets.ProductModel.fillItems(json1);

//and this is the controller's action which is used to get the data in the above example
[ChildActionOnly]
public string ProductModelData()
{
    var info = this.GetDomainService().GetQueryData("ProductModel", "ReadProductModel");
    return info.ToJSON();
}

```

The above way of direct loading is usefull when we want that the data be already present in the DbSet when HTML page is loaded. It reduces the latency of invoking the separate queries (*separate ajax requests*) after the page has been loaded.

Besides the methods inherited from the collection type it has methods and properties specific to only the DbSet type:

DbSet specific methods:

Property	Description
fillItems	Loads the DbSet with locally stored query data. Typically used to fill some lookup DbSets with static data.
acceptChanges	Makes the pending changes as accepted.
rejectChanges	Rejects pending changes.
deleteOnSubmit	Accepts an item which should be deleted after submitting changes to the server, or by simply accepting changes locally.
createQuery	Creates DataQuery instance (<i>by query's name</i>). Later that instance can be used to add filter criteria, sort criteria, add query parameters.

clearCache	Explicitly clears local cache (<i>if it is used or it does nothing</i>)
------------	---------------------------------------------------------------------------

DbSet specific properties:

Property	is readonly	Description
dbContext	Yes	Returns parent DbContext instance
dbName	Yes	Returns the DbSet's name, as it was defined in the metadata.
entityType	Yes	Returns the type of the entities which this DbSet instance can contain.
isSubmitOnDelete	No	If it was set to true, then after any DbSet's item is submitted for delete, the changes is automatically submitted to the server.
query	Yes	Returns last query which is used to load the DbSet
hasChanges	Yes	Returns true if there is some pending changes.
cacheSize	Yes	Returns the count of records currently stored in the cache.

Entities:

The **Entity** - is a concrete implementation of the **CollectionItem** type.

Basically the Entity is a collection item which is specific for the DbSet type. When the DbContext is initialized with the metadata it creates instances of DbSets for all the DbSet's schemas which had been defined on the server (*in the metadata*) and was supplied to the DbContext's initialize method with the options.

Each DbSet instance on its creation creates a new type inherited from the Entity type.

The extended new type has all the properties inherited from the entity type and in addition it has all the field properties which had been provided by the schema defined in the metadata.

It can also has the navigation properties added by the association (*they are not defined in the DbSets schema, but only in the association's definition*).

an example of the DbSet's schema definition:

```
<data:DbSetInfo x:Key="SalesOrderDetail" dbName="SalesOrderDetail" insertDataMethod="Insert{0}"
updateDataMethod="Update{0}" deleteDataMethod="Delete{0}" enablePaging="False" EntityType="{x:Type
dal:SalesOrderDetail}">
  <data:DbSetInfo.fieldInfos>
    <data:FieldInfo fieldName="SalesOrderID" dataType="Integer" maxLength="4" isNullable="False"
isPrimaryKey="1" />
    <data:FieldInfo fieldName="SalesOrderDetailID" dataType="Integer" maxLength="4" isNullable="False"
isAutoGenerated="True" isReadOnly="True" isPrimaryKey="2" />
    <data:FieldInfo fieldName="OrderQty" dataType="Integer" maxLength="2" isNullable="False" />
    <data:FieldInfo fieldName="ProductID" dataType="Integer" maxLength="4" isNullable="False" />
    <data:FieldInfo fieldName="UnitPrice" dataType="Decimal" maxLength="8" isNullable="False" />
    <data:FieldInfo fieldName="UnitPriceDiscount" dataType="Decimal" maxLength="8" isNullable="False" />
    <data:FieldInfo fieldName="LineTotal" dataType="Decimal" maxLength="17" isNullable="False"
isReadOnly="True" />
    <data:FieldInfo fieldName="rowguid" dataType="Guid" maxLength="16" isNullable="False"
isAutoGenerated="True" isReadOnly="True" />
    <data:FieldInfo fieldName="ModifiedDate" dataType="DateTime" maxLength="8" isAutoGenerated="True"
isReadOnly="True" isNullable="False" />
    <data:FieldInfo fieldName="Product" dataType="None" isClientOnly="True" />
    <data:FieldInfo fieldName="ProductName" dataType="String" isCalculated="True" dependentOn="Product"
/>
  </data:DbSetInfo.fieldInfos>
</data:DbSetInfo>
```

Entity specific methods (besides inherited from CollectionItem):

Property	Description
deleteOnSubmit	Queues the item for deletion and the item status (changeType) is set to deleted.
deleteItem	Internally invokes deleteOnSubmit, so the methods are semantically equivalent.
getDbContext	Returns the parent DbContext.
refresh	Invokes the entity's data refresh. The method is asynchronous, it returns a promise.
acceptChanges	Accepts changes. Primarily used internally to accept changes after successful submit.
rejectChanges	Reject changes and restores items values to original.

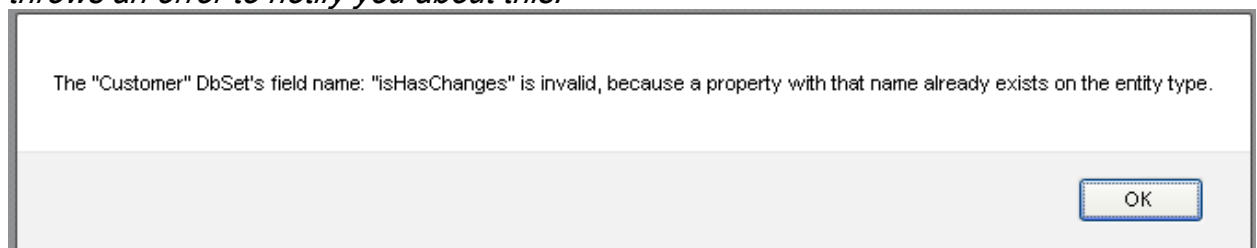
Entity specific properties (besides inherited from CollectionItem):

Property	is readonly	Description
_dbName	Yes	Returns the name of the parent DbSet.
_changeType	Yes	Returns the status (<i>change type</i>) of the entity. Where change type is defined by <code>consts.CHANGE_TYPE = { NONE:0, ADDED:1, UPDATED:2, DELETED:3 };</code>
_serverTimezone	Yes	Returns the time zone of the server.
_isRefreshing	Yes	Returns true if the entity (<i>the item</i>) is refreshing its values .
_isCached	Yes	Returns true if the item is in the DataCache. It is used mostly internally.
isHasChanges	Yes	Returns true if the items data is modified on the client.

When the DbContext creates a DbSet instance for the SalesOrderDetail, the schema info (*serialized to JSON*) is provided for the DbSet's constructor.

The DbSet's constructor creates specific entity type which has all the fields defined in the provided schema.

P.S. - The field names in the schema must not intersect (case sensitively) with the property names inherited from the Entity type. (for example: toString, isHasChanges, isEditing) Although it is unlikely in the real world, but the framework checks this and throws an error to notify you about this.



So we can directly access these fields values using those properties on an instance of the entity.

an example of direct access of the ProductName field:

```
app.dbContext.dbSets.SalesOrderDetail.items.forEach(function(item){
    console.writeln(item.ProductName);
});
```

When a new entity is added, it is in the editing state, so in the end of the entity modifications the `endEdit` method should be called to end the editing. If instead, the `cancelEdit` was called, the new entity would be removed from the collection.

Example of assigning field values to the new entity:

```
//create new entity
var item = dbSet.addNew();

//modify new entity
item.LineTotal = 200;
item.UnitPrice = 100;
item.ProductID = 1;

//commit the changes on the client
item.endEdit();

//commit the changes to the server
app.dbContext.submitChanges();
```

When we assign a new value to a property, the entity implicitly calls its `beginEdit` method (*if the entity is not in the editing state already*).

Every call to the `beginEdit` or `endEdit` method, if it completes successfully, will raise 'begin_edit' or 'end_edit' events. In order to prevent triggering those events, for example, when we want to mass update DbSet items or to update clientonly fields in a readonly DbSet, we can use the DbSet's `isUpdating` property.

an example of bulk updating DbSet's items:

```
//prevent implicit calls to beginEdit method
pDbSet.isUpdating = true; //mark the update started
try
{
    self._dbSet.items.forEach(function(item){
        var key = item._key, lidx = key.lastIndexOf(",");
        key = key.substring(key,lidx);
        var pitem = pDbSet.getItemByKey(key);
        if (!pitem.pcounts){
            pitem.pcounts=[]; //sets the entity field not causing triggering of the events
        }
        pitem.pcounts.push(item);
    });
}
finally
{
    pDbSet.isUpdating = false; //mark the update ended
}
```

Besides the normal fields, entities can have calculated (*readonly, calculated on the client side*) and client only (*editable, but for client side only use*) fields.

Calculated fields:

The calculated fields are just what they are - they calculate their values from other fields of the entity (*or can take data elsewhere*). There must not be circular references when a calculated field is defined, or it will result in an error. The calculated fields are read only. They can depend on other fields (*calculated or not*), and be automatically refreshed when those fields are changed.

A calculated field exists only on the client side, mainly for displaying values that is needed to be calculated.

The calculated fields are declared in the server side metadata in the DbSet's schema.

