jRIAppTS, the RIA framework – user's guide

jRIAppTS, the RIA framework

1.1 *What is the jRIAppTS framework*

jRIAppTS – is an application framework for developing rich internet applications - RIA's. It consists of two parts – a client and a server parts. The client part was written in typescript language. The server part was written in C# and the demo application (*RIAppDemo*) was implemented in ASP.NET MVC.
The Server part resembles Microsoft WCF RIA services, featuring data services which is consumed by clients.
The Client part resembles Microsoft Silverlight client development, only it is based on HTML (*not XAML*), and uses typescript language for coding.
The framework was designed primarily for creating data centric Line of Business (*LOB*) applications which will work natively in browsers without the need for plugins .
The framework supports a wide range of essential features for creating LOB applications, such as, declarative use of *databindings*, integration with the server side dataservice, data templates, client side and server side data validation, localization, authorization, and a set of UI controls, like the *datagrid*, the *stackpanel* , the *dataform* 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 framework was designed for gaining maximum convenience and performance, and for this sake it works in browsers which support ECMA Script 5.1 level of javascript.

Supported browsers include 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 stack implementation of the features required for building real world LOB applications in HTML5. It allows the development in strongly typed environment either on the client or on the server.

Data centric applications are created by using framework's wide range of UI controls. It allows to work with the server originated data in a transparent and a safe way.

The framework contains a set of controls such as:
A *DataGrid* – the control for displaying and editing the data in the table form. It supports databinding, row selection with keyboard keys, sorting by column, data paging, a detail row, data templates, different column's types (*expander column, row selector column, actions column*). For editing it can use the built-in inline editor, and also has the support for a popup editor which uses a data template for its content display.
A *StackPanel* - the control for displaying and editing of the data as a horizontal or vertical list . It uses a data template for its items' display and also has the support for items' selections with the help of keyboard keys and the mouse.
A *ListBox* - the control which encapsulates the HTML select tag and attaches to it the logic to draw the data from the collection type datasource.
A *DataForm* - the control which bounds a datacontext to a region and allows to use special contents elements inside this region. It also provides for summary error display.

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

The framework also has a special element view registered by the name *dynacontent*, which helps to create content regions on the page using data templates. The templates in these regions are easily switchable. This feature enables to create complex single page applications, because a template can represent a whole web page and templates can contain other children templates.

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

1.2 *Licensing*

1.3 *The Framework's files and deployment*

The framework is written in typescript language. So, the typescript code is compiled first into javascript before its use in HTML5 applications.
The typescript code is contained in the Microsoft Visual Studio Project with the name *jriappTS*.
The main file app.ts, contains the Application class and has references to other core files (*modules*) of the framework. When the project is compiled its Post Build Event executes a command: *"C:\Program Files (x86)\Microsoft SDKs\TypeScript\1.6\tsc.exe" "$(ProjectDir)main\app.ts" --d --noImplicitAny --removeComments --target ES5 --out "$(ProjectDir)\..\built\jriapp.js"*
At the end of the compilation in the Project's *built* directory will appear two files: *jriapp.js* and *jriapp.d.ts*.

Note: *There is also a jriapp_strings.d.ts file. This file is a type definition for string resources used in the framework. It reflects structure of lang files like jriapp_en.js.*
You can use node.js to compile the project or use another IDE like jetbrains WebStorm to compile the typescript files.

All the application modules referenced in the *app.ts* file are compiled into a single *jriapp.js* file.

The app.ts has the next references:

```
/// <reference path="..\thirdparty\jquery.d.ts" />
/// <reference path="..\thirdparty\jquery.d.ts" />
/// <reference path="..\thirdparty\moment.d.ts" />
/// <reference path="..\thirdparty\qtip2.d.ts" />
/// <reference path="..\jriapp_strings.d.ts"/>
/// <reference path="consts.ts"/>
/// <reference path="interface.ts"/>
/// <reference path="core_modules.ts"/>
/// <reference path="baseutils.ts"/>
/// <reference path="baseobj.ts"/>
/// <reference path="globalobj.ts"/>
/// <reference path="..\modules\utils.ts"/>
/// <reference path="..\modules\errors.ts"/>
/// <reference path="..\modules\converter.ts"/>
/// <reference path="..\modules\defaults.ts"/>
/// <reference path="..\modules\parser.ts"/>
/// <reference path="..\modules\mvvm.ts"/>
/// <reference path="..\modules\baseElView.ts"/>
/// <reference path="..\modules\binding.ts"/>
/// <reference path="..\modules\template.ts"/>
/// <reference path="..\modules\baseContent.ts"/>
//*** the rest are optional modules, which can be removed if not needed ***
/// <reference path="..\modules\collection.ts"/>
/// <reference path="..\modules\dataform.ts"/>
/// <reference path="..\modules\dynacontent.ts"/>
/// <reference path="..\modules\datepicker.ts"/>
/// <reference path="..\modules\tabs.ts"/>
/// <reference path="..\modules\listbox.ts"/>
/// <reference path="..\modules\datadialog.ts"/>
/// <reference path="..\modules\datagrid.ts"/>
/// <reference path="..\modules\pager.ts"/>
/// <reference path="..\modules\stackpanel.ts"/>
/// <reference path="..\modules\db.ts"/>
```

The client part of the framework is usually deployed (*as in the demo application*) in one folder, *jriapp*, which is located in the *Scripts* web application folder (*it can be renamed if desired*).
In the *jriapp* folder is also a *jriapp.css* (*css styles for the frameworks's UI controls*), *jriapp_en.js* (*a localization file which contains strings used in the framework*) and the *img* folder – which contains image files used by the framework's controls (*which can be replaced with custom ones*).

The demo application which demonstrates capabilities of the framework (*and was also used to test features*) was created using ASP.NET MVC web site project. The project uses a layout page (*_LayoutDemo.cshtml*), which is used by all pages included in the demo web site (*RIAppDemo Visual Studio ASP NET MVC project*).
The ***demoTS*** typescript project contains user modules, and on compilation it produces javascript files which are used in the demo ASP.NET MVC web site. They use traditional module format and are referenced on pages using traditional script tags (*synchronous loading*)
The ***spaAMD*** typescript project also contains user modules, but they are used only for Single Page Application demo page (*included in the demo project*). Those modules use AMD module format, and are loaded by an AMD loader (*require.js*) asynchronously.

The layout page includes core files of the framework (*jriapp.js, jriapp_en.js and jriapp.css*) and several javascript and css files which are used by the demo pages (*jquery.js, bootstrap.js, qtip.js, moment.js*).

Javascript and CSS files references in the _LayoutDemo.cshtml page:

```
<link href="@Url.Content("~/Scripts/bootstrap/css/bootstrap.min.css",true)" rel="stylesheet" type="text/css" />
 <link href="@Url.Content("~/Content/themes/redmond/jquery-ui-1.10.4.custom.min.css")" rel="stylesheet"
type="text/css" />
 <link href="@Url.Content("~/Scripts/qtip/jquery.qtip.min.css",true)" rel="stylesheet" type="text/css" />
 <link href="@Url.Content("~/Scripts/jriapp/jriapp.css",true)" rel="stylesheet" type="text/css" />
 <link href="@Url.Content("~/Content/Site.css",true)" rel="stylesheet" type="text/css" />

 <script src="@Url.Content("~/Scripts/jquery/jquery-1.10.2.js")" type="text/javascript"></script>
 <script src="@Url.Content("~/Scripts/bootstrap/js/bootstrap.min.js",true)" type="text/javascript"></script>
 <script src="@Url.Content("~/Scripts/jquery/jquery-ui-1.10.4.custom.min.js")" type="text/javascript"></script>
  <script src="@Url.Content("~/Scripts/qtip/jquery.qtip.min.js",true)" type="text/javascript"></script>
  <script src="@Url.Content("~/Scripts/moment/moment.min.js",true)" type="text/javascript"></script>
  <script src="@Url.Content("~/Scripts/jriapp/jriapp_en.js",true)" type="text/javascript"></script>
  <script src="@Url.Content("~/Scripts/jriapp/jriapp.js",true)" type="text/javascript"></script>
```

The framework is dependent on several third party javascript libraries: JQuery (*1.7 and higher*), JQuery UI (*for calendars, tabs and the other UI controls*), moment (*for date formatting*), and qtip (*for tooltips)*. Thus, these javascript libraries must always be referenced on a HTML page for the framework's code to work properly. But the framework is designed so these dependences can be easily swapped for other similar ones.

Note: *The bootstrap.js is not required for the framework's functionality. It is included as a library for helping layout on the pages, it also contains a Menu implementation used in the Demo. You can use any javascript libraries with the framework which you can find useful.*

The demo pages in addition to the files included in the layout page also include some specific code for the page - such as the code which contains view models, converters, an application class, css styles and so on.

For example, the *DataGridDemo.cshtml* page contains these references:

```
 <script src="@Url.Content("~/Scripts/RIAppDemo/common.js",true)" type="text/javascript"></script>
 <script src="@Url.Content("~/Scripts/RIAppDemo/header.js",true)" type="text/javascript"></script>
 <script src="@Url.Content("~/Scripts/RIAppDemo/demoDB.js",true)" type="text/javascript"></script>
 <script src="@Url.Content("~/Scripts/RIAppDemo/gridDemo.js",true)" type="text/javascript"></script>
```

Note: *The pages which consume the data service also include a code generated from the dataservice. It contains generated classes and interfaces, strongly typed entities and DbSets and the DbContext class (client side domain model).*

The generated classes are used as a client side domain model and to communicate with the DataService. In addition to the client side checks, the DataService always performs the checks on the server side on submits from the clients.

For example, in the *DataGridDemo.cshtml* the page references the file *~/Scripts/RIAppDemo/demoDB.js* which contains the code that was generated (*it is the client side domain model*) and contains entity types, DbSets, exported interfaces of the objects from the server side, and the strongly typed DbContext class - to communicate with the DataService.

The *~/Scripts/RIAppDemo/gridDemo.js* file contains user defined view models (*used for the MVVM design pattern*) and an application class derived from the framework's *RIAPP.Application* class and depends on classes defined in the *demoDB.js*.

But the *SPADemo.cshtml* page uses another approach to load modules. It loads user modules with the help of *require.js* library.

```
<script type="text/javascript">
        var require ={
            baseUrl: "@Url.Content("~/Scripts/RIAppDemo/SPA/")",
            urlArgs: "bust=v2.0.6.0",
            waitSeconds: 10
        };
 </script>
 <script data-main="SPAConfig.js" type="text/javascript" src="@Url.Content("~/Scripts/require.js",true)"></script>
```

The client side code which users write for developing HTML5 applications consists of *view models*, which are simple javascript objects which expose properties which can be data bound on the HTML page (*using data-bind attribute*). Also, the *view models* can expose commands so that the HTML controls like buttons or anchors could invoke actions (*wrapped in the commands*) on clicking by the user.
The other important pieces of the web application are *element views*, which are roughly equivalent to *AngularJS* directives. They give a way to attach javascript code (*usually, UI controls*) to HTML elements in a declarative way by using a *data-view* attribute for that.
There is of course more than what is said above what you can do with the help of the framework. These things will be explained in more details further in this guide.

Note: *Also, a good thing to learn how the framework works is to see how the demo web application works and to look at its code. It is also good to use Firebug debugger in the Mozilla Firefox browser to further study its working.*

The framework's classes

2.1 *BaseObject class*

The framework uses *BaseObject* as a base class for all classes in the framework.
The *RIAPP.BaseObject* class is defined in a *baseobj.ts* file and this class adds a common logic for an object's destruction and adds support for events in all the objects derived from it.

BaseObject's methods:

Methods	Description
addHandler	adds an event handler for an event (*with optional support for event namespaces like in jQuery*)
removeHandler	removes a handler for an event (*can be used to remove all handlers registered within a namespace*)
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 (*but don't forget to always invoke super mehod!)*
_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.
handleError	it is typically invoked in descendants of the BaseObject on error conditions. It triggers an error event and returns a boolean value of handling result. Many framework's object types override this method, for example, in BaseElView it calls handleError when catching exceptions, and if the error was not handled in this class it calls the application's handleError method. If the application's handleError (*in error event*) method does not handle the error then the global object's handleError will be invoked as a last resort. You can subscribe to the error event on the application and the global objects to handle errors (*and show the error dialog*)
getIsDestroyed	Used to check that the object is destroyed or not.
getIsDestroyCalled	Used to check that the object is destroyed or is destroying.

BaseObject's events:

Event name	Description
error	event is Triggered from the BaseObject's handleError method.
destroyed	event is Triggered when the object's destroy method completed

BaseObject's properties:

Property	Description
_isDestroyed	It is set to true in the BaseObject's destroy method when the destruction of the object is completed. This field has a protected access. For external access to this property use getIsDestroyed method.
_isDestroyCalled	It is set to true (*typically in the descendants*) when the destroy method was called. The destruction of the object is not completed, but it is in the progress. This field has a protected access. For external access to this property use getIsDestroyCalled method.

The *BaseObject* implements the next interfaces:

```
export interface IDisposable {
        destroy(): void;
        getIsDestroyed(): boolean;
        getIsDestroyCalled(): boolean;
        addOnDestroyed(handler: TEventHandler<any, any>, nmspace?: string, context?: BaseObject): void;
        removeOnDestroyed(nmspace?: string): void;
}

export interface IErrorHandler {
   handleError(error: any, source: any): boolean;
}
export interface IBaseObject extends IErrorHandler, IDisposable {
    raisePropertyChanged(name: string): void;
    addHandler(name: string, handler: TEventHandler<any, any>, nmspace?: string, context?: BaseObject, prepend?: boolean): void;
    removeHandler(name?: string, nmspace?: string): void;
    addOnPropertyChange(prop: string, handler: TPropChangedHandler, nmspace?: string, context?: BaseObject): void;
    removeOnPropertyChange(prop?: string, nmspace?: string): void;
    removeNSHandlers(nmspace?: string): void;
    addOnError(handler: TErrorHandler, nmspace?: string, context?: BaseObject): void;
    removeOnError(nmspace?: string): void;
    raiseEvent(name: string, args: any): void;
}
```

Each object has *getIsDestroyCalled* method which indicates if the object is invalid state. You can check this field's value, to be sure that the object is still in valid state after some asynchronous operation is completed (*because in the meantime the object can be disposed*), like in the next example:

```
setTimeout(function () {
          if (self.getIsDestroyCalled())  //the object state is invalid
             return;
          self._checkQueue(property, self._owner[property]);
 }, 0);
```

The *BaseObject* class also has a *getIsDestroyed* method (*and an _isDestroyed field*), which is set to true after the object's destruction is complete. This field is usually checked in overridden destroy methods, so when the object is destroyed to exit the destroy method immediately, like in the next example:

```
destroy() {
          if (this._isDestroyed) //checks, if already destroyed just return and do nothing
             return;
          this._isDestroyedCalled   = true; //set this field to inform that the destruction is started
          this._unbindDS();
          this._clearContent();
          this._$el.removeClass(_css.pager);
          this._el = null;
          this._$el = null;
          super.destroy();
 }
```

The initialization of a new object's instance is done in the object's constructor.