```
<data:FieldInfo fieldName="Name" dataType="String" isCalculated="True"
dependentOn="FirstName,MiddleName,LastName" />
```

The real calculations are defined on the client side in the application's (*or DbContext's*) event handler.

P.S.- The handler is invoked only one time - in the time of the DbContext's initialization with the metadata. So it must be attached before that.

```
//define calculated fields in the application's event handler
app.addHandler('define_calc', function (sender, data) {
    //utility function which used in calculations of Customer Full Name
    function toText(str){
        if (str === null)
            return "";
        else
            return str;
    }

    if (data.dbSetName == 'Customer' && data.fieldName == 'Name') {
        data.getFunc = function () {
            return toText(this.LastName) + ' ' + toText(this.MiddleName) + ' ' + toText(this.FirstName);
        };
    }

    if (data.dbSetName == 'SalesOrderDetail' && data.fieldName == 'ProductName') {
        data.getFunc = function () {
            if (!this.Product)
                return null;
            return this.Product.Name;
        };
    }
});
```

If we manually create an instance of the DbContext (*not the one which is created by the application*), then we can use its own 'define_calc' event to get the calculated fields definitions for this DbContext instance.

```
self._dbContext = global.getType('DbContext').create();
```

```
//define calculated fields in the DbContext's event handler - should be attached before DbContext's initialization
self._dbContext.addHandler('define_calc',function(s,args){
    if (args.dbSetName == 'FoldersDB' && args.fieldName == 'fullPath') {
        args.getFunc = function () {
            return self.getFullPath(this);
        };
    }
});

self._dbContext.initialize({
    serviceUrl:self.service_url,
    metadata:self.metadata
});
```

Client only fields:

The client only fields are just what they are - they are used only on the client, their values can be read and written. They are declared on the server in the DbSet's schema.

```
<data:FieldInfo fieldName="Address" dataType="None" isClientOnly="True" />  
<data:FieldInfo fieldName="Customer" dataType="None" isClientOnly="True" />
```

They can be used to store any values which are needed for the application on the client side, but their modifications (*updates*) are not propagated to the server side.

They can have a fixed type, like **number**, **bool**, **string**, **date** or can have **None** type which allows to store in them values of any type, like an entity or arrays of entities.

Navigation fields:

An entity can also have navigation properties, they are based on foreign key relations between entities. These foreign key relations in the framework are encapsulated in the association type (*which is discussed more deeply further in the doc*). The relations (*parent - child*) are defined in these associations in the metadata definition. There can be also many to many relations which are defined by two associations.

The association definition can state (*optionally*) the names of the navigations fields, and if they were stated, then the association definition extends the corresponding entities with navigational fields. The parent entity can get (*using its navigational field*) an array of child entities and the child entity can get its parent entity. These navigational fields can be used in databindings as well as ordinary entity's fields.

Also one more important use of the navigational fields is if you want to insert parent entity along with child entity in one transaction. Typically (*without using navigation fields*), you would insert parent entity, then submit changes to the server to obtain primary key for the entity (*they are commonly autogenerated on the server*), then assign this primary key to child entities foreign key fields and then submit them to the server. It takes to server transactions and complicates user's experience.

But, with the use of childToParent navigation field, you can assign parent entity directly to this navigation field. On submit, the data service fixes this relation automatically, and the submit is accomplished in one transaction.

an example of assigning parent entity to the navigation field:

```
var cust = this.currentCustomer;  
var ca = this.custAdressView.addNew(); //create brand new entity: CustomerAddress  
ca.CustomerID = cust.CustomerID;  
ca.AddressType = "Main Office"; //this is default, the user can edit it later  
ca.Address = address; //assign parent entity - it also can be new and has no server generated Primary key - the data  
service fixes it on submit  
ca.endEdit();  
//here we can submit changes to the server with dbContext's submitChanges method  
//or do more changes on the client, and submit them when we want in one transaction
```

The Entity and fields validations:

The entity validation process is done on the client and as an additional insurance on the server.

The client side validation is triggered when the new value is assigned to the field and when the entity editing ends (*when the endEdit method is invoked implicitly or explicitly*).

For the most cases of the validation it is usually enough only automatic validation based on the checks defined in the DbSet's schema. The DbSet's schema can include checks for nullability (isNullable), range, regex, maximum length (maxLength), field writability (isReadOnly) and type checking based on the field data type (string, number, bool, date).

When the automatic validation is not enough then we can resort to the custom validation. The types derived from the Collection (*List, Dictionary, DbSet*) have `validate` event, which is used for the custom client side validation.

When all the fields in the entity is validated (*typically when ending the entity's editing*), the event handler receives the arguments in this shape:

```
var args = {item:item, fieldName:null, errors:[]};
```

When one of the fields of the entity is validated (*typically when assigning new value to a field*), the event handler receives the arguments in this shape:

```
var args = {item:item, fieldName:fieldName, errors:[]};
```

The event handler can check custom validation conditions and then can add errors to the error property if it is not validated.

an example of the custom data validation on the client:

```
this._dbSet.addHandler('validate', function (sender, args) {
    if (!args.fieldName){ //full item validation
        if (!!args.item.SellEndDate){ //check it must be after Start Date
            if (args.item.SellEndDate < args.item.SellStartDate){
                args.errors.push('End Date must be after Start Date');
            }
        }
    }
    else //validation of a field value
    {
        if (args.fieldName == "Weight"){
            if (args.item[args.fieldName] > 20000){
                args.errors.push('Weight must be less than 20000');
            }
        }
    }
}, self.uniqueID);
```

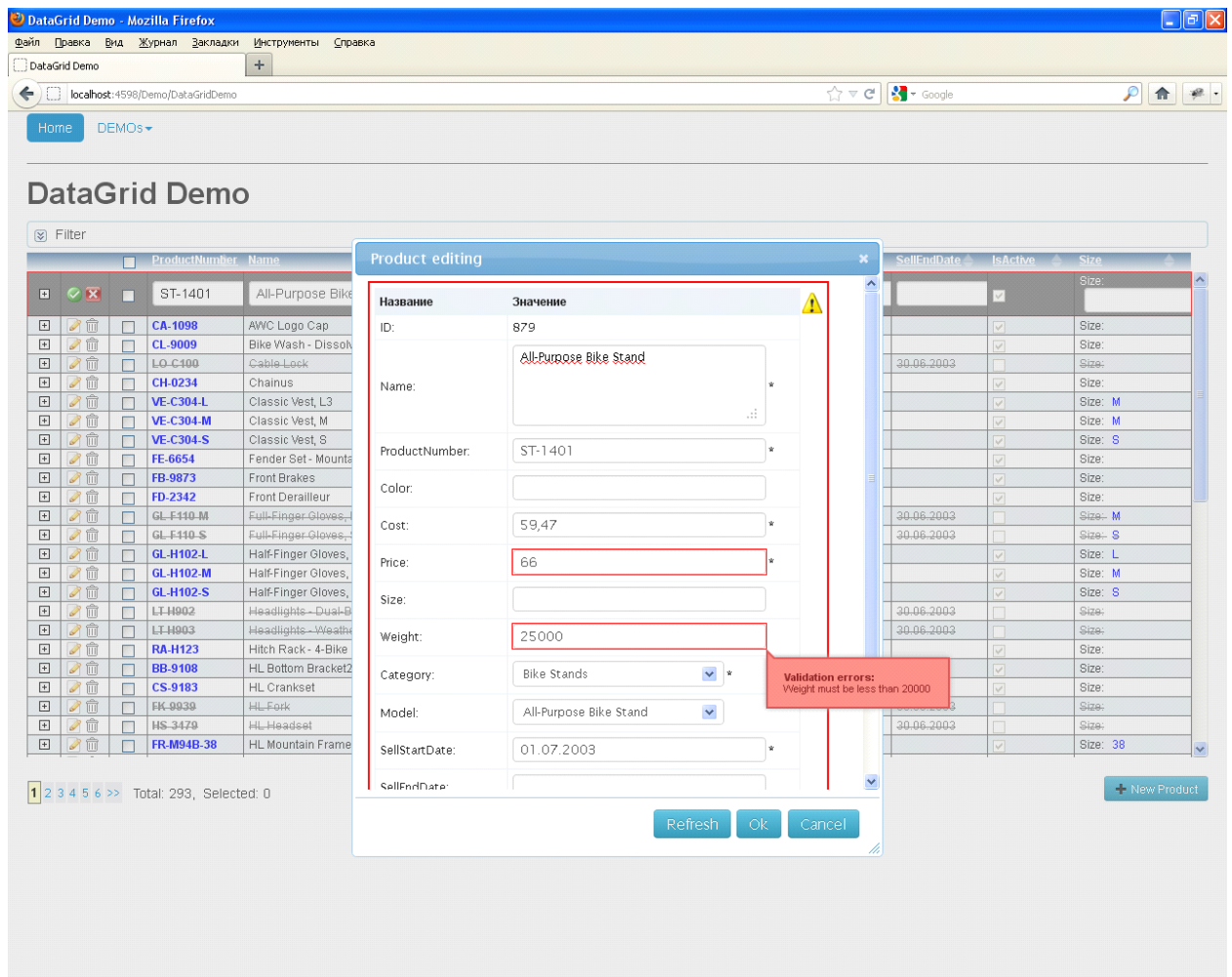
On an unsuccessful client side validation, the errors are added to the DbSet's internal collection of errors, and after that, an error of `ValidationError` type is raised.

The entity remains in the editing state and can not commit the edits when the errors exist.

In order to commit the edits, the correct values should be assigned which pass the validation, or otherwise the edits must be canceled with the `cancelEdit` method.

If the validation error occurred inside the DataBinding's code (*when fields was data bound for UI display*), then the ValidationError is caught inside the DataBinding and handled.

When a collection item (*typically entity*) has errors, then a UI control (*DataGrid, DataForm*) will display errors (*red borders and tooltips on mouse hovering*) next to unsuccessfully validated fields and the data form will have validation summary at the top right corner (*if you hover a mouse over it, the tooltip will be shown*).



The errors are cleared when the correct value is assigned to the field or the changes are canceled by using the `cancelEdit` method.

As it was noted above, the validation is done on the client and the server side, and for the automatic validation there is nothing else to do than to define checks in the DbSet schema.

But for a custom server side validation we should implement an entity validation method in the data service. This method is executed before committing updates to the database, and if the validation had been unsuccessful, the updates would not be committed to the database and the client side will be informed about the error.

an example of server side custom validation:

```
public IEnumerable<ValidationErrorInfo> ValidateProduct(Product product, string[] modifiedField)
{
    LinkedList<ValidationErrorInfo> errors = new LinkedList<ValidationErrorInfo>();
    if (Array.IndexOf(modifiedField, "Name") > -1 &&
        product.Name.StartsWith("Ugly", StringComparison.OrdinalIgnoreCase))
        errors.AddLast(new ValidationErrorInfo { fieldName = "Name", message = "Ugly name" });
    if (Array.IndexOf(modifiedField, "Weight") > -1 && product.Weight > 20000)
        errors.AddLast(new ValidationErrorInfo { fieldName = "Weight", message = "Weight must be less than 20000" });
    if (Array.IndexOf(modifiedField, "SellEndDate") > -1 && product.SellEndDate < product.SellStartDate)
        errors.AddLast(new ValidationErrorInfo { fieldName = "SellEndDate", message = "SellEndDate must be after SellStartDate" });
    if (Array.IndexOf(modifiedField, "SellStartDate") > -1 && product.SellStartDate > DateTime.Today)
        errors.AddLast(new ValidationErrorInfo { fieldName = "SellStartDate", message = "SellStartDate must be before DateTime.Today" });
}
```

```

        errors.AddLast(new ValidationErrorInfo { fieldName = "SellStartDate", message = "SellStartDate must be
prior today" });

        return errors;
    }

```

DataView:

The **DataView** – is a descendant of the collection type. It is used to wrap an existing collection (*typically a DbSet*), from which we want to expose only a partial set of the data (*the use of `fn_filter` property for locally filtering the data*). This partial set of the data can be further locally sorted (*the use of `fn_sort` property for locally sorting the data*).

The DataView is useful when working with the parent – child relationship.

For example, an Order entity can have many OrderItem child entities.

If we wanted to display on a page the Order entities using the DataGrid control, and next to that DataGrid we wanted to display another DataGrid which displays the Order Items child entities,

we could achieve this by loading from the data service the data for the Order entity currently selected in the DataGrid.

When moving to the next Order, we could clear the DbSet which stores the child entities and then reload from the data service another batch of child entities.

But this procedure of repeatedly clearing and loading child records from the service would result in too many server requests, and the delay in displaying child entities.

A better way to achieve the same result, would be to load all child entities for all the loaded parent entities. The OrderItem's DbSet could be wrapped with a DataView, and for the display of a subset of child entities we could just change the filter condition on the DataView. The OrderItems relative to the current Order would be filtered from the local data and the display's refresh would be much faster.

There are two ways of filtering the data in the **DataView**:

1) by providing a subset of the already prefiltered data through `fn_itemsProvider` which is a faster than to use only filtering.

```

//many to many filtering of Customer - CustomerAddress - Address relationship
//this view will expose Addresses related to the current customer in the viewModel
var addressesView = global.GetType("DataView").create(
    {
        dataSource: this._addressesDb,
        fn_sort: function(a,b){return a.AddressID - b.AddressID;},
        fn_filter: function(item){
            if (!self._currentCustomer)
                return false;
            return item.CustomerAddresses.some(function(ca){
                return self._currentCustomer === ca.Customer;
            });
        },
        fn_itemsProvider: function(ds){
            if (!self._currentCustomer)
                return [];
            //get an array of CustomerAddress entites from parentToChildren navigation property on the master entity
            var custAdrs = self._currentCustomer.CustomerAddresses;
            //then filter one more level down to get from CustomerAddress entity the array of Address entities
            return custAdrs.map(function(m){
                return m.Address;
            }).filter(function(m){
                return !!m;
            });
        }
    }
);

```

```
    }  
  });
```

2) only by filtering the data with `fn_filter`

```
var addressInfosView = global.getType('DataView').create(  
  {  
    dataSource: this._addressInfosDb,  
    fn_sort: function(a,b){return a.AddressID - b.AddressID;},  
    fn_filter: function(item){  
      return !item.CustomerAddresses.some(function(CustAdr){  
        return self._currentCustomer === CustAdr.Customer;  
      });  
    }  
  });
```

```
//enable paging in the view  
addressInfosView.isPagingEnabled = true;  
addressInfosView.pageSize = 50;
```

The only mandatory option's property is `dataSource` - which is a `DbSet` instance which is being wrapped up with the `DataView`. We can omit the sorting and filtering functions if we don't want to filter or sort the data.

At a time when we want the data in the `DataView` to be refreshed (*refiltered and resorted*) we can call `DataView`'s `refresh` method.

```
addressInfosView.refresh();
```

DataView's specific methods:

Property	Description
<code>refresh</code>	Refilters and resorts the data in the <code>DataView</code>
<code>clear</code>	Overriden collection's method. Clears only dataview's data, the real data which is in the wrapped <code>DbSet</code> is not touched.
<code>addNew</code>	Overriden collection's method. Adds a new entity (<i>it creates new entity</i>) to the underlying <code>DbSet</code> .
<code>appendItems</code>	Overriden collection's method. Adds an array existing in the <code>DbSet</code> entities to the view. The items skip filtering before adding to the view.

DataView's specific properties:

Property	is readonly	Description
<code>fn_filter</code>	No	the filter function
<code>fn_sort</code>	No	the sort function
<code>fn_itemsProvider</code>	No	the function to provide already prefiltered items
<code>isReadOnly</code>	Yes	Returns wrapped <code>DbSet</code> 's <code>isReadOnly</code> property value.
<code>isPagingEnabled</code>	No	Overriden property. If it was set to true then the view split its data in pages.
<code>canEditRow</code>	Yes	Returns wrapped <code>DbSet</code> 's <code>canEditRow</code> property value.
<code>canAddRow</code>	Yes	Returns wrapped <code>DbSet</code> 's <code>canAddRow</code> property value.
<code>canDeleteRow</code>	Yes	Returns wrapped <code>DbSet</code> 's <code>canDeleteRow</code> property value.
<code>isHasErrors</code>	Yes	Returns wrapped <code>DbSet</code> 's <code>isHasErrors</code> property value.

the Association and the ChildDataView types:

Although the DataViews by themselves can be used for parent - child relationship management, it requires a lot of code. To simplify the obtaining of children of the master entity, or vice versa, master of a child (*in other words navigation properties*), it is better to use an association instance to manage this task. The association stores and updates the map of the parent-child relationship based on foreign keys. The Association definition is done in the metadata on the server side.