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

```
    export class TestObject extends BaseObject {
          _testProperty1: string;
          _testProperty2: string;
          _testCommand: MOD.mvvm.ICommand;
          _month: number;
          _months: DEMODB.KeyValDictionary;
          _format: string;
          _formats: DEMODB.StrKeyValDictionary;

          constructor(initPropValue: string) {
             super();
             var self = this;
             this._testProperty1 = initPropValue;
             this._testProperty2 = null;
             this._testCommand = new MOD.mvvm.Command(function (sender, args) {
                 self._onTestCommandExecuted();
             }, self,
              function (sender, args) {
                    //if this function return false, then the command is disabled
                    return utils.check.isString(self.testProperty1) && self.testProperty1.length > 3;
                });

                this._month = new Date().getMonth() + 1;
                this._months = new DEMODB.KeyValDictionary();
                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 = new DEMODB.StrKeyValDictionary();
                this._formats.fillItems([{ key: 'PDF', val: 'Acrobat Reader PDF' }, { key: 'WORD', val: 'MS Word DOC' }, { key:
'EXCEL', val: 'MS Excel XLS' }], true);
            }
            _onTestCommandExecuted() {
                alert(utils.format("testProperty1:{0}, format:{1}, month: {2}", this.testProperty1, this.format, this.month));
            }
            get testProperty1() { return this._testProperty1; }
            set testProperty1(v) {
                if (this._testProperty1 != v) {
                    this._testProperty1 = v;
                    this.raisePropertyChanged('testProperty1');
                    //let the command to reevaluate preconditions of its executing
                    this._testCommand.raiseCanExecuteChanged();
                }
            }
            get testProperty2() { return this._testProperty2; }
            set testProperty2(v) {
                if (this._testProperty2 != v) {
                    this._testProperty2 = v;
                    this.raisePropertyChanged('testProperty2');
                }
            }
            get testCommand() { return this._testCommand; }
            get testToolTip() {
                return "Click the button to execute the command.<br/>" +
                        "P.S. <b>command is active when the testProperty length > 3</b>";
            }
            get format() { return this._format; }
            set format(v) {
                if (this._format !== v) {
                    this._format = v;
                    this.raisePropertyChanged('format');
                }
            }
            get formats() { return this._formats; }
            get month() { return this._month; }
            set month(v) {
                if (v !== this._month) {
                    this._month = v;
                    this.raisePropertyChanged('month');
                }
            }
            get months() { return this._months; }
        }
```

An object's instance can trigger two predefined events: '*error*', '*destroyed*'.

An *error* event is triggered from the BaseObject's *handleError* method.
The error event handler can set *isHandled* to tru, and after that the error handling is successfully finished without going up the stack.

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

You can override the *BaseObject's* *_getEventNames* method in order to define new custom events in derived classes, as in the example:

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

Users can subscribe to events by using an *addHandler* method, as in the example:

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

Note: *Many framework's classes have strongly typed versions of methods to add and remove event handlers to simplify event subscriptions, like these ones:*

```
 addOnStatusChanged(fn: (sender: BaseCollection<TItem>, args: ICollItemStatusArgs<TItem>) => void, nmspace?:
string)
{
        this.addHandler('status_changed', fn, nmspace);
}
 removeOnStatusChanged(nmspace?: string) {
        this.removeHandler('status_changed', nmspace);
}
```

The last argument of the *addHandler* method is an event's *namespace*, which is an optional parameter and can be used to selectively remove subscriptions. For this purpose you can use the *removeNSHandlers* method. To remove subscriptions to events you can also use the *removeHandler* method, and supply the event's name to it (*and optionally the event's namespace*).

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

An event can be triggered by using the *raiseEvent* method, as in the example:

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

You can also subscribe to property change notifications using an *addOnPropertyChange* method, as in the example:

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

If you want to get notifications for all property changes you can supply '*' instead of a real property name. Inside the handler you can obtain the name of the property which triggered the notification by using *args.property* value, as in the example:

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

In order to unsubscribe from a property change notification you can use a *removeOnPropertyChange* method, or use a *removeNSHandlers* method, as in the example:

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

2.2 *Global class*

All the code in the client side part of the framework is structured into modules (*namespaces*). The *Global* object's instance (*an instance of a RIAPP.Global class*) is created by the framework on time of loading of the framework's jriapp.js file. It is a singleton object. The RIAPP.Global class is defined in *globalobj.ts* file. An instance of this object can be accessed in the code via the exported *RIAPP.global* variable. It can also be accessed by the global property on the Application's object instance.

The *RIAPP.global* - is used to hold references for application instances and it manages subscriptions to several *window.document*'s events.
It also subscribes to *window.onerror* event to handle unhandled errors (*Shows an alert window*). It also dispatches window document's *keydown* events to an UI control (*a DataGrid or a StackPanel*) currently selected on the HTML page, so that only the selected (*focused*) instance of the user control can handle keyboard events (*for example,it is used for selecting a row in the DataGrid with the help of up or down keyboard keys when the datagrid in its focused state*). The Global object exposes the *load* event which is triggered when all the HTML DOM is parsed (*like the jQuery's ready method*).

For convenience, the global object exposes references to the window HTML DOM object and to the jQuery function. A global object also holds references for registered converters (*some converters are registered globally in the converter module*).

The global object has the *defaults* property, which exposes an instance of the Defaults object class (*it is instantiated in the defaults core module*). Using this property you can set or change the default values used in the framework, such as: default date format, time format, decimal point, thousand's separator, *datepicker*'s defaults, path to the images, as in the example:

```
global.defaults.dateFormat = 'DD.MM.YYYY'; //russian style date format which is the default
global.defaults.imagesPath = '/Scripts/jriapp/img/';
(<any>global.$).datepicker.regional['ru'] = {
                closeText: 'Закрыть',
                prevText: '&#x3c;Пред',
                nextText: 'След&#x3e;',
                currentText: 'Сегодня',
                monthNames: ['Январь', 'Февраль', 'Март', 'Апрель', 'Май', 'Июнь',
                  'Июль', 'Август', 'Сентябрь', 'Октябрь', 'Ноябрь', 'Декабрь'],
                monthNamesShort: ['Янв', 'Фев', 'Мар', 'Апр', 'Май', 'Июн',
                  'Июл', 'Авг', 'Сен', 'Окт', 'Ноя', 'Дек'],
                dayNames: ['воскресенье', 'понедельник', 'вторник', 'среда', 'четверг', 'пятница', 'суббота'],
                dayNamesShort: ['вск', 'пнд', 'втр', 'срд', 'чтв', 'птн', 'сбт'],
                dayNamesMin: ['Вс', 'Пн', 'Вт', 'Ср', 'Чт', 'Пт', 'Сб'],
                dateFormat: 'dd.mm.yy',
                firstDay: 1,
                isRTL: false
 };
```

```
global.defaults.datepicker.datepickerRegion = 'ru';
```

The date formats use a *moment.js* format style.
But the defaults for the *datepicker* use a jQuery UI datepicker's date format style.

Global object's methods:

Method	Description
findApp	Finds an application instance by its name.
registerType	Registers a type by its name. The type can be later retrieved anywhere in the code by using *getType* method.
getType	Retrieves the registered type by its name.
registerConverter	Registers a converter by its name. The registered converters can be used in the databinding's expressions by their names. P.S. - *The Application class also has the registerConverter method. If you register a converter with an application by the same name as in global class then it will be used by the databindings of this application.*
registerElView	Registers an element view by its name. The registered element's views are used directly (*using their names*) or indirectly in the databindings.
getImagePath	Get common images paths used in the framework by their names. It just appends a provided image name to the default path for the images. It is a helper method.
loadTemplates	Loads templates as a batch (*as a file with templates*) from the server. They are later available to all application instances. It should be used only before the application instance is created. Usually it is done in the *global.onload* event handler.
addOnUnResolvedBinding	Adds an event handler which will be executed on each currently unresolved databinding path. This is used for debugging to check that the declaratively defined databindings are properly resolved. P.S.- *you can see how it is used in the demo.*

Global object properties:

Property	ReadOnly	Description
$	Yes	JQuery function for easier access in the code.
window	Yes	DOM Window object instance
document	Yes	DOM Document object instance
currentSelectable	No	A currently selected control a *DataGrid* or a *StackPanel* which accepts keyboard input like Up or Down keys for scrolling in them by the keyboard keys. It is set automatically when the control is clicked on a page. P.S.- *you can set it in the code, to make sure that the control has the keyboard input.*
defaults	Yes	An Instance of the *Default* object type, to access or to change the default values.
UC	--	the namespace (*empty object instance*) for attaching any custom code. You can attach any code to it which can

		be accessed globally.
utils	Yes	An Object which contains common utility methods.
isLoading	Yes	Returns true if an application's instance or the global object is still loading templates from the server.

Global object events:

Event name	Description
unload	Triggered when the browser's window unloads.
load	Triggered when the document DOM structure is fully loaded. It is used like JQuery.ready event handler. This event is used to create an application instance in its event handler. Because it marks the time when all javascript modules are loaded and the Global object instance is ready.

2.3 *Application class*

The *RIAPP.Application* - class, an instance of which represents the main context of an application on the page.
On creation of an application's instance you can provide an options object to the constructor. The options interface is defined as:

```
export interface IAppOptions {
	application_name?: string;
	user_modules?: { name: string; initFn: (app: Application) => any; }[];
	application_root?: Document | HTMLElement;
}
```

The options include a name for the application (*can be used if you have several applications on a HTML page, if not then the default is OK*). It also includes an array of objects for user modules initialization. The *initFn* is invoked when the user module is initialized by the application.
Also, the options can provide the application's root. It is a scope (*a region*) of the application on the HTML page. By default, the whole HTML document is the scope and the application root refers to the window.document property. But if you have several applications on a HTML page you can provide different scopes (*usually, a div element*) for each application.

The main method of the application is the *startUp* method, which is used to trigger execution of a callback function (*which is provided as an argument*) . It also performs the data binding for the application region.
The callback function used as a sandbox environment, in which instances of view models (*they are usually defined in user modules*) can be created (*but they are better be created in the application's onStartup method which can be overridden in derived classes*).
Any other user defined objects can be created there too. After executing the callback function, the application invokes the protected *onStartup* method (*which can be overridden in derived application classes*) and then the application performs data binding on the HTML page.

Usually every SPA (*single page application*) uses a specialized application class (*derived from the Application class*), which is defined in a custom user module. It can also accept an extended options object.

For example, the datagrid demo example extends its application options by adding more properties to it.

```
 export interface IMainOptions extends IAppOptions {
        service_url: string;
        permissionInfo?: MOD.db.IPermissionsInfo;
        images_path: string;
        upload_thumb_url: string;
        templates_url: string;
        productEditTemplate_url: string;
        sizeDisplayTemplate_url: string;
        modelData: any;
        categoryData: any;
        sse_url: string;
        sse_clientID: string;
 }
```

The demo uses a derived application's class which exposes instances of view models (*which are defined in that or in other modules*) and it also exposes an instance of DbContext (*for communication with the data service*).

```
    //strongly typed aplication's class
    export class DemoApplication extends Application {
        _dbContext: DEMODB.DbContext;
        _errorVM: COMMON.ErrorViewModel;
        _headerVM: HEADER.HeaderVM;
        _productVM: ProductViewModel;
        _uploadVM: UploadThumbnailVM;
        _sseVM: SSEVENTS.SSEventsVM;
        _sseMessage: string;

        constructor(options: IMainOptions) {
            super(options);
            var self = this;
            this._dbContext = null;
            this._errorVM = null;
            this._headerVM = null;
            this._productVM = null;
            this._uploadVM = null;
            this._sseVM = null;
        }
        onStartUp() {
            var self = this, options: IMainOptions = self.options;
            this._dbContext = new DEMODB.DbContext();
            this._dbContext.initialize({ serviceUrl: options.service_url, permissions: options.permissionInfo });
            function toText(str) {
                if (str === null)
                    return '';
                else
                    return str;
            };

            this._dbContext.dbSets.Product.defineIsActiveField(function (item) {
                return !item.SellEndDate;
            });
            this._errorVM = new COMMON.ErrorViewModel(this);
            this._headerVM = new HEADER.HeaderVM(this);
            this._productVM = new ProductViewModel(this);
            this._uploadVM = new UploadThumbnailVM(this, options.upload_thumb_url);
            function handleError(sender, data) {
                self._handleError(sender, data);
            };
            //here we could process application's errors
            this.addOnError(handleError);
            this._dbContext.addOnError(handleError);
            if (!!options.sse_url && SSEVENTS.SSEventsVM.isSupported()) {
                this._sseVM = new SSEVENTS.SSEventsVM(options.sse_url, options.sse_clientID);
            this._sseVM.addOnMessage((s, a) => { self._sseMessage = a.data.message;
self.raisePropertyChanged('sseMessage'); });
            this._sseVM.open().fail((err) => { self._handleError(self._sseVM, { error: err.message }); });
        }

        //adding event handler for our custom event
        this._uploadVM.addOnFilesUploaded(function (s, a) {
            //need to update ThumbnailPhotoFileName
            a.product._aspect.refresh();
        });

        if (!!options.modelData && !!options.categoryData) {
            //the data was embedded into HTML page as json, just use it
            this.productVM.filter.modelData = options.modelData;
            this.productVM.filter.categoryData = options.categoryData;
            this.productVM.load().done(function (loadRes) {
                //alert(loadRes.outOfBandData.test);
                return;
            });

        }
        else {
            //there was no embedded data for the filter on the HTML page, so load it first, and then load products
            this.productVM.filter.load().done(() => {
                self.productVM.load().done(function (loadRes) {/*alert(loadRes.outOfBandData.test);*/ return; });
            });
        }
        super.onStartUp();
    }
    private _handleError(sender, data) {
        debugger;
        data.isHandled = true;
        this.errorVM.error = data.error;
        this.errorVM.showDialog();
    }
    //really, the destroy method is redundant here because the application lives while the page lives
    destroy() {
        if (this._isDestroyed)
            return;
        this._isDestroyCalled = true;
        var self = this;
        try {
            self._errorVM.destroy();
            self._headerVM.destroy();
            self._productVM.destroy();
            self._uploadVM.destroy();
            self._dbContext.destroy();
            if (!!self._sseVM)
                self._sseVM.destroy();
        } finally {
            super.destroy();
        }
    }
    get options() { return <IMainOptions>this._options; }
    get dbContext() { return this._dbContext; }
    get errorVM() { return this._errorVM; }
    get headerVM() { return this._headerVM; }
    get productVM() { return this._productVM; }
    get uploadVM() { return this._uploadVM; }
    //server side events
    get sseVM() { return this._sseVM; }
    get sseMessage() { return this._sseMessage; }
}
```

An application's instance is usually created in the *global.onload* handler (*which has the semantics of JQuery's ready method*) and in there the application is started by calling the application's *startUp* method.

```
RIAPP.global.addOnLoad(function (sender, a) {
        var global = sender;
        //initialize images folder path
        global.defaults.imagesPath = mainOptions.images_path;
        //create and then start application
        var thisApp = new DemoApplication(mainOptions);

        thisApp.startUp((app) => {
        });
});
```

Before calling the *startUp* method you can register template groups, or start loading data templates from the server.