```
<data:Metadata x:Key="FolderBrowser">
  <data:Metadata.DbSets>
    <data:DbSetInfo dbSetName="FoldersDB" enablePaging="False" EntityType="{x:Type
models:FolderModel}" deleteDataMethod="Delete{0}">
      <data:DbSetInfo.fieldInfos>
        <data:FieldInfo fieldName="Key" dataType="String" maxLength="255" isNullable="False"
isAutoGenerated="True" isReadOnly="True" isPrimaryKey="1" />
        <data:FieldInfo fieldName="ParentKey" dataType="String" maxLength="255" isNullable="True"
isReadOnly="True" />
        <data:FieldInfo fieldName="Name" dataType="String" maxLength="255" isNullable="False"
isReadOnly="True" />
        <data:FieldInfo fieldName="Level" dataType="Integer" isNullable="False" isReadOnly="True" />
        <data:FieldInfo fieldName="HasSubDirs" dataType="Bool" isNullable="False" isReadOnly="True" />
        <data:FieldInfo fieldName="IsFolder" dataType="Bool" isNullable="False" isReadOnly="True" />
        <data:FieldInfo fieldName="fullPath" dataType="String" isCalculated="True" />
      </data:DbSetInfo.fieldInfos>
    </data:DbSetInfo>
  </data:Metadata.DbSets>
  <!--the association definition -->
  <data:Metadata.Associations>
    <data:Association name="FolderToParent" parentDbSetName="FoldersDB" childDbSetName="FoldersDB"
childToParentName="Parent" parentToChildrenName="Children" onDeleteAction="Cascade" >
      <data:Association.fieldRels>
        <data:FieldRel parentField="Key" childField="ParentKey"></data:FieldRel>
      </data:Association.fieldRels>
    </data:Association>
  </data:Metadata.Associations>
</data:Metadata>
```

When we define an association in the metadata, we describe which key in child entity relates to the key in parent entity. We also give the name for navigation properties (*parentToChildrenName and childToParentName*). These properties are automatically added to the entities in parent and child DbSets.

After creating and initializing DbContext with the metadata, we can obtain an association instance by using DbContext's getAssociation method.

```
var assoc = self.dbContext.getAssociation(' FolderToParent ');
```

Association's methods:

Property	Description
getChildItems	Accepts an entity and returns an array of child entities
getParentItem	Accepts an entity and returns master entity

Association's properties:

Property	is readonly	Description
app	Yes	Returns current application instance.

name	Yes	Returns the name of the association as it was defined in the metadata.
parentToChildrenName	Yes	Returns the name of the navigation property on the entity which is used to get an array of child entities.
childToParentName	Yes	Returns the name of the navigation property on the entity which is used to get master entity.
parentDS	Yes	Returns parent DbSet in parent-child relationship.
childDS	Yes	Returns child DbSet in parent-child relationship.
onDeleteAction	Yes	Returns enum value of what is the action to take when a parent entity is deleted, as it was defined in the metadata.

In order to further simplify the creation of the DataView which exposes only child entities for some parent entity, there exists the [ChildDataView](#) which is the descendant of the DataView type. This extended DataView type has [parentItem](#) property which is assignable, and is used to refresh the view so it can expose only child entities for the current parent entity.

```
var custAssoc = app.dbContext.getAssociation('CustAddrToCustomer');
//the view to filter CustomerAddresses related to the current customer only
var custAddressView = global.getType('ChildDataView').create(
{
    association:custAssoc,
    fn_sort: function(a,b){return a.AddressID - b.AddressID;}
});

custAddressView.parentItem = someParent;
```

One more benefit of the ChildView, it is automatic reflection of the changes in parent-child relationship: if items are added or deleted from the relationship, then the view is updated automatically.

Working with the data on the server side

4.1 Data service

The data service application is implemented in C# language and requires Microsoft Net Framework 4 be installed on the server side computer.

The data service implements a public interface which can be integrated in a web service framework, such as ASP.net.

```
public interface IDomainService: IDisposable
{
    MetadataInfo ServiceGetMetadata();
    GetDataResult ServiceGetData(GetDataInfo getInfo);
    ChangeSet ServiceApplyChangeSet(ChangeSet changeSet);
    RefreshRowInfo ServiceRefreshRow(RefreshRowInfo getInfo);
    InvokeResult ServiceInvokeMethod(InvokeInfo parameters);
}
```

The interface contains methods which are invoked from the JRIApp client (*using DbContext type*).

The data service is typically hosted in ASP.NET MVC framework due to the web framework's very convenient design for the invocation of server side methods through Ajax calls.

RIAPP.DataService.Mvc assembly contains a descendant of System.Web.Mvc.Controller class: [DataServiceController](#), which encapsulates Data service methods invocations in a mvc controller.

The only method which needs to be overridden and implemented in the descendant of the class which can be used in applications is [CreateDomainService](#) method.

The base DataService ([BaseDomainService](#) class) is implemented in RIAPP.DataService assembly. It is an abstract class, it has two abstract methods which are needed to be implemented in a descendant are [GetMetadata](#) and [ExecuteChangeSet](#) methods.

For example, in EFDomainService class (*which is designed to work with the Microsoft's Entity Framework*) the [ExecuteChangeSet](#) method saves updates in the System.Data.Objects.[ObjectContext](#) inside the transaction's scope.

```
protected override void ExecuteChangeSet()
{
    using (TransactionScope transScope = new TransactionScope(TransactionScopeOption.RequiresNew,
        new TransactionOptions { IsolationLevel = IsolationLevel.ReadCommitted,
            Timeout = TimeSpan.FromMinutes(1.0) }))
    {
        this.DB.SaveChanges();

        transScope.Complete();
    }
}
```

The [GetMetadata](#) method is typically implemented to return the metadata which is stored and edited in a WPF control as a user control resource. This kind of storing and editing of the metadata is very convenient due to its usability (*declarative style, nodes can be collapsed, intellisense*).

```
<UserControl x:Class="RIAppDemo.BLL.DataServices.RIAppDemoMetadata"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    mc:Ignorable="d"
    xmlns:data="clr-namespace:RIAPP.DataService;assembly=RIAPP.DataService"
    xmlns:dal="clr-namespace:RIAppDemo.DAL;assembly=RIAppDemo.DAL"
    xmlns:models="clr-namespace:RIAppDemo.BLL.Models"
    d:DesignHeight="30" d:DesignWidth="30">
    <UserControl.Resources>
        <data:Metadata x:Key="FolderBrowser">
            <data:Metadata.DbSets>
                <data:DbSetInfo dbSetName="FoldersDB" enablePaging="False" EntityType="{x:Type
models:FolderModel}" deleteDataMethod="Delete{0}">
                    <data:DbSetInfo.fieldInfos>
                        <data:FieldInfo fieldName="Key" dataType="String" maxLength="255" nullable="False"
isAutoGenerated="True" readOnly="True" primaryKey="1" />
                        <data:FieldInfo fieldName="ParentKey" dataType="String" maxLength="255" nullable="True"
isReadOnly="True" />
                        <data:FieldInfo fieldName="Name" dataType="String" maxLength="255" nullable="False"
isReadOnly="True" />
                        <data:FieldInfo fieldName="Level" dataType="Integer" nullable="False" readOnly="True" />
                        <data:FieldInfo fieldName="HasSubDirs" dataType="Bool" nullable="False" readOnly="True" />
                    </data:DbSetInfo.fieldInfos>
                </data:DbSetInfo>
            </data:Metadata.DbSets>
        </data:Metadata>
    </UserControl.Resources>
</UserControl>
```



```

        <data:FieldInfo fieldName="IsFolder" dataType="Bool" isNullable="False" isReadOnly="True" />
        <data:FieldInfo fieldName="fullPath" dataType="String" isCalculated="True" />
    </data:DbSetInfo.fieldsInfos>
</data:DbSetInfo>
</data:Metadata.DbSets>
<data:Metadata.Associations>
    <data:Association name="FolderToParent" parentDbSetName="FoldersDB" childDbSetName="FoldersDB"
childToParentName="Parent" parentToChildrenName="Children" onDeleteAction="Cascade" >
        <data:Association.fieldRels>
            <data:FieldRel parentField="Key" childField="ParentKey"></data:FieldRel>
        </data:Association.fieldRels>
    </data:Association>
</data:Metadata.Associations>
</data:Metadata>
</UserControl.Resources>
</UserControl>

```

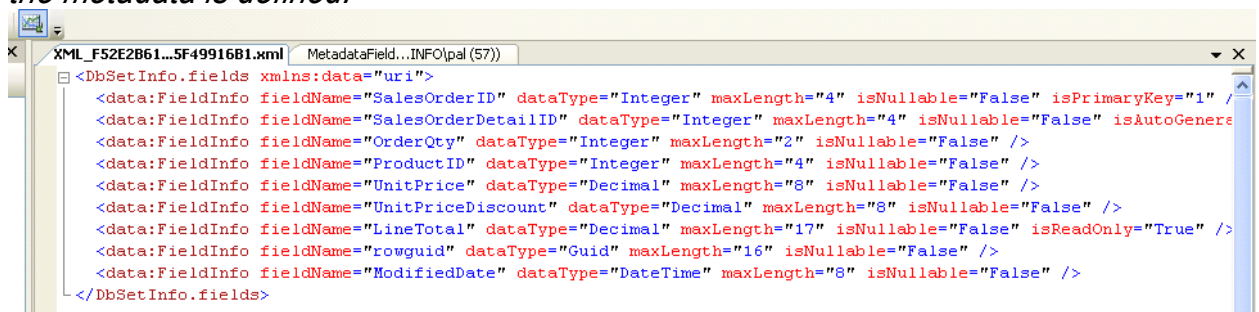
The above metadata is returned as a Metadata class instance by creating an instance of the WPF control, and getting the needed resource from it by its key.

```

protected override Metadata GetMetadata()
{
    return (Metadata)(new RIAppDemoMetadata()).Resources["FolderBrowser"];
}

```

P.S. - *It is not necessary to define the metadata manually. It can be automated, because the DbSet definitions are stored in XAML (xml) and you can make XML generation by looking for entities in the assemblies and reflecting on their attributes to create XML which can be pasted in the metadata. For example, I created a sql script (MetadataFields.sql in the Demo's App_Data folder) which helps me to create fields definitions from tables in MS SQL server. Then I paste and tweak some attributes and the metadata is defined.*



The DataService class typically implements query methods, which are distinguished from the other methods by using **Query** attribute on them.

```

[Query]
public QueryResult<LookUpProduct> ReadProductLookUp(GetDataInfo getInfo)
{
    int? totalCount = null;
    var res = QueryHelper.PerformQuery(this.DB.Products, getInfo, ref totalCount).Select(p => new
    LookUpProduct { ProductID = p.ProductID, Name = p.Name }).AsEnumerable();
    var queryResult = new QueryResult<LookUpProduct>(res, totalCount);
    return queryResult;
}

```

The query methods also return **QueryResult** instance. In order to simplify the querying, there is a **QueryHelper** class, which can be used to perform queries in more generic way. The helper understands how to treat **GetDataInfo** parameter to extract and use filters, sorting and paging info generated on the client side (*you can look inside it to see how it is done*).

For a one DbSet there can be several differently named query methods. The client can decide which one to use by their names.

an example of two query methods implementation for Product DbSet:

```
[Query]
public QueryResult<Product> ReadProduct(GetDataInfo getInfo, int[] param1, string param2)
{
    int? totalCount = null;
    var res = QueryHelper.PerformQuery(this.DB.Products, getInfo, ref totalCount).AsEnumerable();
    var queryResult = new QueryResult<Product>(res, totalCount);
    //example of returning out of band information and use it on the client (it can be more useful than it)
    queryResult.extraInfo = new { test = "ReadProduct Extra Info: " + DateTime.Now.ToString("dd.MM.yyyy
HH:mm:ss") };
    return queryResult;
}

[Query]
public QueryResult<Product> ReadProductByIds(GetDataInfo getInfo, int[] productIDs)
{
    int? totalCount = null;
    var res = this.DB.Products.Where(ca => productIDs.Contains(ca.ProductID));
    return new QueryResult<Product>(res, totalCount);
}
```

Query methods can accept additional parameters like in the above ReadProductByIds method. Clients can supply them for custom filtering or for executing a stored procedure that requires some parameters.

Query methods (*like the above ReadProduct method*) can also return an out of band info (*which is automatically serialized into json along with the query result*). The out of band info can include any information which can be used on the client for testing and application development purposes.

If you want to load a large number of rows at once (*for example, in the client rows caching scenario when query's loadPageCount > 0 and the paging is enabled*) then you can set the maximum rows which can be fetched in one batch by the query. Just set on the DbSet the FetchSize attribute to the number of rows. If the query returns more rows than FetchSize then the rows will be returned in batches (*batch size = FetchSize*) transparently to the client.

P.S- *in the real world applications it is advisable to set FetchSize to 2000-2500. If you will try to load too many rows at once without setting FetchSize then the json payload can be too big for the browser to process and it can throw an error.*

an example of setting the fetchsize for product entities:

```
<data:DbSetInfo dbSetName="Product" isTrackChanges="True"
validateDataMethod="Validate{0}" refreshDataMethod="Refresh{0}"
insertDataMethod="Insert{0}" updateDataMethod="Update{0}" deleteDataMethod="Delete{0}"
enablePaging="True" pageSize="100" FetchSize="2000" EntityType="{x:Type dal:Product}">
```

Aside from query methods, there can be a set of CRUD methods (*they are optional*) to perform inserts, deletes and updates on the entities. Their names are defined in the DbSet's metadata as insertDataMethod, updateDataMethod, deleteDataMethod.

```

<data:DbSetInfo dbSetName="CustomerAddress" insertDataMethod="Insert{0}" updateDataMethod="Update{0}"
deleteDataMethod="Delete{0}" enablePaging="False" EntityType="{x:Type dal:CustomerAddress}">
  <data:DbSetInfo.fieldInfos>
    <data:FieldInfo fieldName="CustomerID" dataType="Integer" maxLength="4" isNullable="False"
isReadOnly="False" isNeedOriginal="True" isPrimaryKey="1" />
    <data:FieldInfo fieldName="AddressID" dataType="Integer" maxLength="4" isNullable="False"
isReadOnly="False" isNeedOriginal="True" isPrimaryKey="2" />
    <data:FieldInfo fieldName="AddressType" dataType="String" maxLength="50" isNullable="False"
isReadOnly="False" isNeedOriginal="False" />
    <data:FieldInfo fieldName="rowguid" dataType="Guid" maxLength="16" isNullable="False"
isReadOnly="True" isRowTimeStamp="True" isNeedOriginal="True" />
    <data:FieldInfo fieldName="ModifiedDate" dataType="DateTime" maxLength="8" isNullable="False"
isReadOnly="True" isNeedOriginal="False" />
  </data:DbSetInfo.fieldInfos>
</data:DbSetInfo>

```

They are typically have templated names, such as Insert{0}. The real name is generated by replacing {0} with dbSetName. In this case the real names will be: **InsertCustomerAddress**, **UpdateCustomerAddress** and **DeleteCustomerAddress**. However their names can be typed without {0}, and then it will be used as is.

```

[Authorize(Roles = new string[] { ADMIN_ROLE })]
public void InsertCustomerAddress(CustomerAddress customerAddress)
{
    customerAddress.ModifiedDate = DateTime.Now;
    customerAddress.rowguid = Guid.NewGuid();
    this.DB.CustomerAddresses.InsertOnSubmit(customerAddress);
}

[Authorize(Roles = new string[] { ADMIN_ROLE })]
public void UpdateCustomerAddress(CustomerAddress customerAddress)
{
    CustomerAddress orig = this.GetOriginal<CustomerAddress>();
    this.DB.CustomerAddresses.Attach(customerAddress, orig);
}

[Authorize(Roles = new string[] { ADMIN_ROLE })]
public void DeleteCustomerAddress(CustomerAddress customerAddress)
{
    this.DB.CustomerAddresses.Attach(customerAddress);
    this.DB.CustomerAddresses.DeleteOnSubmit(customerAddress);
}

```

Aside from query and CRUD methods there are 3 more special types of the methods which can be used in the data service: Entity refresh methods, Custom validation methods and the service methods (*which can be invoked from clients directly by their name*).

Entity refresh methods

is used to refresh an entity's data from the service. The refresh also can be made by using a query method, but the refresh methods are more convenient to use from the client (*just use entity's **refresh** method*), and they are more performant. The refresh method name definition is done in the metadata as **refreshDataMethod**.