To handle errors you can subscribe to the application's *'error'* event.
This event is triggered when some object's instance inside the application catches an error and executes the *handleError* method (*usually, databindings and user defined view models do this*). If an error is not handled in the application's error hander, the error is passed on to the global object, where it can be handled in the global's error event handler.

The Application class implements IApplication interface:

```
export interface IApplication extends IErrorHandler, IDisposable {
    _getInternal(): IInternalAppMethods;
    addOnStartUp(fn: TEventHandler<IApplication, any>, nmspace?: string, context?: BaseObject): void;
    removeOnStartUp(nmspace?: string): void;
    registerElView(name: string, vw_type: elviewMOD.IViewType): void;
    registerConverter(name: string, obj: RIAPP.IConverter): void;
    getConverter(name: string): IConverter;
    registerType(name: string, obj: any): void;
    getType<T>(name: string): T;
    registerObject(name: string, obj: any): void;
    getObject<T>(name: string): T;
    loadTemplates(url: string): IPromise<any>;
    loadTemplatesAsync(fn_loader: () => IPromise<string>): IPromise<any>;
    registerTemplateLoader(name: string, fn_loader: () => IPromise<string>): void;
    getTemplateLoader(name: string): () => IPromise<string>;
    registerTemplateGroup(name: string, group: {
        fn_loader?: () => IPromise<string>;
        url?: string;
        names: string[];
    }): void;
    bind(opts: bindMOD.IBindingOptions): bindMOD.Binding;
    uniqueID: string;
    appName: string;
    appRoot: Document | HTMLElement;
    contentFactory: contentMOD.IContentFactory;
}
```

Application's methods:

Method name	Description
registerElView	Registers element views in the application type system
registerType	Registers an object type (*class*) by its name. The type can be later retrieved by *getType* method.
getType	Returns the registered type by its name.
registerObject	Almost the same as the *registerType* method, only this method is used to register object's instances instead of the types (*classes*). The object is automatically unregistered when it is destroyed.

getObject	Returns the registered object by its name.
registerConverter	Registers a converter by its name.
getConverter	Returns the registered converter by its name.
startUp	Starts an application's instance. It accepts a callback function which is executed on the application startUp.
registerTemplateLoader	Registers function which loads an individual template asynchronously (*on as needed basis*) - returns a promise which resolves with a template as a string containing HTML. P.S.- *See the DataGrid demo, for an example how it is used*.
loadTemplates	The same as the global's *loadTemplates*, only it loads templates into the application's scope. They are available only to the application which it loads. This method must be used before the application's *startUp* method is invoked.
registerTemplateGroup	Registers a group of templates to be loaded on as needed basis from the server. Accepts a group's name and options for the group. Each template's group can contain one or several templates. P.S. - *templates names must be unique between different groups. See the Single Page Application demo for an example.*

Applications's properties:

Property	Read Only	Description
options	Yes	Exposes application's options.
appRoot	Yes	Exposes the root (*an element or a window.document*) of the application.
appName	Yes	the application's unique name. If it is not provided on creation with the options, it will have a default value '*default*'.
global	Yes	exposes an instance of the *RIAPP.Global* object for easier access to the global object.
contentFactory	Yes	exposes a content factory which is used by the application.
UC	---	a namespace (*empty object instance*) for attaching custom user code.
VM	---	a namespace (*empty object instance*) for attaching user defined view models. (*You can attach view model's instances to this namespace. But it is better to expose them as properties of a derived application class*)
app	Yes	Returns a *self* reference. It can be used to assign an application's instance as a source for databindings. It can be helpful in some cases, because the databinding's source expression (*if we use a fixed source*) is evaluated starting from the application's instance and we can not leave it empty in that case. {this.dataContext,to=viewModel,source=app} This is the same as: {this.dataContext,source=viewModel} We can not use empty source like this (*invalid usage*) ~~{this.dataContext,to=viewModel,source=}~~

		If we don't use the source at all, like in the next expression: {this.dataContext,to=viewModel} Then the source is not fixed and is defined by the current data context and is changed when the data context is changed.

Usually any real world application uses user defined modules. Their names and the init functions of the user modules are provided with the application's options. For example, the DEMO application (*GridDemo example*) uses 3 user modules - COMMON, HEADER, GRIDDEMO, SSEVENTS.

```
//properties must be initialized on HTML page
   export var mainOptions: IMainOptions = {
      service_url: null,
      permissionInfo: null,
      images_path: null,
      upload_thumb_url: null,
      templates_url: null,
      productEditTemplate_url: null,
      sizeDisplayTemplate_url: null,
      modelData: null,
      categoryData: null,
      sse_url: null,
      sse_clientID: null,
      user_modules: [{ name: "COMMON", initFn: COMMON.initModule },
         { name: "HEADER", initFn: HEADER.initModule },
         { name: "GRIDDEMO", initFn: initModule },
         { name: "SSEVENTS", initFn: SSEVENTS.initModule }]
};
```

You can define a function (*conventionally named initModule*) in the user module provided to the application (*in the options*) , which accepts a parameter and returns the current module, like this:

```
 function initModule(app: Application) {
     return GRIDDEMO;
 };
```

This function is invoked when the application initializes its modules. You can register converters, object instances and etc in this function, like this:

```
 export function initModule(app: Application) {
           app.registerConverter('listTypeConverter', new ListTypeConverter());
           app.registerConverter('channelConverter', new ChannelConverter());
           app.registerConverter('errorColorConverter', new ErrorColorConverter());
           return MAIL;
 };
```

You can omit the reference to a module in the application's options (*as it was shown above*) if you don't need the *initModule* function for initialization. Some modules don't need to register anything with the application.

For example, the GRIDDEMO module (*shown above*) does not register anything. So you can omit referencing it in the options, and you can also remove from the module the *initModule* function.

This part of code in that case will look like this:

```
//user modules used by this application
 user_modules: [{ name: "COMMON", initFn: COMMON.initModule },
   { name: "HEADER", initFn: HEADER.initModule }
 ]
```

Also, it is not necessary to return from the *initModule* function a module's reference. This function can return any value (*even a null*). This value will be stored in the application's module property value (*it is a map of loaded modules by their names*).
You can use it like this:

```
var someModule = app.module['someModuleName'];
```

Although, it is not needed in typescript version of the framework.

2.4 *Binding class*

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

Binding's properties:

Property	Description
targetPath	a path for a property which is updated when the source property's value changes
sourcePath	a path for a property which provides a value to the target property

mode	mode of the binding *RIAPP.MOD.binding.BINDING_MODE*. It is defined as *export enum BINDING_MODE {
	OneTime = 0,
	OneWay = 1,
	TwoWay = 2,
	BackWay=3
	}*
	P.S. - *BackWay* is the mode that works just as *OneWay* mode only it works in opposite direction - the data flow is from the target to the source.

source	the source of the data (*must be a descendant of the BaseObject*)
target	the target of the data (*must be a descendant of the BaseObject*)
converter	converts the data when it flows between the source and the target (*for example, an object value to a string value and backward*)
converterParam	the converter can use a parameter to adjust data conversion (*for example, a formatting style*)
isSourceFixed	Returns true if we set the source in the databinding expression explicitly.
isDisabled	used to turn off the databinding when it is not needed, to conserve resources.

The target of the data binding can be explicitly set when an instance of the *Binding* class is created in the code. The application's class has a helper method *bind* which creates and returns an instance of the *Binding*.

An example of databinding objects' properties in code (typescript code):

```
appInstance.bind({ sourcePath: 'selectedSendListID', targetPath: 'sendListID',
   source: this._sendListVM, mode:  RIAPP.MOD.binding.BINDING_MODE.OneWay,
   target: this._uploadVM, converter: null, converterParam: null
});
```

At the time when the data bindings are created by the application from the data binding expressions (*which are defined declaratively*), the application evaluates all the paths used in the expression to resolve the real object instances, then it creates instances of the Binding class.

When a declarative binding expression is parsed, a HTML DOM element is wrapped with a class derived from the *BaseElView* class (*which is defined in baseElView module*) to expose properties which can be databound. Element views serve the purpose of real data binding targets (*in place of raw DOM elements*). The selection of which descendant of the BaseElView class to create is determined by a HTML element tag or it can be determined by specifying a name of the element view in a custom *data-view* attribute.

Note: *You can look at the element view class to see which properties it exposes. The exposed properties can be data bound.*

For example, you can specify to create a custom *Expander* element view for a span tag (*instead of the default element view for the span tag*) by specifying a registered name of the element view:

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

You can also provide some optional parameters to the element view instance by using the options in the *data-view* attribute value, as in the example:

```
<table data-bind-1="{this.dataSource,to=dbSet,source=productVM}"
data-bind-2="{this.grid,to=grid,mode=BackWay,source=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}}">
```

Two other examples of using the data-view options:

```
<select size="1" data-bind-1="{this.dataSource,to=filter.ProductModels}"
data-bind-2="{this.selectedValue,to=filter.modelID,mode=TwoWay}"
data-bind-3="{this.selectedItem,to=filter.selectedModel,mode=TwoWay}"
data-bind-4="{this.toolTip,to=filter.selectedModel.Name}"
data-view="options:{valuePath=ProductModelID,textPath=Name}"></select>


<input type="text" id="saleStart1" placeholder="Enter Date"
data-bind="{this.value,to=filter.saleStart1,mode=TwoWay,converter=dateConverter}"
data-view="name:datepicker,options={datepicker:{ showOn:button,yearRange:'-15:c',changeMonth: true,
changeYear: true }}"/>
```

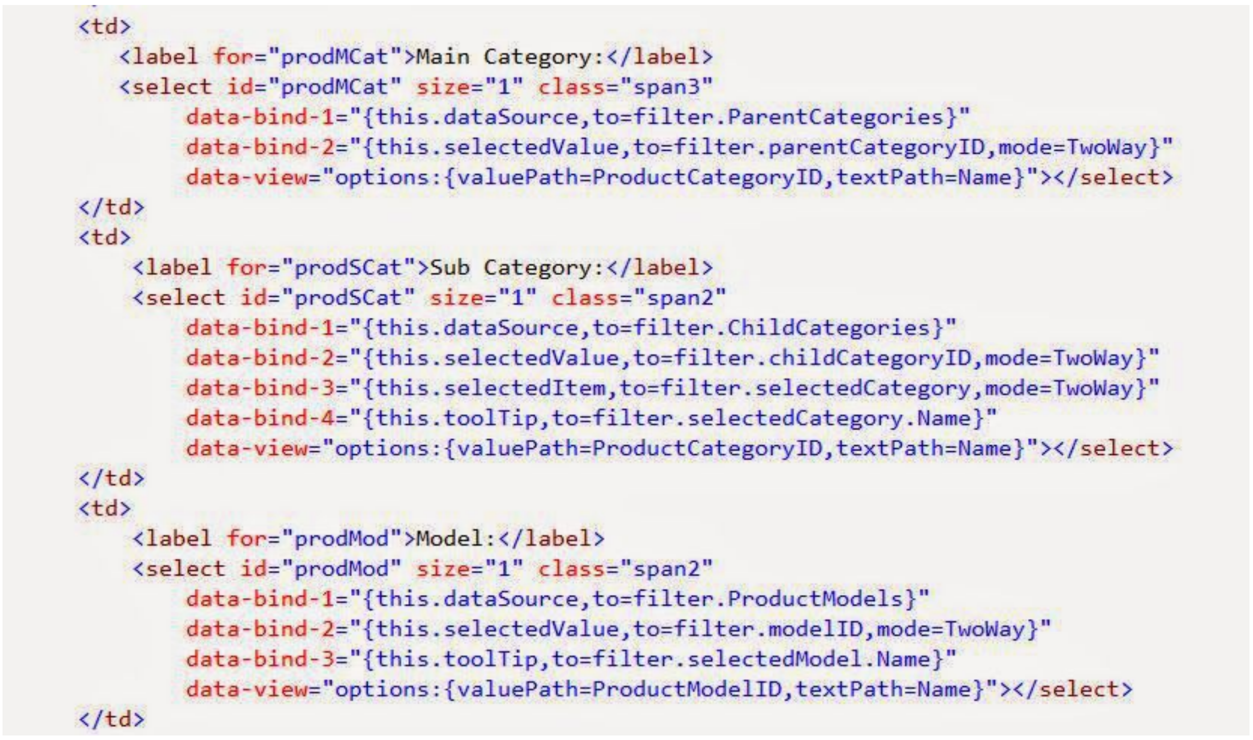
The data binding expressions are contained inside a custom data-bind attribute's value. The data-bind attribute's value can contain multiple data binding expressions. Each expression is enclosed in curly braces {} (*you can omit them if you have only one expression, but it is not recommended*). The target of the data binding in this expression is always the element view which is created when the framework's application code parses this expression.

Warning: The data binding targets must always be the properties exposed by the element view (*a wrapper of the HTML DOM element*) and not directly to the HTML DOM element's properties which the element view wraps in itself.

The new version of the framework (*2.2.0 and above*) supports putting each individual data binding expression into its own data-bind attribute. For example you can write the above data bindings for a table as:

```
<table data-bind-1="{this.dataSource,to=dbSet,source=productVM}"
data-bind-2="{this.propChangedCommand,to=propChangeCommand,source=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}}">
```

A new way of putting databinding expressions in its own data-bind attribute:



```
<td>
   <label for="prodMCat">Main Category:</label>
   <select id="prodMCat" size="1" class="span3"
        data-bind-1="{this.dataSource,to=filter.ParentCategories}"
        data-bind-2="{this.selectedValue,to=filter.parentCategoryID,mode=TwoWay}"
        data-view="options:{valuePath=ProductCategoryID,textPath=Name}"></select>
</td>
<td>
    <label for="prodSCat">Sub Category:</label>
    <select id="prodSCat" size="1" class="span2"
        data-bind-1="{this.dataSource,to=filter.ChildCategories}"
        data-bind-2="{this.selectedValue,to=filter.childCategoryID,mode=TwoWay}"
        data-bind-3="{this.selectedItem,to=filter.selectedCategory,mode=TwoWay}"
        data-bind-4="{this.toolTip,to=filter.selectedCategory.Name}"
        data-view="options:{valuePath=ProductCategoryID,textPath=Name}"></select>
</td>
<td>
    <label for="prodMod">Model:</label>
    <select id="prodMod" size="1" class="span2"
        data-bind-1="{this.dataSource,to=filter.ProductModels}"
        data-bind-2="{this.selectedValue,to=filter.modelID,mode=TwoWay}"
        data-bind-3="{this.toolTip,to=filter.selectedModel.Name}"
        data-view="options:{valuePath=ProductModelID,textPath=Name}"></select>
</td>
```

The expression: `{this.dataSource,to=mailDocsVM.dbSet,mode=OneWay,source=sendListVM}`

instructs to bind the *dataSource* property on the current element view to the property path *mailDocsVM.dbSet* on the source (*an instance of the SendListVM view model in this case, which is exposed through the application's property)* in *OneWay* mode (*which is the default value, and can be omitted here*).

If the databindings are NOT inside data templates or data forms and **without explicitly providing the source**, then the source will be the **application's instance** (*by default*), but if they **are** inside data templates, the implicit source is the current **data context** (*data templates as well as data forms have data contexts*).