```

<data:DbSetInfo dbSetName="Product" isTrackChanges="True" validateDataMethod="Validate{0}"
refreshDataMethod="Refresh{0}" insertDataMethod="Insert{0}" updateDataMethod="Update{0}"
deleteDataMethod="Delete{0}" enablePaging="True" pageSize="25" FetchSize="200" EntityType="{x:Type
dal:Product}">
  <data:DbSetInfo.fieldInfos>
    //unrelevant markup ...
  </data:DbSetInfo.fieldInfos>
</data:DbSetInfo>

```

an example of the refresh method implementation:

```
public Product RefreshProduct(RefreshRowInfo refreshInfo)
{
    return QueryHelper.GetRefreshedEntityDataHelper<Product>(this.DB.Products, refreshInfo);
}
```

Custom validation methods

is used to validate an entity with custom checks. The validation method name definition is done in the metadata as `validateDataMethod`. The validation method has two parameters: the first is the entity, and the second is an array of field names modified.

an example of a server side validation method implementation:

```
public IEnumerable<ValidationErrorInfo> ValidateProduct(Product product, string[] modifiedField)
{
    LinkedList<ValidationErrorInfo> errors = new LinkedList<ValidationErrorInfo>();
    if (Array.IndexOf(modifiedField, "Name") > -1 &&
        product.Name.StartsWith("Ugly", StringComparison.OrdinalIgnoreCase))
        errors.AddLast(new ValidationErrorInfo { fieldName = "Name", message = "Ugly name" });
    if (Array.IndexOf(modifiedField, "Weight") > -1 && product.Weight > 20000)
        errors.AddLast(new ValidationErrorInfo { fieldName = "Weight", message = "Weight must be less than 20000" });
    if (Array.IndexOf(modifiedField, "SellEndDate") > -1 && product.SellEndDate < product.SellStartDate)
        errors.AddLast(new ValidationErrorInfo { fieldName = "SellEndDate", message = "SellEndDate must be after SellStartDate" });
    if (Array.IndexOf(modifiedField, "SellStartDate") > -1 && product.SellStartDate > DateTime.Today)
        errors.AddLast(new ValidationErrorInfo { fieldName = "SellStartDate", message = "SellStartDate must be prior today" });

    return errors;
}
```

Service methods

is used to be invoked from the client directly by their names, to make some sort of processing on the server and optionally to return a result in the form of a primitive type or a complex object. These methods are distinguished from the others by the `Invoke` attribute. They also can have `Authorize` attributes.

```
[Invoke()]
public string TestInvoke(int[] param1, string param2)
{
    return string.Format("TestInvoke method invoked with param1: {0} param2: {1}", param1, param2);
}
```

Data Service Metadata

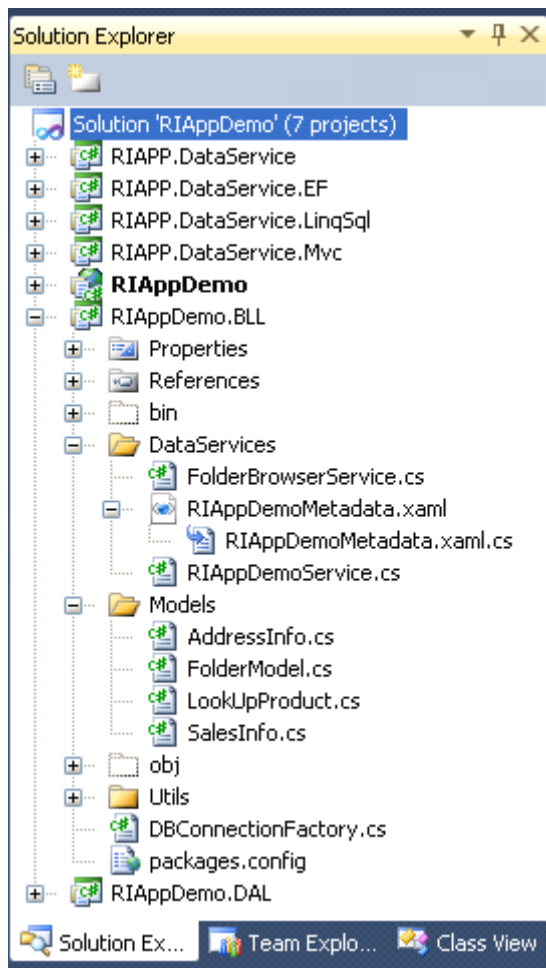
as it was said above it is typically created in the form WPF user control resources using a XAML markup. For the metadata and data services is better to use separate assembly (*class library*) in the solution.

In the demo solution they were put in *RIAppDemo.BLL* class library project.

RIAppDemoMetadata.xaml is the WPF user control that contains metadata.

RIAppDemoService.cs - file contains DataService for demo project.

FolderBrowserService.cs - file contains DataService for file and folder browser implementation demo.



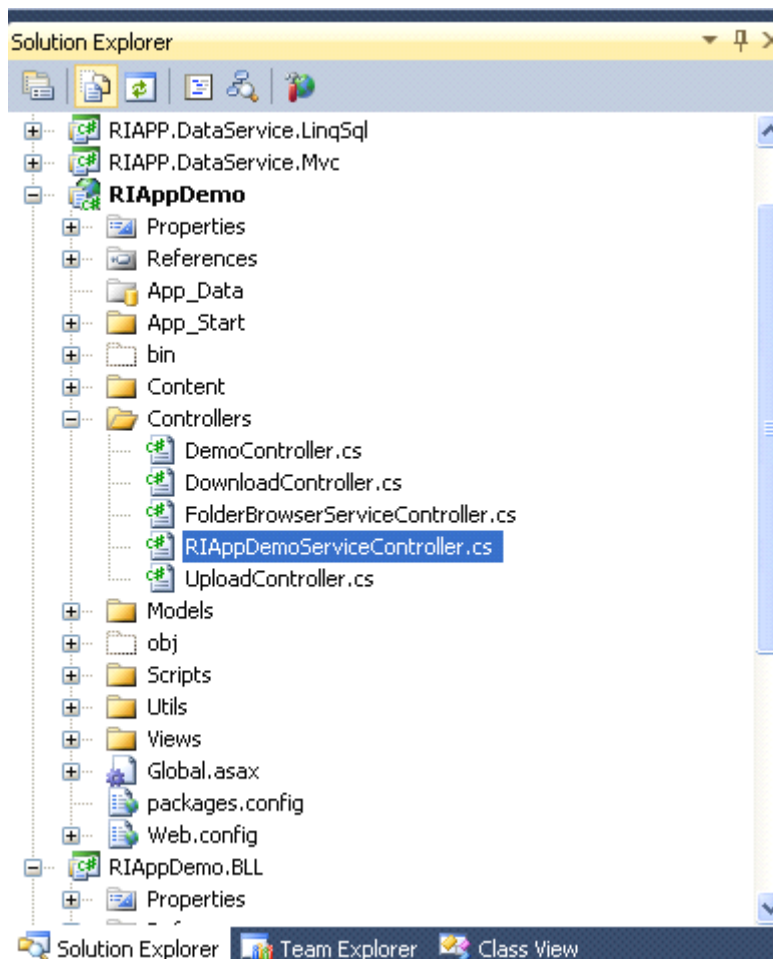
The ASP.Net MVC project *RIAppDemo* references that *RIAppDemo.BLL* library. But inside the ASP.NET MVC project the DataServices are exposed through MVC controllers. First web request from the client hits the controller and then the controller relays the request to the encapsulated DataService.

an example of the MVC controller implementation (from Demo project):

```
[SessionState(SessionStateBehavior.Disabled)]
public class RIAppDemoServiceController : DataServiceController<RIAppDemoService>
{
    protected override IDomainService CreateDomainService()
    {
        IDomainService svc = new RIAppDemoService(this.User);
        return svc;
    }

    [ChildActionOnly]
    public string ProductModelData()
    {
        var info = this.GetDomainService().GetQueryData("ProductModel", "ReadProductModel");
        return info.ToJSON();
    }

    [ChildActionOnly]
    public string ProductCategoryData()
    {
        var info = this.GetDomainService().GetQueryData("ProductCategory", "ReadProductCategory");
        return info.ToJSON();
    }
}
```



The base `DataServiceController` is implemented in *RIAPP.DataService.Mvc* class library and implements methods relaying requests to the real data service.