When the *source* is explicitly provided in the data binding expression, the data binding path is always evaluated starting from the application's instance. The above expanded expression path can be represented in pseudocode as:
[Current Application's instance].sendListVM.mailDocsVM.dbSet.

Very often you can omit the source attribute in a data binding's expression (*using an implicit source, not a fixed one*), and can write previous binding expression as:

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

But then you should pay attention to where the data binding is used! If you use this data binding expression inside a data template, then the path evaluation will start from the data context object which is assigned to the data template (*data templates have a dataContext property*), and the data context's value can change when the program is running. (*the same is true for the dataform's data context*)

The data 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 a null value (*effectively changing the data binding implicit source property value*).
Otherwise, if you explicitly provide the source attribute in a data binding expression, then, even if it was used inside a data template, the source will be fixed. (*and the path's evaluation in that case always starts from the application's instance- the application is the implicit data context*).

The above examples were a shortcut style of the data binding expression, but you can write a data binding expression in another (*expanded, and rarely used, but correct*) way:

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

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

You can use **:** separator instead of **=** separator (*which separates a name and a value*) in the databinding expressions, such as:

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

It is just a matter of personal preference which separator to use, **=** or **:**.

Instances of databindings are created not only on the startup of an application, they can also be created when the application is running. It can happen when UI controls (*element views*) on the page create data templates during their life cycle. Data templates can include data binding expressions, and they are evaluated at the time when instances of the data templates are created.
For example, when a *DataGrid* control instance is databound to a datasource (*or the dataSource is refreshed*), the datagrid creates cells for each grid's row. The grid's *DataCell* can have a templated data content (*cells in which content is defined by data templates*), and when the template instances are created then the databindings in the elements are evaluated. Later on, when the template instances are destroyed (*when a row in the DataGrid is removed*), instances of the databindings are also destroyed with the instance of the template.

The Converters

Databindings can use *converters* to convert values from the source to the target and vice-versa.

an example of a converter definition (typescript code):

```
export class UppercaseConverter extends MOD.converter.BaseConverter {
            convertToSource(val, param, dataContext) {
                if (utils.check.isString(val))
                    return val.toLowerCase();
                else
                    return val;
            }
            convertToTarget(val, param, dataContext) {
                if (utils.check.isString(val))
                    return val.toUpperCase();
                else
                    return val;
            }
 }
```

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=demoVM}" />
```

A special case is if a method of the converter (*any of it*) returns an *undefined* value. In this case, the data binding ignores the value returned by the converter and does not updates the source or the target,

an example of a converter which returns an undefined value

```
export class ListTypeConverter extends MOD.converter.BaseConverter {
            convertToSource(val, param, dataContext) {
                return !!val ? param : undefined;
            }
            convertToTarget(val, param, dataContext) {
                return (val == param) ? true : false;
            }
}
```

In the above example, the converter returns an *undefined* value when the value from the target is *false*. This prevents it from updating the property value on the source in this case (*see the collections demo, only one radio button updates the source - the one which is checked, although they are databound to the single source property*).

Debugging databindings

The RIAPP module has a globally available variable *DebugLevel* which has a *DEBUG_LEVEL* type defined as:

```
export enum DEBUG_LEVEL {
      NONE= 0, NORMAL= 1, HIGH= 2
}
```

When you want to perform a testing how your application works it is recommended to set the *DebugLevel* variable to a level above *NONE*. When the *DebugLevel* is *NORMAL* than the global's *addOnUnResolvedBinding* event handler is triggered when the databinding's path can not be resolved.

an example of testing for unresolved databindings' paths:

```
 RIAPP.global.addOnUnResolvedBinding((s, args) => {
        var msg = "unresolved databound property for";
        if (args.bindTo == RIAPP.BindTo.Source) {
           msg += " Source: "
        }
        else {
           msg += " Target: "
        }
        msg += "'" + args.root + "'";
        msg += ", property: '" + args.propName + "'";
        msg += ", binding path: '" + args.path + "'";

 //here can be a custom logger, or any notification mechanism
        console.log(msg);
  });
```

When the *DebugLevel* is *HIGH* then if the path is unresolved, a javascript debugger kicks in. In the *HIGH* debug level mode there are also performed checks on whether a property name exists on the object when the application executes *raisePropertyChanged, addOnPropertyChange, removeOnPropertyChange* methods.

2.5 *The Command class*

The Command provide the means for a declarative execution of methods defined on view models. Element views can expose properties which accept commands' implementations in view models (*for example, a button's element view, for the click scenario*).

an example of binding a custom command to a button's command property:

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

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

an example of a command's implementation (typescript code):

```
this._uploadCommand = new MOD.mvvm.Command(function (sender, param) {
                try {
                   self.uploadFiles(self._fileEl.files);
                } catch (ex) {
                   self.handleError(ex, this);
                }
             }, self, function (sender, param) {
                return self._canUpload();
 });
```

The first parameter of the command's constructor is a callback function (*the action*), which is invoked when a command is triggered by the UI element (*in this case when the button is clicked*).
The second parameter is an object which defines a *this* context for the command's action (*inside the action method this will be a this property value*).
The third parameter is a callback function which returns a boolean result. It determines if the command is currently in an enabled or disabled state.

If we want to trigger reevaluation of the condition when the command is disabled or enabled, then we call the command's *raiseCanExecuteChanged* method, as in the example:

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

The command action function accepts two parameters – the first is the sender object, which is the invoker of the command (*typically, an element view's instance like button*), the second argument is a parameter which can be explicitly provided in the data binding expression.

an example of a HTML markup inside the data template's definition:

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

Note: *{this.commandParam} expression binds commandParam property on the element view to the current data context .*

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

```
this._dialogCommand = new MOD.mvvm.Command(function (sender, param) {
                try {
                    //using command parameter to provide the product item
                    self._product = param;
                    self.id = self._product.ProductID;
                    self._dialogVM.showDialog('uploadDialog', self);
                } catch (ex) {
                    self.handleError(ex, this);
                }
            }, self, function (sender, param) {
                return true;
});
```

2.6 *Element views*

The Element view is a wrapper around a HTML DOM element's and can also wrap other controls (*for example, JQuery UI pluggins*) which you wish to use in a declarative way. It exposes properties which can be databound. The element views help you to use data bindings declaratively. They are created when the data binding's expressions are parsed.

Note: *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 class. The Element views are objects which are all derived from the BaseObject, so they can expose properties which can be databound.*

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

For example, the StackPanelElView uses options to determine how it should be displayed - horizontally or vertically. The *TextBoxElView* uses an *updateOnKeyUp* option, to decide when to update the databinding's source – when the textbox loses a focus (*the default value*) or when a *keyup* event occurs.

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

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

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

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

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

Registration of element views in the baseElView.ts file:

```
global.registerElView('busy_indicator', BusyElView);
global.registerElView('input:checkbox', CheckBoxElView);
global.registerElView('threeState', CheckBoxThreeStateElView);
global.registerElView('input:text', TextBoxElView);
global.registerElView('input:hidden', HiddenElView);
global.registerElView('textarea', TextAreaElView);
global.registerElView('input:radio', RadioElView);
global.registerElView('input:button', ButtonElView);
global.registerElView('input:submit', ButtonElView);
global.registerElView('button', ButtonElView);
global.registerElView('a', AnchorElView);
global.registerElView('abutton', AnchorElView);
global.registerElView('expander', ExpanderElView);
global.registerElView('span', SpanElView);
global.registerElView('div', BlockElView);
global.registerElView('section', BlockElView);
global.registerElView('block', BlockElView);
global.registerElView('img', ImgElView);
```

An element view can be simple, exposing only several properties of the DOM element (*like the TextBoxElView*) and can also be complex (*like the GridElView*) encapsulating a complex UI control.

Complex element views are usually defined in a module where the control is defined. For example, the *GridElView* is defined in the grid.ts module.

BaseElView – base element view type, which provides support for the display of validation errors, and also provides several properties for all descendants of this type. The *BaseElView* and all its descendants can accept a *tip* and a *css* options. The first option sets a *tooltip* to the wrapped HTML DOM element, and the second option 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	An application instance, which created this element's view
$el	Yes	A jQuery wrapper of the DOM element
el	Yes	The wrapped HTML DOM element
uniqueID	Yes	A unique id which can be used as a namespace, for event's subscription inside element's view code (*in the constructor and the methods*)
isVisible	No	Determines the DOM element visibility on the page
propChangedCommand	No	A command (*usually, defined in a view model*) for property change notification. *For example, a GridElView invokes this command when the element view's grid property changes. So, the view model can obtain an instance of the element view through this notification mechanism.*
toolTip	No	A valid HTML string which can be provided for display over element view's DOM element.
css	No	A css class which can be provided for the element view's DOM element. *It is useful to change display style of the element based on the data bound data.*
validationErrors	No	Validation errors are internally used by the framework. *Binding class sets this property if a validation error occured, for the error display.*

InputElView – a descendant of the *BaseElView* class. It is not used directly, but is used as a base class for several element views (*TextBoxElView, CheckBoxElView, RadioElView*). It adds a property *isEnabled* to all of its descendants and a *value* property.

TextBoxElView – a wrapper around a <input type="text"/> element.
Besides inherited properties, it exposes a *value* property, which exposes the 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 changed by using the *updateOnKeyUp* option, so the value is changed on each *keyup* event.

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

TextAreaElView – a wrapper around a <textarea /> element. It is a descendant of the *BaseElView*. It has *isEnabled, value* (*to get or set text*), *rows, cols, wrap* properties.

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

CheckBoxElView – a wrapper around a <input type="checkbox"/> element.
It exposes *checked* property of the HTML DOM element.

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

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

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

CommandElView - a descendant of the *BaseElView*. It is not used directly, but is used as a base class for several element views (*like ButtonElView, AnchorElView*). It adds two properties *command* and *commandParam* to all of its descendants. The descendants use an internal *invokeCommand* method to trigger the command's action (*typically, when a button or a link is clicked*). It also exposes an *isEnabled* property.

ButtonElView - a descendant of the *CommandElView*. It is a wrapper around a <button/> or a <input type="button" /> DOM element. It exposes *value, text, html* properties of the button element. It also exposes a boolean *preventDefault* property. It can be used to choose if the button should trigger its default action or not. This element view's command property can be data bound to a command implementation, so to trigger an action when the button is clicked.

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

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

AnchorElView - a descendant of the *CommandElView*. It is a wrapper around an <a/> element. The link can display a text or an image (*which can be determined by the options*). It exposes *imageSrc, html, text, href, preventDefault* properties of the DOM element. The Anchor DOM element behaves similar to the button element, but has a different default action.

```
<a class="btn btn-info btn-small" data-bind="{this.command,to=loadCommand}"><i class="icon-
search"></i>&nbsp;Filter</a>
```

ExpanderElView - a descendant of the *AnchorElView*. It adds a default image to the anchor element, which switches its appearance depending on the expanded or collapsed state. When the image is clicked it triggers the command (*if it is databound*) to invoke an action on the view model. The element view is registered by the name '*expander*'.

```
<a href="#" data-bind="{this.command,to=expanderCommand,source=headerVM}"
```

```
data-view="name=expander"></a>

//an example of the definition of the command on the view model
this._expanderCommand = new MOD.mvvm.Command(function (sender, param) {
      if (sender.isExpanded) {
            self.expand();
      }
      else
            self.collapse();
}, self, null);
```

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

an example of a data template which uses theTemplateElView (*see the DataGrid demo*):

```
<!--upload thumbnail dialog template-->
<div id="uploadTemplate" style="margin:5px;" data-role="template"
data-bind="{this.command,to=templateCommand,source=uploadVM}"
data-view="name=template">
    <!--a dummy form action to satisfy the 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.html,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 a command's definition databound to the TemplateElView's command property (*see UploadThumbnailVM in gridDemo.ts*):

```
//executed when template is loaded or unloading
this._templateCommand = new MOD.template.TemplateCommand(function (sender, param) {
                  try {
                      var template = param.template, $ = global.$,
                       fileEl = $('input[data-name="files-to-upload"]', template.el);

                      if (fileEl.length == 0)
                          return;
```

```
                    if (param.isLoaded) {
                        fileEl.change(function (e) {
                            $('input[data-name="files-input"]', template.el).val($(this).val());
                        });
                        $('*[data-name="btn-input"]', template.el).click(function (e) {
                            e.preventDefault();
                            e.stopPropagation();
                            fileEl.click();
                        });
                    }
                    else {
                        fileEl.off('change');
                        $('*[data-name="btn-input"]', template.el).off('click');
                    }
                } catch (ex) {
                    self.handleError(ex, this);
                }
            }, self, function (sender, param) {
                return true;
 });
```

SpanElView - a wrapper around the <span/> element. It is used to databind a string property to the content inside the span DOM element. It exposes *value, text, html, color, fontSize* properties which can be data bound. The *value* property is semantically equivalent to the *text* property.

Warning: *Data binding to the html property should be used carefully, because it inserts a HTML content inside the 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,source=testObject1}"></span>
<span data-bind="{this.text,to=testProperty2,source=testObject1}"></span>
<span data-bind="{this.html,to=testProperty3,source=testObject1}"></span>
```

BlockElView - a descendant of the *SpanElView*. It is a wrapper around the <div/> element or some other block element like a <section/>. In addition to the properties inherited from the *SpanElView*, it adds *borderColor, borderStyle, width, height* properties, which can be data bound. It is also registered by the name *block*, which can be provided in a data-view attribute. But for the <section/> and the <div/> elements it is not needed (*because it is the default element view for them*).

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

ImgElView - a wrapper around the <img/> element. It exposes the DOM image's *src* property.

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

BusyElView - a wrapper around any block HTML DOM element (*typically, a div element*). It exposes the *isBusy* and the *delay* property.
It is used to display an animated loader gif image above the content of the HTML DOM element to which it is attached. The element view is registered by the name *busy_indicator*.

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

GridElView - a wrapper around the <table/> element. It is used to attach the logic and the markup of the *DataGrid* control to the HTML DOM element. It exposes the *dataSource* and the *grid* property. It is used to display the datagrid with the data obtained through the data source.

Note: *the grid property is read only and exposes the encapsulated DataGrid control.*

an example of the markup for the DataGrid (*see the DataGrid demo*):

```
<table data-name="gridProducts" data-bind="{this.dataSource,to=dbSet,source=productVM}
            {this.grid,to=grid,mode=BackWay,source=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>
           <th data-column="width:50px,type:row_actions"></th>
           <th data-column="width:40px,type:row_selector,rowCellCss:selectorCell,colCellCss:selectorCol"></th>
           <th data-column="width:100px,sortable:true,title:ProductNumber"
             data-content="fieldName:ProductNumber,css:{displayCss:'number-display',editCss:'number-edit'},readOnly:true">
             </th>
           <th data-column="width:25%,sortable:true,title:Name" data-content="fieldName:Name,readOnly:true"></th>
           <th data-column="width:90px,title:'Weight',sortable:true" data-content="fieldName:Weight,readOnly:true"></th>
           <th data-column="width:15%,title=CategoryID,sortable:true,sortMemberName=ProductCategoryID"
             data-content= "fieldName=ProductCategoryID,name:lookup,
options:{dataSource=dbContext.dbSets.ProductCategory,valuePath=ProductCategoryID,textPath=Name},readOnly:true">
             </th>
           <th data-column="width:100px,sortable:true,title='SellStartDate'"
             data-content="fieldName=SellStartDate,readOnly:true">
             </th>
           <th data-column="width:100px,sortable:true,title='SellEndDate'"
             data-content="fieldName=SellEndDate,readOnly:true">
             </th>
           <th data-column="width:90px,sortable:true,title='IsActive'" data-content="fieldName=IsActive,readOnly:true"></th>
           <th data-column="width:10%,title=Size,sortable:true,sortMemberName=Size"
             data-content="template={displayID=sizeDisplayTemplate}">
             </th>
       </tr>
       </thead>
       <tbody></tbody>
</table>
```

It is defined in the *datagrid.ts* file.

PagerElView - a wrapper around a block HTML DOM element (*typically, a <div/>*). It is used to attach the logic and the markup of the *Pager* control to the HTML DOM element. It exposes the *dataSource* and the *pager* properties. The element view is registered by the name *pager* and it is defined in the *pager.ts* file.

an example of the markup for the Pager (*see the DataGrid demo*):

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

StackPanelElView - a wrapper around a block HTML DOM element (*typically, a <div/>*). It is used to attach the logic and the markup of the *StackPanel* control to the block element. It exposes a *dataSource* and a *panel* properties. It is used to display horizontally or vertically stacked panels (*which use data templates*) on the page. The element view is registered by the name *stackpanel* and it is defined in the *stackpanel.ts* file.

an example of the markup for the StackPanel (*see a CollectionsDemo in the demo*):

```
<div style="border: 1px solid gray;float:left;width:150px; min-height:65px; max-height:250px; overflow:auto;"
data-bind="{this.dataSource,to=historyList,source=demoVM}"
data-view="name=stackpanel,options:{templateID:stackPanelItemTemplateV,orientation:vertical}"></div>
```

SelectElView - a wrapper around the <select/> element.
It is used to attach the logic of the *ListBox* control to the HTML DOM element.
It exposes *isEnabled, dataSource, selectedValue, selectedItem*, and *listBox* properties. The data source of the element view provides the data to fill the list items (*options*).

an example of the markup for the select elements:

```
<select id="prodMCat" size="1" class="span3"
data-bind="{this.dataSource,to=filter.ParentCategories}
{this.selectedValue,to=filter.parentCategoryID,mode=TwoWay}"
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 - a wrapper around the <div/> or the <form/> element. It attaches the logic of the *DataForm* control to them for managing the data context (*see more info in the DataForm control section of this guide*). It exposes the *dataContext* and the *form* properties. It is used to display the data for editing and viewing purposes. The element view is registered by the name *dataform* and it is defined in the *dataform.ts* file.

an example of the markup for the dataform:

```
<div style="width: 100%; margin:0px;" 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'},name:multyline,options:{rows:3,cols:20,wrap:hard}">
                    </span>
                </td>
            </tr>
            <tr>
```

```
                <td>
                    ProductNumber:
                </td>
                <td>
                    <span data-content="fieldName:ProductNumber"></span>
                </td>
            </tr>
            <tr>
                <td>
                    Color:
                </td>
                <td>
                    <span data-content="fieldName:Color"></span>
                </td>
            </tr>
            <tr>
                <td>
                    Cost:
                </td>
                <td>
                    <span data-content="fieldName:StandardCost"></span>
                </td>
            </tr>
            <tr>
                <td>
                    Price:
                </td>
                <td>
                    <span data-content="fieldName:ListPrice,readOnly:true"></span>
                </td>
            </tr>
            <tr>
                <td>
                    Size:
                </td>
                <td>
                    <span data-content="fieldName:Size"></span>
                </td>
            </tr>
            <tr>
                <td>
                    Weight:
                </td>
                <td>
                    <span data-content="fieldName:Weight"></span>
                </td>
            </tr>
            <tr>
                <td>
                    Category:
                </td>
                <td>
                    <span data-content="fieldName=ProductCategoryID,name:lookup,
options:{dataSource=dbContext.dbSets.ProductCategory,valuePath=ProductCategoryID,textPath=Name},
css:{editCss:'listbox-edit'}">
                    </span>
                </td>
            </tr>
            <tr>
                <td>
                    Model:
                </td>
                <td>
                    <span data-content="fieldName=ProductModelID,name:lookup,
options:{dataSource=dbContext.dbSets.ProductModel,valuePath=ProductModelID,textPath=Name},
css:{editCss:'listbox-edit'}">
                    </span>
                </td>
            </tr>
            <tr>
                <td>
                    SellStartDate:
                </td>
                <td>
                    <span data-content="fieldName:SellStartDate"></span>
                </td>
            </tr>
            <tr>
                <td>
                    SellEndDate:
                </td>
                <td>
                    <span data-content="fieldName:SellEndDate"></span>
                </td>
            </tr>
            <tr>
                <td>
                    DiscontinuedDate:
                </td>
                <td>
                    <span data-content="fieldName:DiscontinuedDate"></span>
                </td>
            </tr>
            <tr>
                <td>
                    rowguid:
                </td>
                <td>
                    <span data-content="fieldName:rowguid"></span>
                </td>
            </tr>
            <tr>
                <td>
                    When Modified:
                </td>
                <td>
                    <span data-content="fieldName=ModifiedDate"></span>
                </td>
            </tr>
            <tr>
                <td>
                    IsActive:
                </td>
                <td>
                    <span data-content="fieldName=IsActive"></span>
                </td>
            </tr>
        </tbody>
      </table>
</div>
```

DatePickerElView - a wrapper around the <input type='text'/> element.
It is used to attach the logic of the *datepicker* control to the HTML DOM element. It is registered by the name *datepicker* and it is defined in the *datepicker.ts* file.

an example of the usage of the DatePickerElView:

```
<input type="text" placeholder="Enter Date"
data-bind="{this.value,to=filter.saleStart1,mode=TwoWay,converter=dateConverter}"
data-view="name:datepicker,options={datepicker:{ showOn:button,yearRange:'-15:c',changeMonth: true,changeYear:
true }}"/>
```

TabsElView - a wrapper around the <div/> element. It attaches a JQuery UI Tabs plugin logic to the HTML DOM element for managing the content displayed in the tabs. It exposes *tabsEvents* property. The element view is registered by the name *tabs* and it is defined in the *tabs.ts* file.

Note: *The tabsEvents is used to get notifications on tabs events (like when a tab was selected) and handle them in the view model.*

an example of the HTML markup for the tabs:

```
<div id="productDetailsTemplate" style="width: 100%; margin: 0px;" data-role="template">
      <div data-name="tabs" style="margin: 5px; padding: 5px; width: 95%;"
      data-bind="{this.tabsEvents,to=tabsEvents,source=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=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=uploadVM}{this.commandParam}"
data-view="options={tip='click me to upload product thumbnail photo'}">Upload product thumbnail</a>
               </div>
           </div>
       </div>
       <!--myTabs-->
    </div>
</div>
```

an example of handling tabs' events in the view model:

```
 //#begin MOD.tabs.ITabsEvents
           addTabs(tabs: MOD.tabs.ITabs): void {
               console.log('tabs created');
           }
           removeTabs(): void {
               console.log('tabs destroyed');
           }
           onTabSelected(tabs: MOD.tabs.ITabs): void {
               console.log('tab selected: '+ tabs.tabIndex);
           }
```

```
//#end MOD.tabs.ITabsEvents
```

DynaContentElView - a wrapper around a block HTML DOM element (*typically, a <div/>*). It exposes the *templateID* and the *dataContext* properties.
It is used to mark a block element as a content region which will contain a template content. The data templates (*used for display in this region*) can be switched at runtime. When templates are switching then the current template unloads, and is replaced with a new one.
The template switching is triggered when the *templateID* (*the currently displayed template*) property value is changed. The *dataContext* property is used to provide a data context to the currently displayed template.
It can apply animations between views transitioning if its *animation* property will be databound to a property on view model which exposes an instance of an *Animation* object.
The element view is registered by the name *dynacontent* and it is defined in the *dynacontent.ts* file.

an example of the markup for the dynacontent:

```
<div id="demoDynaContent"
data-bind="{this.templateID,to=viewName,source=customerVM.uiMainView}
{this.dataContext,source=customerVM}{this.animation,to=animation,source=customerVM.uiMainView}"
data-view="name=dynacontent"></div>
```

Note: *Look at the SPA Demo (Single Page Application) for an example how it is used.*

Custom built element views - In addition to the core element views, it is easy to add a custom element view.
For example, in the demo application there have been added several custom element views, such as *AutoCompleteElView, DownloadLinkElView, FileImageElView* – they all have been defined in user modules.

```
<!--using a custom element view for the display of a 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" })'}"/>
```

2.7 *Data templates*

The Data templates are pieces of the HTML markup which can be used by UI controls for cloning their structure and displaying them on the page. A data template’s definition (*HTML markup*) must have an *id* attribute for referencing it in the options of the UI controls.
They are used in the framework by creating instances of a *Template* class. This class is defined in the *template.ts* file. Internally it retrieves a template’s definition (*HTML string*) by its *id*, and creates the DOM structure from it. It also processes databindings inside the template, and sets the source on them (*a dataContext property value of the template*).
The data template’s *dataContext* property can be set or reset at any time on the data template’s instance.

The framework contains several built-in controls, which can use the data templates: *DataEditDialog, DataGrid, StackPanel, DynaContentElView*.

the example shows programmatic creation of the template's instance:

```
 _createTemplate() {
                  return new templMOD.Template({
                     app: this.app,
                     templateID: this._templateID,
                     dataContext: null,
                     templEvents: this
                  });
}
```

The data templates can be defined in four ways (*by their loading method*):

1) Define them on the page in a special section for the templates
2) Preload them all from the server in a single file per page (*at the start of the application*)
3) Make them loadable on as needed basis (*register a loader function*)
4) Register a group of templates for loading them on as needed basis (*but they will be loaded in groups*)

The first way - *defining them on the page*

The templates are defined in a special section on the page (*but not necessarily one section, there can be several such sections on the page*). Each section should have a special *css* class "*ria-template*", which makes the section invisible on the page and distinguishable from other regions. Each template must have a *data-role* attribute with a value "template". The templates loaded by this method available to all application's instances (*they are in the global scope and can be used by all applications*).

Note: *This method is the simplest way for the templates definition. And in many cases it is the best.*

an example of a template's section on the page:

```
@*invisible section to hold data templates*@
<section class="ria-template">

@*data template's – id attribute is mandatory*@
<div id="stackPanelItemTemplate" data-role="template" class="stackPanelItem" >
  <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 second way - *loading them all at the start from the server*

The templates are defined on the server in a text file (*at least the url should provide a text content*). The rules for a templates definition are the same as in the first way, but the text contains just the templates definitions (*without a section tag around them*). They are loaded by the application's *loadTemplates* method. (*See the DataGrid Demo for an example*). You must start the loading before calling the application's *startUp* method.

Note: *This method is not much different from defining templates on the page, only it allows not to clutter the page with templates definition and define them separately. Another difference is that each application instance will have its own copy of the templates.*

an example of a loading templates using loadTemplates:

```
RIAPP.global.addOnLoad(function (sender, a) {
        var global = sender;
        global.defaults.imagesPath = mainOptions.images_path;
        var thisApp = new DemoApplication(mainOptions);

        //example of how to load templates from the server
         thisApp.loadTemplates(mainOptions.templates_url);

        thisApp.startUp((app) => {
        });
});
```

The third way - *loading them on as needed basis*

The templates can be loaded when they are needed. The application should register a loader function per template. The loader function must return a promise which is resolved (*if all is ok*) to a *html* string.
The loader function is agnostic of the way of obtaining a template definition, you only need to return a promise from it. Each time the template is needed, the loader function will be executed. So it is advisable to cache the result inside this function, to prevent an excessive network traffic.

Note: *This method of loading of templates is not very efficient (if the caching is not used), but can be helpful when on each template's loading it should be generated on the server dynamically by the server side code.*

an example of loading templates by registering a loader function:

```
RIAPP.global.addOnLoad(function (sender, a) {
        var global = sender;
        global.defaults.imagesPath = mainOptions.images_path;
        var thisApp = new DemoApplication(mainOptions);

        thisApp.registerTemplateLoader('productEditTemplate', function () {
            return thisApp.global.$.get(mainOptions.productEditTemplate_url);
        });

        //using memoize pattern so there will not be repeated loads of the same template
        thisApp.registerTemplateLoader('sizeDisplayTemplate',
          (function() {
              var savePromise;
              return function () {
                  if (!!savePromise)
                      return savePromise;
                  savePromise = thisApp.global.$.get(mainOptions.sizeDisplayTemplate_url);
                  return savePromise;
              };
          } ())
        );

        thisApp.startUp((app) => {
        });
});
```

The fourth way - *loading templates in groups on as needed basis*

The templates can be loaded in groups (*several templates per group*), and their definitions are automatically cached on the client. For each group of templates you register a unique group's name and also the names of templates the group includes. The group registration should be done before the application's *startUp* method is invoked.
When the application will need a template, then the whole group (*containing the needed template*) will be loaded from the server (*See the Single Page Application demo for an example*). The group is loaded only one time, all the templates in the group are cached on the client. Any template from the group later on will be served from the cache.

Note: *This method is very good for complex SPAs, because the SPA usually displays many different screen views without reloading the page. So, it is good to define groups of templates per each screen view. Groups will be loaded when they are needed and only when they are needed.*

an example of loading templates in groups:

```
RIAPP.global.addOnLoad(function (sender, a) {
        var global = sender;
        global.defaults.imagesPath = mainOptions.images_path;
        var thisApp = new DemoApplication(mainOptions);

        thisApp.registerTemplateGroup('custGroup',
        {
          url: mainOptions.spa_template1_url,
          names: ["SPAcustTemplate", "goToInfoColTemplate", "SPAcustDetailTemplate", "customerEditTemplate",
         "customerDetailsTemplate", "orderEditTemplate",
         "orderDetEditTemplate", "orderDetailsTemplate", "productTemplate1", "productTemplate2",
         "prodAutocompleteTemplate"]
        });

        thisApp.registerTemplateGroup('custInfoGroup',
        {
            url: mainOptions.spa_template2_url,
            names: ["customerInfo", "salespersonTemplate1", "salespersonTemplate2",
                    "salePerAutocompleteTemplate"]
        });

        thisApp.registerTemplateGroup('custAdrGroup',
        {
            url: mainOptions.spa_template3_url,
            names: ["customerAddr", "addressTemplate", "addAddressTemplate", "linkAdrTemplate",
                    "newAdrTemplate"]
        });

        thisApp.startUp((app) => { });
});
```

2.8 *Data contents*

The *data content* is used for displaying specific content types in a predetermined way. For example, a boolean value can be displayed as a checkbox and a string value as a textbox (*using an input element with a text type*). Also, these values can have a different display if they are in a different state - a string value can be displayed as a text inside a span element (*instead of textbox element*) when the item is not in editing state.
This is an exact situation which the data content handles.

A *data-content* attribute can be used only inside data forms or data grids (*to define datacell's contents*).

If the data content is databound to an object which supports a *RIAPP.IEditable* interface (*entities and collection items implement it*), it starts to observe changes in the editing state of the object. When the state changes then the appearance of the data content is also changed.

an example of using data contents inside a data form:

```
<form action="#" style="width: 100%" data-bind="{this.dataContext}" data-view="name=dataform">
        <table style="width: 95%; border: none; table-layout: fixed; background-color: transparent;">
            <colgroup>
                <col style="width: 125px; 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:CustomerID,css:{displayCss:'custInfo',editCss:'custEdit'}"></span>
                    </td>
                </tr>
                <tr>
                    <td>Title:
                    </td>
                    <td>
                        <span data-content="fieldName:Title,css:{displayCss:'custInfo',editCss:'custEdit'}"></span>
                    </td>
                </tr>
                <tr>
                    <td>FirstName:
                    </td>
                    <td>
                        <span data-content="fieldName:FirstName,css:{displayCss:'custInfo',editCss:'custEdit'}"></span>
                    </td>
                </tr>
                <tr>
                    <td>MiddleName:
                    </td>
                    <td>
                        <span data-content="fieldName:MiddleName,css:{displayCss:'custInfo',editCss:'custEdit'}"></span>
                    </td>
                </tr>
                <tr>
                    <td>LastName:
                    </td>
                    <td>
                        <span data-content="fieldName:LastName,css:{displayCss:'custInfo',editCss:'custEdit'}"></span>
                    </td>
                </tr>
                <tr>
                    <td>Suffix:
                    </td>
                    <td>
                        <span data-content="fieldName:Suffix,css:{displayCss:'custInfo',editCss:'custEdit'}"></span>
                    </td>
                </tr>
                <tr>
                    <td>CompanyName:
                    </td>
                    <td>
                        <span data-content="fieldName:CompanyName,css:{displayCss:'custInfo',editCss:'custEdit'}"></span>
                    </td>
                </tr>
                <tr>
                    <td>SalesPerson:
                    </td>
                    <td>
                        <span data-content="template={displayID=salespersonTemplate1,editID=salespersonTemplate2},
                                   css:{displayCss:'custInfo',editCss:'custEdit'}"></span>
                    </td>
                </tr>
                <tr>
```

```
                    <td>Email:
                    </td>
                    <td>
                      <span data-content="fieldName=EmailAddress,css:{displayCss:'custInfo',editCss:'custEdit'}"></span>
                    </td>
                  </tr>
                  <tr>
                    <td>Phone:
                    </td>
                    <td>
                      <span data-content="fieldName:Phone,css:{displayCss:'custInfo',editCss:'custEdit'}"></span>
                    </td>
                  </tr>
                </tbody>
              </table>
        </form>
```

There are two types of the data content :

1) Those which use the fields directly.
2) Those which use the data templates.

The first option (*use a field*):

The simplest option, but the way it is displayed is predefined by a field's data type. For example, when the field's data type is a text, then in not editing state the data content is rendered as a text inside the <span/> tag, and when in editing state it is rendered in the <input type="text"/> tag. All these data content's types are derived from *BindingContent* type. Currently, the framework includes: *BoolContent, DateContent, DateTimeContent, NumberContent, StringContent, MultyLineContent,* and *LookupContent* types.

A class 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 data bound. The decision is made by the *ContentFactory*, which is used to create instances of the data content types in the application. But this decision can be changed by explicitly specifying the name of the data content and some data contents may also need options for their initialization.

an example of specifying a multyline option for the data content:

```
 <span data-content="fieldName:Name,css:{displayCss:'name-display',editCss:'name-
edit'},name:multyline,options:{rows:3,cols:20,wrap:hard}"></span>
```

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

```
 <span data-content="fieldName=ProductCategoryID,name:lookup,
          options:{dataSource=dbContext.dbSets.ProductCategory,
          valuePath=ProductCategoryID,textPath=Name},css:{editCss:'listbox-edit'}"></span>
```

an example of specifying a datepicker name for the data content:

```
<span data-content="fieldName:SellEndDate,name:datepicker"></span>
```

also you can use in the data content the readOnly option to ensure that it will not be displayed in the editing mode.

an example of specifying a readOnly option for the data content:

```
<th data-column="width:20%,title=Address2,sortable:true"
data-content="fieldName:Address.AddressLine2,readOnly:true" ></th>
```

The second option (*use the templates*):

It is more versatile than the first option because the template can have more complex display. Inside the template you can databind several fields.
The only drawback here - is that it needs more efforts than the first option.
The templated data content is implemented in the framework by a *TemplateContent* class.

an example of markup for a templated data content:

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

Note: *editID or displayID can be omitted.*

2.9 *User Controls*

DataGrid:

The *DataGrid* is a control for attaching the logic to the table HTML element. Without this control the table's content is static. Using this UI control the table is turned into a desktop applications equivalent of the data grid.

The *DataGrid* control adds to the table the following features:

1) Data binding to a data source
2) Inline or pop up data editors
3) Paging the data (*with the help of a pager control*)
4) Sorting the data on column clicking
5) Specialized column types (*row selector, row actions, row expander*)
6) Usage of templates for the data display and editing
7) Support for a visual row state (*its display*) based on a field's value
8) Column headers are fixed (*not scrollable with the data*) like in desktop data grids
9) Support for use of the keyboard keys (*up, down, left, right, space*)
10) Row selection (*when a row selector column is present*) by a space key
11) Navigation between pages using a pageup or pagedown keys.

The DataGrid – is defined in the datagrid.ts file. There are several types in the module which are needed for the DataGrid's functionality (*BaseCell, DataCell, ExpanderCell, ActionsCell, RowSelectorCell, DetailsCell, Row, DetailsRow, BaseColumn, DataColumn, ExpanderColumn, ActionsColumn, RowSelectorColumn, DataGrid*).

The data grid control's constructor accepts options for the control. They are defined as:

```
export interface IGridOptions {
        isUseScrollInto: boolean;
        isUseScrollIntoDetails: boolean;
        containerCss: string;
        wrapCss: string;
        headerCss: string;
        rowStateField: string;
        isCanEdit: boolean;
        isCanDelete: boolean;
        isHandleAddNew: boolean;
        details?: { templateID: string; };
        editor?: datadialog.IDialogConstructorOptions;
```

```
 }
```

Where *datadialog.IDialogConstructorOptions* are defined as:

```
export interface IDialogConstructorOptions {
                dataContext?: any;
                templateID: string;
                width?: any;
                height?: any;
                title?: string;
                submitOnOK?: boolean;
                canRefresh?: boolean;
                canCancel?: boolean;
                fn_OnClose?: (dialog: DataEditDialog) => void;
                fn_OnOK?: (dialog: DataEditDialog) => number;
                fn_OnShow?: (dialog: DataEditDialog) => void;
                fn_OnCancel?: (dialog: DataEditDialog) => number;
                fn_OnTemplateCreated?: (template: template.Template) => void;
                fn_OnTemplateDestroy?: (template: template.Template) => void;
}
```

In order to allow a declarative use of the control, there's a supplementing element view - *GridElView*.

an example of a data grid definition (HTML markup):

```
  <table data-name="gridCustAddr"
data-bind="{this.dataSource,to=custAdressView,source=customerVM.customerAddressVM}"
data-view="options={wrapCss:findAddrTableWrap,isCanDelete=false,isCanEdit=true,isUseScrollInto=false}">
  <thead>
  <tr>
 <th data-column="width:50px,type:row_actions" ></th>
 <th data-column="width:40%,title=AddressType" data-content="fieldName:AddressType" ></th>
 <th data-column="width:60%,title=Address,sortable:true" data-content="fieldName:Address.AddressLine1"></th>
  </tr>
 </thead>
               <tbody>
              </tbody>
 </table>
```

Columns in the datagrid are defined by adding to a <th /> tag a *data-column* and a *data-content* attributes.
The *data-column* attribute can have *width, title, sortable, rowCellCss, colCellCss, type, sortMemberName* options.

A *sortMemberName* option can contain several field names separated by semicolons, as in the next example:

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

The *type* option determines what is the type (*kind*) of the data column. If the *type* option is omitted, then it has a default type - the data column. The other types of columns can be *actions*, *expander* or *row selector* columns.
The *data-content* attribute is used only in the data columns (*columns for display and editing the data*) and specifies the field which will be displayed in the data cell.

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

The data content can also use the data templates.

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

The grid only displays the data when its data source is databound.

Often, there can be the need to handle datagrid's events in the view model's code.
To obtain the datagrid's instance in your view model you can databind *grid* property on the element's view using the *BackWay* databinding mode to a property which will accept the datagrid's instance on your view model.

```
data-bind="{this.grid,to=grid,mode=BackWay,source=productVM}"
```

An example of adding event handlers to a datagrid in the view model's code:

```
protected _addGrid(grid: MOD.datagrid.DataGrid): void {
                var self = this;
                if (!!this._dataGrid)
                    this._removeGrid();
                this._dataGrid = grid;
                this._dataGrid.addOnPageChanged(function (s, args) {
                    self.onDataPageChanged();
                }, this.uniqueID, this);
                this._dataGrid.addOnRowSelected(function (s, args) {
                    self.onRowSelected(args.row);
                }, this.uniqueID, this);
                this._dataGrid.addOnRowExpanded(function (s, args) {
                    if (args.isExpanded)
                        self.onRowExpanded(args.expandedRow);
                    else
                        self.onRowCollapsed(args.collapsedRow);
                }, this.uniqueID, this);
                this._dataGrid.addOnCellDblClicked(function (s, args) {
                    self.onCellDblClicked(args.cell);
                }, this.uniqueID, this);
            }
    protected _removeGrid(): void {
                if (!this._dataGrid)

                    return;
                this._dataGrid.removeNSHandlers(this.uniqueID);
                this._dataGrid = null;
            }
   get grid(): MOD.datagrid.DataGrid { return this._dataGrid; }
   set grid(v: MOD.datagrid.DataGrid) {
                if (!!v)
                    this._addGrid(v);
                else
                    this._removeGrid();
     }
```

Note: *But the SPA demo uses another solution to subscribe to the datagrid's events in the view models. It uses an extended GridElView which is derived from the framework's GridElView. This new element view exposes a property of a custom interface IGridEvents type. The element view and a view model are connected using this property using declarative databinding. This solution requires more coding, but is better for automatic testing.*
You can look at the SPA demo how it is done.

Another important thing, which is used in the declarative data grid definition, is the *data-view* attribute which allows to provide options for the control:

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

A very important option is the table's wrap style, using it, you can add a vertical scrolling for the table's data (*the table's body*), without scrolling the table's column header along with the data.

an example of the overall markup for a table rendered on the page:

```
<!-- the table container is added when grid's code is attached to the table -->
<div class="ria-table-container tableContainer">
   <!-- 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 tableHeader" 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 real table is wrapped in the div tag, for scrolling the data -->
   <div class="ria-table-wrap tableWrap">
      <table class="ria-data-table" data-view=" … "   data-bind=" … "
         data-name="gridProducts"  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 an overall markup for a table's row 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 DataGrid's options can also 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 element's 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 element's scrollInto method. The default is true.
rowStateField - a name of the field. You can implement a IRowStateProvider interface in your user module and expose it as a property in your view model. When the rowStateField is changed, IRowStateProvider's method getCSS will be invoked and the field's value can be inspected, and based on that value can be selected the css style for the row display (*see the DataGrid demo for the example*):

Note: *The DataGrid has a feature of changing the row display depending on some field's value. For example, the DataGrid demo uses this feature to display differently rows when the isActive field value true or false. It is done by providing in the grid's options the name of the field to observe, as in rowStateField:IsActive . The GridElView exposes the stateProvider property. You need to provide (usually through databinding) a class instance which implements MOD.datagrid.IRowStateProvider interface.*

An example of implementation a row state provider for the datagrid:

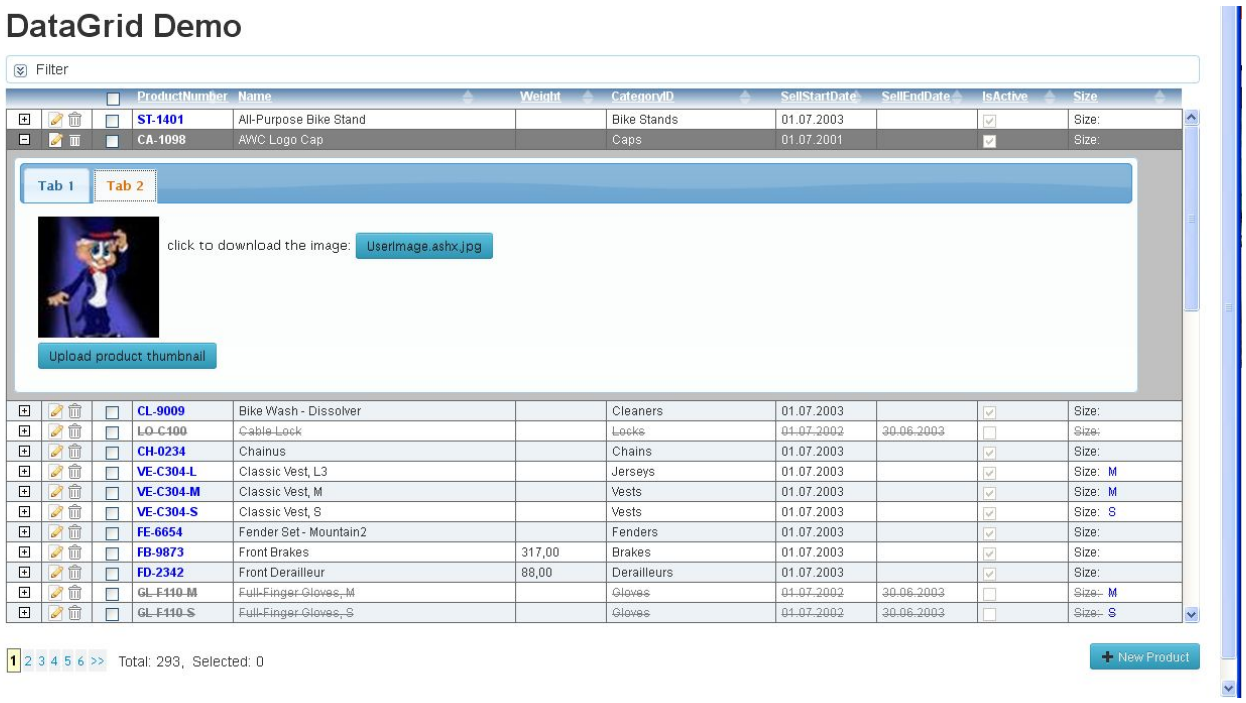
```
class RowStateProvider implements MOD.datagrid.IRowStateProvider {
        getCSS(item: collMod.ICollectionItem, val: any): string {
            return (!val) ? 'rowInactive' : null;
        }
}
```

isCanEdit and *isCanDelete* - if the options values are false. Then the grid will be in readonly mode.
isHandleAddNew - if set to true, then if the grid's datasource adds a new item, then the grid will automatically show data edit dialog for editing this item. The default is false.

The DataGrid's 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.
canRefresh - can be used to suppress showing the refresh button in the dialog.
canCancel - can be used to suppress showing the cancel button in the dialog.