```
public abstract class DataServiceController<T> : Controller
    where T : BaseDomainService
{
    protected abstract IDomainService CreateDomainService();
    private IDomainService _DomainService;

    [ChildActionOnly]
    public string Metadata()
    {
        var info = this.DomainService.ServiceGetMetadata();
        return info.ToJSON();
    }

    [HttpPost]
    public ActionResult GetMetadata()
    {
        var info = this.DomainService.ServiceGetMetadata();
        return Json(info);
    }

    [HttpPost]
    public ActionResult GetItems(GetDataInfo getInfo)
    {
        var res = this.DomainService.ServiceGetData(getInfo);
        return Json(res);
    }

    [HttpPost]
    public ActionResult SaveChanges(ChangeSet changeSet)
    {
        var res = this.DomainService.ServiceApplyChangeSet(changeSet);
        return Json(res);
    }

    [HttpPost]
    public ActionResult RefreshItem(RefreshRowInfo getInfo)
    {
        var res = this.DomainService.ServiceRefreshRow(getInfo);
        return Json(res);
    }
}
```



```

[HttpPost]
public ActionResult InvokeMethod(InvokeInfo invokeInfo)
{
    var res = this.DomainService.ServiceInvokeMethod(invokeInfo);
    return Json(res);
}

protected IDomainService DomainService
{
    get
    {
        if (this._DomainService == null)
        {
            this._DomainService = this.CreateDomainService();
        }
        return this._DomainService;
    }
}

protected T GetDomainService()
{
    return (T)this.DomainService;
}

protected override void Dispose(bool disposing)
{
    if (disposing && this._DomainService != null)
    {
        this._DomainService.Dispose();
        this._DomainService = null;
    }
    base.Dispose(disposing);
}
}
}

```

The metadata definition in the Demo project:

```

<UserControl x:Class="RIAppDemo.BLL.DataServices.RIAppDemoMetadata"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    mc:Ignorable="d"
    xmlns:data="clr-namespace:RIAPP.DataService;assembly=RIAPP.DataService"
    xmlns:dal="clr-namespace:RIAppDemo.DAL;assembly=RIAppDemo.DAL"
    xmlns:models="clr-namespace:RIAppDemo.BLL.Models"
    d:DesignHeight="300" d:DesignWidth="300">
    <UserControl.Resources>
        <data:Metadata x:Key="MainDemo">
            <data:Metadata.DbSets>
                <data:DbSetInfo dbSetName="Customer" insertDataMethod="Insert{0}" updateDataMethod="Update{0}" deleteDataMethod="Delete{0}" enablePaging="True" EntityTypeInfo="{x:Type models:Customer}" />
                <data:DbSetInfo dbSetName="CustomerAddress" insertDataMethod="Insert{0}" updateDataMethod="Update{0}" deleteDataMethod="Delete{0}" enablePaging="True" EntityTypeInfo="{x:Type models:CustomerAddress}" />
                <data:DbSetInfo dbSetName="Address" validateDataMethod="Validate{0}" insertDataMethod="Insert{0}" updateDataMethod="Update{0}" deleteDataMethod="Delete{0}" enablePaging="True" EntityTypeInfo="{x:Type models:Address}" />
                <data:DbSetInfo dbSetName="Product" isTrackChanges="True" validateDataMethod="Validate{0}" refreshDataMethod="Refresh{0}" insertDataMethod="Insert{0}" updateDataMethod="Update{0}" deleteDataMethod="Delete{0}" enablePaging="True" EntityTypeInfo="{x:Type models:Product}" />
                <data:DbSetInfo dbSetName="ProductModel" insertDataMethod="Insert{0}" updateDataMethod="Update{0}" deleteDataMethod="Delete{0}" enablePaging="True" EntityTypeInfo="{x:Type models:ProductModel}" />
                <data:DbSetInfo dbSetName="SalesOrderHeader" insertDataMethod="Insert{0}" updateDataMethod="Update{0}" deleteDataMethod="Delete{0}" enablePaging="True" EntityTypeInfo="{x:Type models:SalesOrderHeader}" />
                <data:DbSetInfo dbSetName="SalesOrderDetail" insertDataMethod="Insert{0}" updateDataMethod="Update{0}" deleteDataMethod="Delete{0}" enablePaging="True" EntityTypeInfo="{x:Type models:SalesOrderDetail}" />
                <data:DbSetInfo dbSetName="ProductCategory" insertDataMethod="Insert{0}" updateDataMethod="Update{0}" deleteDataMethod="Delete{0}" enablePaging="True" EntityTypeInfo="{x:Type models:ProductCategory}" />
                <data:DbSetInfo dbSetName="SalesInfo" enablePaging="True" EntityTypeInfo="{x:Type models:SalesInfo}" />
                <data:DbSetInfo dbSetName="LookUpProduct" enablePaging="True" EntityTypeInfo="{x:Type models:LookUpProduct}" />
                <data:DbSetInfo dbSetName="AddressInfo" enablePaging="False" EntityTypeInfo="{x:Type models:AddressInfo}" />
            </data:Metadata.DbSets>
            <data:Metadata.Associations>
                <data:Association name="CustAddrToCustomer" parentDbSetName="Customer" childDbSetName="CustomerAddress" childToParentName="CustomerAddress" />
                <data:Association name="CustAddrToAddress" parentDbSetName="Address" childDbSetName="CustomerAddress" childToParentName="Address" />
                <data:Association name="CustAddrToAddress2" parentDbSetName="AddressInfo" childDbSetName="CustomerAddress" childToParentName="AddressInfo" />
                <data:Association name="OrdDetailsToOrder" parentDbSetName="SalesOrderHeader" childDbSetName="SalesOrderDetail" childToParentName="SalesOrderHeader" />
                <data:Association name="OrdDetailsToProduct" parentDbSetName="Product" childDbSetName="SalesOrderDetail" childToParentName="Product" />
                <data:Association name="OrdersToCustomer" parentDbSetName="Customer" childDbSetName="SalesOrderHeader" childToParentName="Customer" />
                <data:Association name="OrdersToShipAddr" parentDbSetName="Address" childDbSetName="SalesOrderHeader" childToParentName="Address" />
                <data:Association name="OrdersToBillAddr" parentDbSetName="Address" childDbSetName="SalesOrderHeader" childToParentName="Address" />
            </data:Metadata.Associations>
        </data:Metadata>
        <data:Metadata x:Key="FolderBrowser">
            <data:Metadata.DbSets>
                <data:DbSetInfo dbSetName="FolderBrowser" insertDataMethod="Insert{0}" updateDataMethod="Update{0}" deleteDataMethod="Delete{0}" enablePaging="True" EntityTypeInfo="{x:Type models:FolderBrowser}" />
            </data:Metadata.DbSets>
            <data:Metadata.Associations>
                <data:Association name="FolderBrowserToFolderBrowser" parentDbSetName="FolderBrowser" childDbSetName="FolderBrowser" childToParentName="FolderBrowser" />
            </data:Metadata.Associations>
        </data:Metadata>
    </UserControl.Resources>
</UserControl>

```

Each metadata definition has key (*unique name*) by which it is taken from the control in the DataService's GetMetadata method.


```
protected override Metadata GetMetadata()
{
    return (Metadata)(new RIAppDemoMetadata()).Resources["MainDemo"];
}
```

P.S.: *The above method is executed in the STA thread by the DataService, and the metadata is cached inside the dataservice so the next time retrieval will be faster and consumed less resources.*

The Metadata class contains two collections: *DbSets* and *Associations*. the DbSets collection contains DbSetInfo typed items (*which represent entities*), which in their turn contain collection of FieldInfo items (*which represent entity's fields*). Inside each of the Fieldinfo is defined attributes of the field: its name, its data type and other attributes. Fieldinfo is initialized with default attribute values:

```
this.isPrimaryKey = 0;
this.isRowTimeStamp = false;
this.dataType = DataType.None;
this.isNullable = true;
this.maxLength = -1;
this.isReadOnly = false;
this.isAutoGenerated = false;
this.allowClientDefault = false;
this.dateConversion = DateConversion.None;
this.isClientOnly = false;
this.isCalculated = false;
this.isNeedOriginal = true;
/*
    this.range = null;
    this.regex = null;
*/
```

Their meaning could be understood from their names. But several attributes need explanations:

isPrimaryKey - is an integer typed attribute. So if you have two fields which are in a composite primary key, then for the first field you set isPrimaryKey=1 and for the second isPrimaryKey=2. Each entity must have a primary key to uniquely identify the entity.

Primary key fields are not editable (*should be readonly*), and should be generated on the server when new entity is added. The other way is to assign (*generate values*) primary key fields on a new added entity on the client. But to allow the readonly field assignment on the client (*only for new entities*) you should make **allowClientDefault=true**. It allows generating new values for the field on the client (*for newly added entities only*).

dateConversion - determines the server to client date values conversions. It is meaningfull only for Date fields. You can choose between three values.