An example of the DataGrid on a web page with 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		✓	Size:
CA-1098	AWC Logo Cap		Caps	01.07.2001		✓	Size:

Tab 1 Tab 2

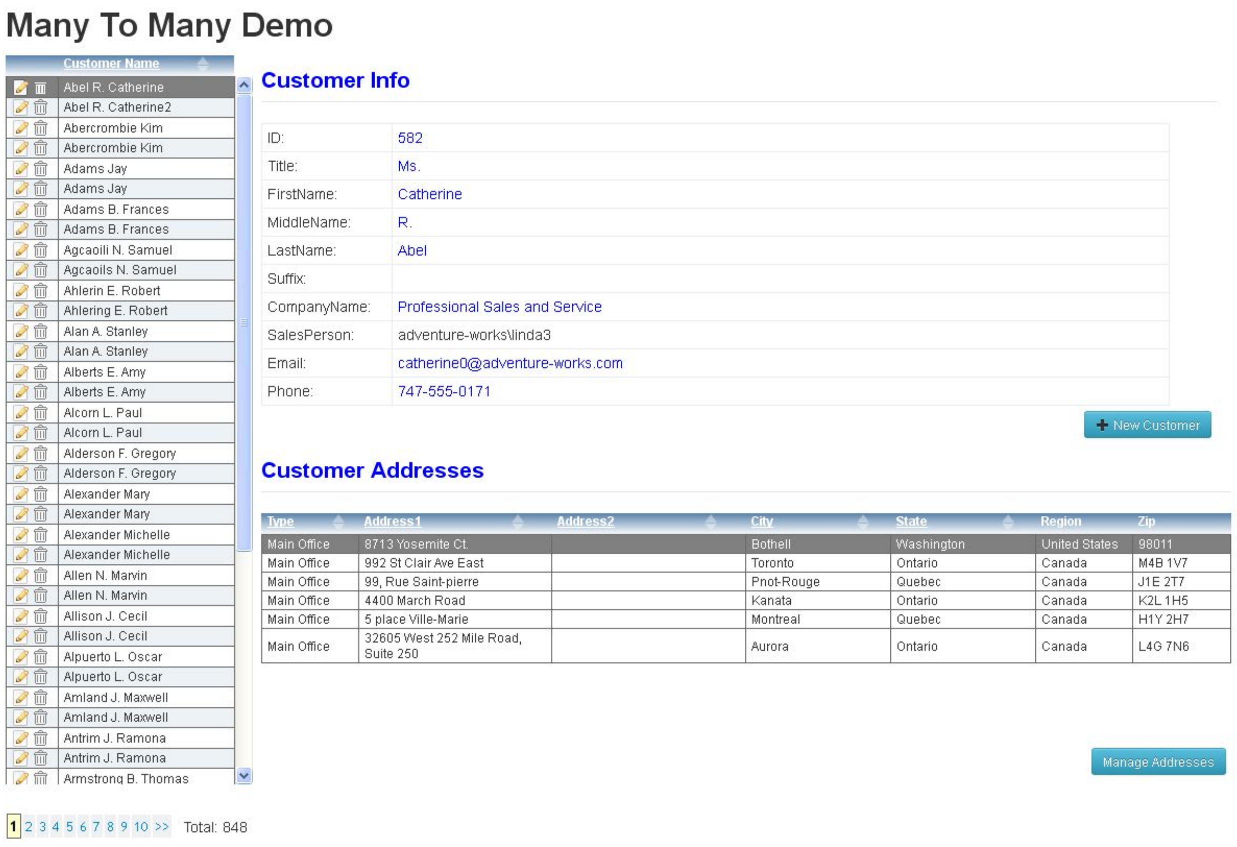
click to download the image: UserImage.ashx.jpg

Upload product thumbnail

ProductNumber	Name	Weight	CategoryID	SellStartDate	SellEndDate	IsActive	Size
CL-9009	Bike Wash - Dissolver		Cleaners	01.07.2003		✓	Size:
~~LO-C100~~	~~Cable Lock~~		~~Locks~~	~~01.07.2002~~	~~30.06.2003~~		~~Size:~~
CH-0234	Chainus		Chains	01.07.2003		✓	Size:
VE-C304-L	Classic Vest, L3		Jerseys	01.07.2003		✓	Size: M
VE-C304-M	Classic Vest, M		Vests	01.07.2003		✓	Size: M
VE-C304-S	Classic Vest, S		Vests	01.07.2003		✓	Size: S
FE-6654	Fender Set - Mountain2		Fenders	01.07.2003		✓	Size:
FB-9873	Front Brakes	317,00	Brakes	01.07.2003		✓	Size:
FD-2342	Front Derailleur	88,00	Derailleurs	01.07.2003		✓	Size:
~~GL-F110-M~~	~~Full-Finger Gloves, M~~		~~Gloves~~	~~01.07.2002~~	~~30.06.2003~~		~~Size:~~ M
~~GL-F110-S~~	~~Full-Finger Gloves, S~~		~~Gloves~~	~~01.07.2002~~	~~30.06.2003~~		~~Size:~~ S

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

An example of using several DataGrids on a web page:



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
Agcaoili N. Samuel
Agcaoils 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

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

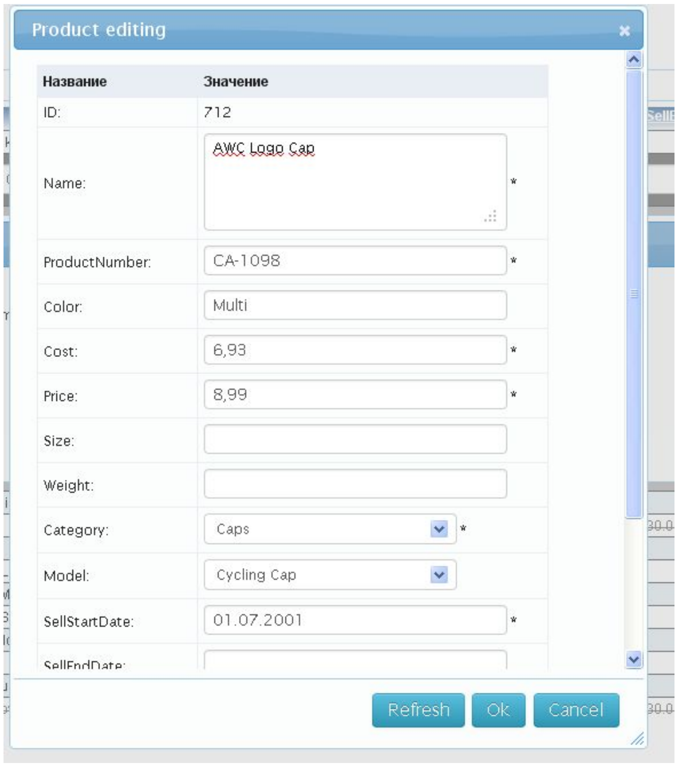
Customer Info

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

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		Pnot-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

An example of a data edit dialog display:


Product editing

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:	

DataPager:

The *Data Pager* is a UI control. Like all the controls defined in the framework, in order to use it declaratively, it has a special element view - the *PagerElView*.
The DataPager is defined in the *pager.ts* file.
In order to display it on the page, you need to add the markup, as in the example:

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

DataEditDialog:

The *DataEditDialog* is a control used to display modal popup dialogs. This control is used by the *DataGrid* control to display the edit dialog. But this control can also be used independently from the *DataGrid*.

The *DataEditDialog* uses a data template for its visual display.
The dialog's options and the dialog are defined in the *datadialog.ts* file.

```
export interface IDialogConstructorOptions {
            dataContext?: any;
            templateID: string;
            width?: any;
            height?: any;
            title?: string;
            submitOnOK?: boolean;
            canRefresh?: boolean;
            canCancel?: boolean;
            fn_OnClose?: (dialog: DataEditDialog) => void;
            fn_OnOK?: (dialog: DataEditDialog) => number;
            fn_OnShow?: (dialog: DataEditDialog) => void;
            fn_OnCancel?: (dialog: DataEditDialog) => number;
```

```
                fn_OnTemplateCreated?: (template: template.Template) => void;
                fn_OnTemplateDestroy?: (template: template.Template) => void;
}
```

For the use of the dialog in code it is useful to use a special view model:

```
 export class DialogVM extends mvvm.ViewModel<IApplication> {
       private _factories: { [name: string]: () => DataEditDialog; };
       private _dialogs: { [name: string]: DataEditDialog; };

       constructor(app: IApplication) {
           super(app);
           this._factories = {};
           this._dialogs = {};
       }
       createDialog(name: string, options: IDialogConstructorOptions) {
           var self = this;
           //the map stores functions those create dialogs (aka factories)
           this._factories[name] = function () {
               var dialog = self._dialogs[name];
               if (!dialog) {
                   dialog = new DataEditDialog(self.app, options);
                   self._dialogs[name] = dialog;
               }
               return dialog;
           };
           return this._factories[name];
       }
       showDialog(name: string, dataContext: any) {
           var dlg = this.getDialog(name);
           if (!dlg)
               throw new Error(utils.format('Invalid Dialog name:  {0}', name));
           dlg.dataContext = dataContext;
           dlg.show();
           return dlg;
       }
       getDialog(name: string) {
           var factory = this._factories[name];
           if (!factory)
               return null;
           return factory();
       }
       destroy() {
           if (this._isDestroyed)
               return;
           this._isDestroyCalled = true;
           var self= this, keys = Object.keys(this._factories);
           keys.forEach(function (key: string) {
               self._dialogs[key].destroy();
           });
           this._factories = {};
           this._dialogs = {};
           super.destroy();
       }
}
```

Then, it simplifies creation of the dialogs in code.

```
  this._dialogVM = new COMMON.DialogVM(app);
  var dialogOptions: MOD.datadialog.IDialogConstructorOptions = {
                  templateID: 'invokeResultTemplate',
                  width: 600,
                  height: 250,
                  canCancel: false, //no cancel button
                  title: 'Result of a service method invocation',
                  fn_OnClose: function (dialog) {
                      self.invokeResult = null;
                  }
```

```
 };
 this._dialogVM.createDialog('testDialog', dialogOptions);
```

With the help of *DialogVM* in order to show the dialog, it is only needed to invoke a DialogVM's *showDialog* method. The method expects two parameters: the name of the dialog, and the data context.

```
self._dialogVM.showDialog('testDialog', self);
```

Note: *You can use one DialogVM instance to create several dialogs. The dialogs are lazily initialized (that is only when they are used - on the first call to the showDialog method).*

The option's property *fn_OnTemplateCreated* requires more explanations.
This option's property is used to provide a function which will be invoked when the instance of the data template used by the dialog is created.
By using this template's instance you can get HTML DOM elements inside the template.

an example using fn_onTemplateCreated option's property:

```
//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>

//create dialog options
var dialogOptions: MOD.datadialog.IDialogConstructorOptions = {
                    templateID: 'treeTemplate',
                    width: 650,
                    height: 700,
                    title: self._includeFiles ? 'File Browser' : 'Folder Browser',
                    fn_OnTemplateCreated: function (template) {
                        var dialog = this, $ = global.$; //the function is executed in the context of the dialog
                        var $tree = global.$(fn_getTemplateElement(template, 'tree'));
                        var options: IFolderBrowserOptions = utils.mergeObj(app.options, { $tree: $tree, includeFiles:
self._includeFiles });
                        self._folderBrowser = new FolderBrowser(options);
                        self._folderBrowser.addOnNodeSelected(function (s, a) {
                            self.selectedItem = a.item;
                        }, self.uniqueID)
                    },
                    fn_OnShow: function (dialog) {
                        self.selectedItem = null;
                        self._folderBrowser.loadRootFolder();
                    },
                    fn_OnClose: function (dialog) {
                        if (dialog.result == 'ok' && !!self._selectedItem) {
                            self._onSelected(self._selectedItem, self._selectedItem.fullPath);
                        }
                    }
                };
//initialize dialog with the name and the options
 this._dialogVM.createDialog('folderBrowser', dialogOptions);
//the command which shows dialog when executed
 this._dialogCommand = new MOD.mvvm.Command(function (sender, param) {
                    try {
                        self._dialogVM.showDialog('folderBrowser', self);
                    } catch (ex) {
                        self.handleError(ex, this);
                    }
                }, self, function (sender, param) {
                    return true;}
                );
```

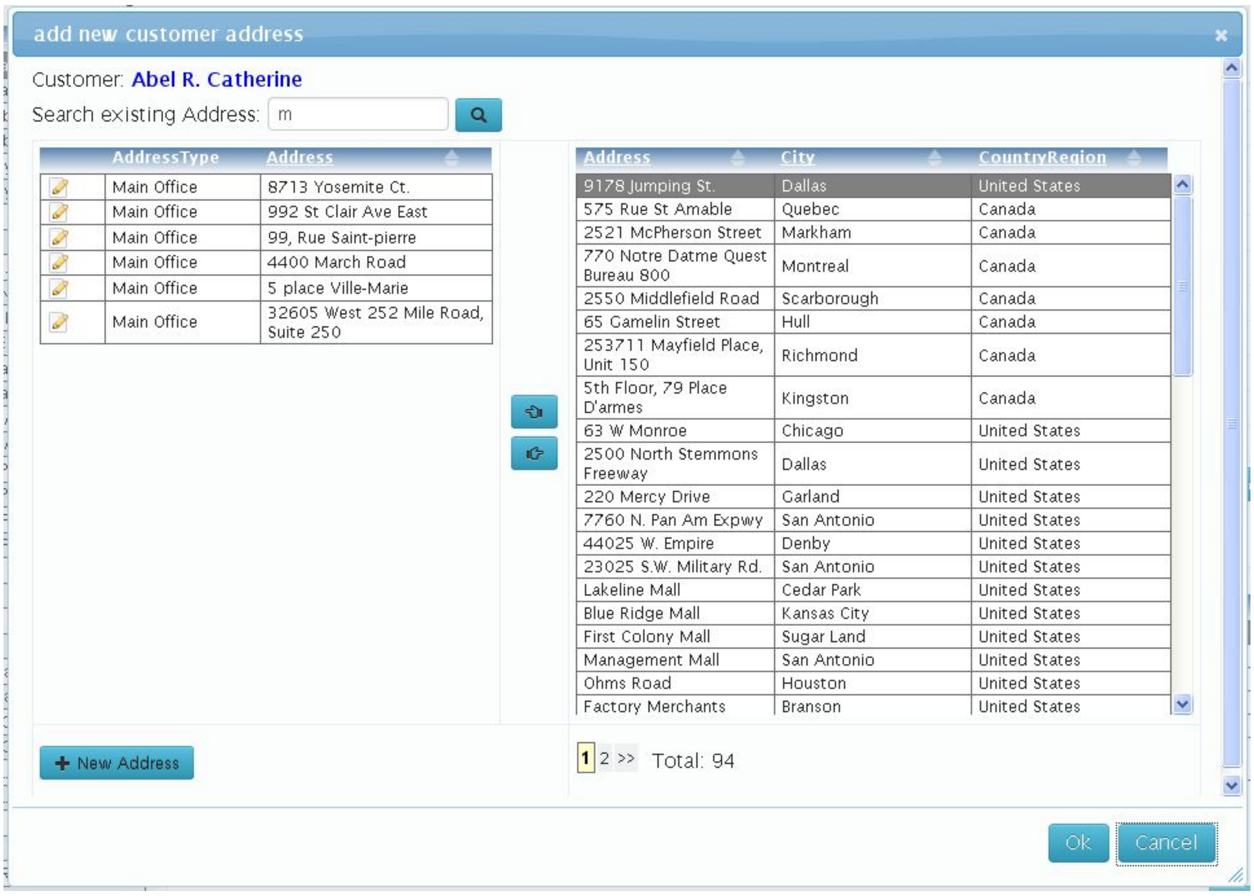
<u>The dialog has the following options:</u>

fn_OnOK - function is invoked when the user clicks the dialog's OK button. The result of this function is checked by the dialog. If this function returns *DIALOG_ACTION.StayOpen*, then the dialog is not closed (*see the ManyToMany demo's view model for example*).

submitOnOK - when is set to true, then when the OK button is clicked the dialog submits changes to the server and waits for the submit completion. If the changes are submitted without errors then the dialog is closed, in the other case the dialog stays open.