```
public enum DateConversion : int
{
    None=0, ServerLocalToClientLocal=1, UtcToClientLocal=2
}
```

The first option is no conversion. The remaining options take server and client timezones into considerations. For example, if you choose *ServerLocalToClientLocal* then the date values will be converted from server local time to client local time (*and vice versa*).

if you choose `UtcToClientLocal` (*but make sure the dates on the server are really UTC*), then the dates values will be converted from UTC to client local time zone (*and vice versa*). This is helpful for distributed applications, because clients and servers can be in different time zones.

`isNeedOriginal` - the default is true. Typically setting it to false conserves a little of bandwidth. But it can be done judicially, because if you set *isNeedOriginal* to false for the fields which needs original values to check row modifications (*optimistic concurrency*) when applying updates- the update will fail, saying that the row was modified before you applied updates. So if you want less unpredictable behaviour, then I recommend, leave the attributes to their default- *true* values.

`isAutoGenerated` - says that the field does not accept updates from the client (*it ignores them*), and generates values on the server.

`isRowTimeStamp` - if set to true is used to mark the field that it always needs original values for updates, and the field always returns to the client updated value from the server. It is typically `readOnly` on the client. You don't need to set for them `isAutoGenerated = true`, because they are autogenerated by themselves (*but it will do no harm*).

`range` - the attribute sets validation checks for numeric range of the value as

`range="100,5000"` or for Dates as `range="2000-01-01,2015-01-01"`

`regex` - the attributes sets validation checks for string values by the regex expression, like in `regex="^[a-z0-9-]+(\\.[a-z0-9-]+)*@[a-z0-9-]+(\\.[a-z0-9-]+)*(\\.[a-z]{2,4})$"`

Associations

The Associations collection contains `Association` typed items.

The association defines foreign key references and navigation fields names.

Association's properties:

Property	Description
<code>name</code>	unique name of the association. could be used to get association by its name on the client.
<code>parentDbSetName</code>	DbSetName of the parent DbSet
<code>childDbSetName</code>	DbSetName of the child DbSet
<code>childToParentName</code>	The name which will be added as a navigation property on child entities to refer to parent entity. P.S.- <i>The navigation property appends automatically to entity. Their name are defined only in association metadata.</i>
<code>parentToChildrenName</code>	The name which will be added as a navigation property on parent entities to refer to children entities. P.S.- <i>The navigation property appends automatically to entity. Their name are defined only in association metadata.</i>
<code>onDeleteAction</code>	Could be set to <code>NoAction</code> , <code>Cascade</code> or <code>SetNulls</code> . It determines what client will do when a parent entity is deleted. <code>NoAction</code> - means you manually can delete its children.
<code>fieldRels</code>	A collection of key to key relations between parent and children.

The associations defined in metadata typically used only on the client for using navigation properties. But they can be used to load from the dataservice full hierarchy of entities.

For example, for Customer - CustomerAddress - Address relationship.

```
<data:Association name="CustAddrToCustomer" parentDbSetName="Customer"
childDbSetName="CustomerAddress" childToParentName="Customer"
parentToChildrenName="CustomerAddresses" >
  <data:Association.fieldRels>
    <data:FieldRel parentField="CustomerID" childField="CustomerID"></data:FieldRel>
  </data:Association.fieldRels>
</data:Association>

<data:Association name="CustAddrToAddress" parentDbSetName="Address" childDbSetName="CustomerAddress"
childToParentName="Address" parentToChildrenName="CustomerAddresses">
  <data:Association.fieldRels>
    <data:FieldRel parentField="AddressID" childField="AddressID"></data:FieldRel>
  </data:Association.fieldRels>
</data:Association>
```

We can use query method to load associated entities if our custom includeNav parameter was set to true on client. The method inspects this parameter, and if it was set to true, it creates QueryResult with navigation hierarchy which should be loaded to the client with the Customer entity.

```
[Query]
public QueryResult<Customer> ReadCustomer(GetDataInfo getInfo, bool? includeNav)
{
    string[] includeHierarchy = new string[0];
    if (includeNav == true)
    {
        DataLoadOptions opt = new DataLoadOptions();
        opt.LoadWith<Customer>(m => m.CustomerAddresses);
        opt.LoadWith<CustomerAddress>(m => m.Address);
        this.DB.LoadOptions = opt;

        //we can conditionally include entity hierarchy into results
        //making the path navigations decisions on the server enhances security
        //we can not trust clients to define navigation's expansions because it can influence the server performance
        //and is not good from security's standpoint
        includeHierarchy = new string[] { "CustomerAddresses.Address" };
    }

    int? totalCount = null;
    var res = QueryHelper.PerformQuery(this.DB.Customers, getInfo, ref totalCount).AsEnumerable();
    return new QueryResult<Customer>(res, totalCount, includeHierarchy);
}
```

But using this feature, it can cause slower query execution in real world scenarios. In many cases it is better to load child and parent entities from the client using separate queries.

For example, we could load CustomerAddress entities by an array of CustomerIDs.

```
[Query]
public QueryResult<CustomerAddress> ReadAddressForCustomers(GetDataInfo getInfo, int[] custIDs)
{
    int? totalCount = null;
    var res = this.DB.CustomerAddresses.Where(ca => custIDs.Contains(ca.CustomerID));
    return new QueryResult<CustomerAddress>(res, totalCount);
}
```

and then load addresses by their IDs.

```
[Query]
```

```

public QueryResult<Address> ReadAddressByIds(GetDataInfo getInfo, int[] addressIDs)
{
    int? totalCount = null;
    var res = this.DB.Addresses.Where(ca => addressIDs.Contains(ca.AddressID));
    return new QueryResult<Address>(res, totalCount);
}

```

Authorization

can be applied on two levels - data service class level and method level.

Authorization is applied by adding `Authorize` attribute to the whole data service or a data service's method. `Authorize` attribute can include roles. Without including role it is simply checked that the user is authenticated. the `Authorize` attribute is optional, and when it is not applied then it is assumed that the access is allowed on that level.

The access is first checked at the data service level, and if it is not allowed, further checks is not made.

The only exeption is when on the method is applied `AllowAnonymous` attribute. In that case the access to the method is allowed, irrespective of the data service level authorization checks.

The next level (*method's level*) encompasses query methods, CRUD methods, Refresh and service (*invoke*) methods attributes checks.

If the user belongs to one of the roles in the roles array then the access is allowed.

```

[Authorize()]
public class RIAppDemoService : LinqForSqlDomainService<RIAppDemoDataContext>
{
    ///other services methods are not shown here (only two methods with attributes)

    [AllowAnonymous()]
    [Query]
    public QueryResult<ProductModel> ReadProductModel(GetDataInfo getInfo)
    {
        int? totalCount = null;
        var res = QueryHelper.PerformQuery(this.DB.ProductModels, getInfo, ref totalCount).AsEnumerable();
        return new QueryResult<ProductModel>(res, totalCount);
    }

    [Authorize(Roles = new string[] { ADMIN_ROLE })]
    public void UpdateSalesOrderDetail(SalesOrderDetail soDetail)
    {
        SalesOrderDetail orig = this.GetOriginal<SalesOrderDetail>();
        this.DB.SalesOrderDetails.Attach(soDetail, orig);
    }
}

```

The two level authorization scheme is convenient because the minimum access rights to the service can be applied on the service level (*for example, that the user should be authenticated*), and individual methods could enforce that the method can be accessed only by certasin roles, or be allowed unauthenticated access to them.

The described behaviour of the authorization is applied only to out of the box features. The authorization can be extended (*or replaced*) by creating a custom authorizer which implements `IAuthorizer` interface.

```

public interface IAuthorizer
{

```

```

void CheckUserRightsToExecute(IEnumerable<MethodInfo> methods);
void CheckUserRightsToExecute(MethodInfo method);
System.Security.Principal.IPrincipal principal { get; }
Type serviceType { get; }
}

```

The BaseDomainService class has a virtual method which creates and returns an authorizer instance. This method can be overridden in the descendants.

```

protected virtual IAuthorizer GetAuthorizer()
{
    return new Authorizer(this.GetType(), this.CurrentPrincipal);
}

```

There are two more features which a real world LOB application should care about: [Change tracking](#) and [Error logging](#).

Tracking changes

The BaseDomainService has virtual method [OnTrackChange](#) which can be overridden in the DomainService and can be used to obtain a diffgram (*xml representation*) of changes for an entity on which change tracking is enabled in the metadata ([isTrackChanges="True"](#)).

```

/// <summary>
/// here can be tracked changes to the entities
/// for example: product entity changes is tracked and can be seen here
/// </summary>
protected override void OnTrackChange(string dbSetName, ChangeType changeType, string diffgram)
{
}

```

an example of the diffgram on a Product entity:

```

<changes>
  <Name old="Classic Vest, L2" new="Classic Vest, L3" />
  <StandardCost old="23.749" new="23.74" />
  <ListPrice old="63.5" new="100" />
  <Size old="L" new="M" />
</changes>

```

Error logging

The Error logging can be implemented in the data service by overriding [OnError](#) data service method and logging errors in there.

```

/// <summary>
/// Error logging could be implemented here
/// </summary>
/// <param name="ex"></param>
protected override void OnError(Exception ex)
{
}

```

One more method which can be overridden in the DataService is a Dispose method, which is used to clean up resources.

an example of a Dispose method override:

```
protected override void Dispose(bool isDisposing)
{
    if (this._connection != null)
    {
        this._connection.Close();
        this._connection = null;
    }

    base.Dispose(isDisposing);
}
```