Note: *For the submitOnOk to work, the data context used for the dialog needs to implement the RIAPP.ISubmittable interface.*

<u>an example of a more complex data dialog</u>:



Customer: **Abel R. Catherine**

Search existing Address: m

	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 Datme Quest Bureau 800	Montreal	Canada
2550 Middlefield Road	Scarborough	Canada
65 Gamelin 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

<u>DataForm:</u>

The *DataForm* is a control that attaches the data context to a region. The data context is provided to the *DataForm* through its *dataContext* property.
The DataForm control is usually attached to a block tag (*<div/> or <form/>*).

The DataForm also can use data contents inside it (*they can display the data in editing state and not in editing state differently*).
The data form also automatically displays a summary of validation errors.
A dataform can have nested dataforms which use their own data context. A nested dataform's datacontext can be also databound to the parent's form data context or a property on it.

A dataform can be placed directly on the HTML page or inside a data template.

Dataforms make it easier to change the data context inside their scope and they make the databindings paths shorter, and therefore easier to write.

The DataForm class is defined in the *dataform.ts* file.
In order to make it possible to attach the DataForm to an element declaratively, there is a special element view - a *DataFormElView*. It is registered by the name *dataform*.

an example of a DataForm usage on the page:

```
 <form action="#" style="width: 100%" data-bind="{this.dataContext,to=Address}" data-view="name=dataform">
        <dl class="dl-horizontal">
            <dt><span class="addressLabel">AddressLine1:</span></dt>
            <dd>
                <!--inside data form we can use span tag with data-content attribute-->
                <span class="address" data-content="fieldName:AddressLine1"></span>
            </dd>
            <dt><span class="addressLabel">AddressLine2:</span></dt>
            <dd>
                <span class="address" data-content="fieldName:AddressLine2"></span>
            </dd>
            <dt><span class="addressLabel">City:</span></dt>
            <dd>
                <span class="address" data-content="fieldName:City"></span>
            </dd>
            <dt><span class="addressLabel">StateProvince:</span></dt>
            <dd>
                <span class="address" data-content="fieldName:StateProvince"></span>
            </dd>
            <dt><span class="addressLabel">CountryRegion:</span></dt>
            <dd>
                <span class="address" data-content="fieldName:CountryRegion"></span>
            </dd>
            <dt><span class="addressLabel">PostalCode:</span></dt>
            <dd>
                <span class="address" data-content="fieldName:PostalCode"></span>
            </dd>
        </dl>
</form>
```

StackPanel:

The *StackPanel* - is a control which is used to attach the logic and markup for displaying a vertical or horizontal list of objects to a block tag (*usually, a <div/> tag*). Each object from a collection databound to the *StackPanel*'s data source property is displayed using the data template. It is very much like ASP.NET repeater control, only fully functional on the client side. The control allows to use keyboard keys (*left, right or up,down*) to navigate to the previous and the next elements in the list.

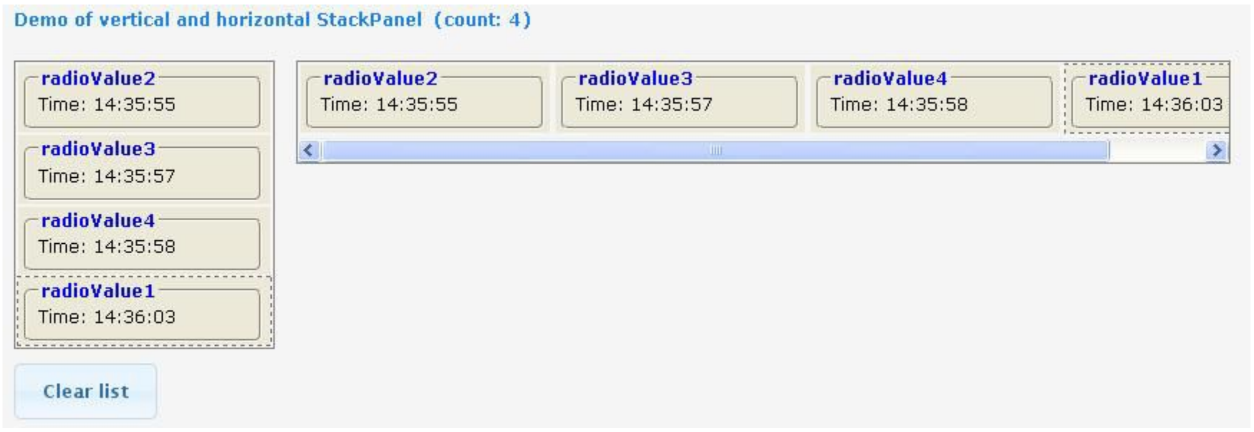
The StackPanel control is defined in the *stackpanel.ts* file. In order to use the control declaratively there is a special element view - *StackPanelElView*. It is registered by the name *stackpanel*.

an example of usage of two StackPanels on a web 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=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=demoVM}"
data-view="name=stackpanel,options:{templateID:stackPanelItemTemplate,orientation:horizontal}"></div>
```

an example of display of the above StackPanels on a web 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-elvwkey0="s_10">
   <div class="stackpanel-item" style="display: inline-block;" data-key="clkey_0">
      <div class="stackPanelItem" style="width: 170px;">
         <fieldset>
            <legend><span data-elvwkey0="s_14">radioValue2</span> </legend>Time:&nbsp; <span
               data-elvwkey0="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 a control which is used to attach the logic of a combobox to the <select/> element .
It is displayed on the page like an ordinary combobox, only the options in it are created by using the data from the datasource.

The *ListBox* control is defined in *listbox.ts* file. In order to use the control declaratively there is a special element view - *SelectElView*.
This control is also used internally in the *LookupContent* to display lookup fields for editing a text box value.

an example of using two ListBoxes on a web page:

```
<td>
            <label for="prodMCat">Main Category:</label>
             <select id="prodMCat" size="1" class="span3"
                    data-bind-1="{this.dataSource,to=filter.ParentCategories}"
                    data-bind-2="{this.selectedValue,to=filter.parentCategoryID,mode=TwoWay}"
                    data-bind-3="{this.textProvider,to=optionTextProvider}"
                    data-bind-4="{this.stateProvider,to=optionStateProvider}"
                    data-view="options:{valuePath=ProductCategoryID,textPath=Name,statePath=Name}"></select>
          </td>
          <td>
             <label for="prodSCat">Sub Category:</label>
             <select id="prodSCat" size="1" class="span2"
                    data-bind-1="{this.dataSource,to=filter.ChildCategories}"
                    data-bind-2="{this.selectedValue,to=filter.childCategoryID,mode=TwoWay}"
                    data-bind-3="{this.selectedItem,to=filter.selectedCategory,mode=TwoWay}"
                    data-bind-4="{this.toolTip,to=filter.selectedCategory.Name}"
                    data-bind-5="{this.textProvider,to=optionTextProvider}"
                    data-bind-6="{this.stateProvider,to=optionStateProvider}"
                    data-view="options:{valuePath=ProductCategoryID,textPath=Name,statePath=Name}"></select>
          </td>
```

The SelectElView exposes the textProvider and the stateProvider properties. They can be used to assign objects (usually through databinding) which will provide custom text for the list options and css class for them, respectively. The statePath is used to provide a name for the field which on changing its value will change the option's state (the same semantics as the rowStateField in the case of datagrid).
You can look at the GridDemo project for example how it is used.

An example of implementation of an option text and state providers for a listBox:

```
class OptionTextProvider implements MOD.listbox.IOptionTextProvider {
          getText(item: collMod.ICollectionItem, itemIndex: number, text: string): string {
              if (itemIndex > 0)
                  return itemIndex + ') ' + text;
              else
                  return text;
          }
  }

 class OptionStateProvider implements MOD.listbox.IOptionStateProvider {
          getCSS(item: collMod.ICollectionItem, itemIndex: number, val: any): string {
              //var name: string = val;
              if (itemIndex % 2 == 0)
                  return "gray-bgc";
              else
                  return "white-bgc";
          }
 }
```

Working with the data on the client side

3.1 *Working with the simple collection's data*

The framework's collection hierarchy starts from generic *BaseCollection<TItem extends CollectionItem>* type (*defined in the collection.ts file*), which is the base abstract type for all specialized collections. The *CollectionItem* is the base class for all item types in these collections. Each collection item implements *ICollectionItem* interface

```
interface ICollectionItem extends IBaseObject {
              _aspect: ItemAspect<ICollectionItem>;
              _key: string;
```

}

These base classes are defined in the *collection* module, which has also *BaseList*, and *BaseDictionary* definitions. All the collections in the framework use generics argument for their item's type.
These collections are used as data sources for the controls like the *DataGrid*, the *StackPanel*, the *ListBox*.
Every collection instance has the *currentItem* property and events which notify the controls about changing the current position, adding or removing an item, and also about starting or ending of editing of the item. Only one item in the collection can be in the editing state.

All the types in the framework, including a *BaseCollection* and a *CollectionItem* classes are descendants of the *BaseObject* class, and they inherit all the properties, methods and events of that base type.

Collection's properties:

Property	Is readOnly	Description
permissions	Yes	Exposes permissions for updating, inserting and refreshing the entities in the collection. *export interface IPermissions { canAddRow: boolean; canEditRow: boolean; canDeleteRow: boolean; canRefreshRow: boolean; }*
currentItem	No	The 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 supports the paging.
isEditing	Yes	Returns true if the collection is in editing state.
isHasErrors	Yes	Returns true if the collection items have validation errors.
isLoading	No	Returns true if the items are loading.
isUpdating	No	Can be set to true to mark the collection for bulk updates. Used when it is needed to update the items without triggering *begin_edit* and *end_edit* events. Which prevents UI controls from flickering.
pageCount	Yes	Calculated number of the pages (*if the paging is supported*)
options	Yes	Exposes the options object. export interface ICollectionOptions { enablePaging: boolean; pageSize: number; }

Collection's methods:

Method	Description
getFieldInfos	Returns an array of a IFieldInfo for the item's fields.
getFieldInfo	Returns information about a field properties by its name. export interface IFieldInfo { isPrimaryKey: number;

	isRowTimeStamp: boolean;
	dataType: number;
	isNullable: boolean;
	maxLength: number;
	isReadOnly: boolean;
	isAutoGenerated: boolean;
	allowClientDefault: boolean;
	dateConversion: number;
	isClientOnly: boolean;
	isCalculated: boolean;
	isNeedOriginal: boolean;
	dependentOn: string;
	range: string;
	regex: string;
	isNavigation: boolean;
	fieldName: string;
	dependents?: string[];
	fullName?: string;
	}

getFieldNames	Returns an array of field names.
cancelEdit	Cancels editing
endEdit	Ends editing if there are no validation errors and returns true, otherwise return false without ending the editing.
getItemsWithErrors	Returns an array of the items which have validation errors.
addNew	Creates and returns a new item. Leaving the collection in the editing state. You can cancel adding the new item by invoking a cancelEdit method, otherwise you can commit editing with calling the endEdit method.
getItemByPos	Returns item (*if item is found*) by position or a null value.
getItemByKey	Returns an item by the key (*or a null*). The key value is a CollectionItem's property _key which has a string type. The Dictionary class uses one property on the item as a key value. The DbSet's items are returned from the data service layer and they all have a server side generated key (*string values of the primary key separated by* ***;***). If an item is created on the client side before it is submitted to the server, then the _key property have a client side generated value (*till it successfully submitted*).
findByPK	Returns an item (or a null) by primary key values. Primary key values provided as arguments ordered exactly as the fields in the primary key.
moveFirst	Moves the current position to the first item in the collection. Returns a boolean value indicating if the move was successful. Accepts an optional boolean parameter *skipDeleted*, which indicates if deleted (*and not submitted*) items should be skipped.
movePrev	Moves the current position to the previous item in the collection. Returns a boolean value indicating if the move was successful. Accepts an optional boolean parameter *skipDeleted*, which indicates if deleted (*and not submitted*) items should be skipped.
moveNext	Moves the current position to the next item in the collection. Returns a boolean value indicating if the move was successful. Accepts an optional boolean parameter *skipDeleted*, which indicates if deleted (*and not submitted*) items should be skipped.
moveLast	Moves the current position to the last item in the collection. Returns a boolean value indicating if the move was successful. Accepts an optional boolean parameter *skipDeleted*, which

	indicates if deleted (*and not submitted*) items should be skipped.
goTo	Moves the current item position to the one provided in the method's argument. Returns a boolean value indicating if the move was successful.
forEach	The same functionality as an array's forEach method for iterating on the collection's items.
removeItem	Immediately removes the item from the collection.
sortLocal	Sorts the collection items locally on the client. The first parameter must be an array of field names, the second 'ASC' or 'DESC'.
sortLocalByFunc	Sorts the collection items locally on the client. Expects a sorting function.
clear	Removes all the items from the collection.

Collection's events:

Event	Description
begin_edit	Triggered when an item started editing.
end_edit	Triggered when an item ended editing.
fill	Triggered when the population of the collection with items is started or ended.
coll_changed	Triggered when the collection has been changed - items added, removed, item has changed the *_key* property, or the collection has been reset. (*It is primarily used by UI controls for observing the collection changes*)
item_deleting	Triggered before an item was deleted.
item_added	Triggered after a new item was added to the collection.
item_adding	Triggered before a new item was added to the collection.
validate	Triggered when an item is validated.
current_changing	Triggered when the current item is changing.
page_changing	Triggered when the current page is changing.
errors_changed	Triggered when an item's error status is changed.
status_changed	Triggered when an item's status property value (*deleted, updated, unchanged, added*) is changed.
clearing	Triggered before the collection is cleared (*emptied*)
cleared	Triggered after the collection is cleared (*emptied*)
commit_changes	Triggered when the changes on a collection item are accepted or rejected.

The collection item exposes two properties _key and _aspect in addition to its properties exposing the data fields of the item.
The _key property exposes the unique string value (*the key*) of the item.
The _aspect property exposes ItemAspect<TItem extends ICollectionItem> class instance.
This aspect property has properties and methods which are needed to work with the item and the collection uses them internally and they can be also used in the code.

ItemAspect's properties:

Property	Is readOnly	Description
isNew	Yes	Returns true if the item is created by the *addNew* method on the collection and before the changes are

		commited to the server.
isDeleted	Yes	Returns true if the item in the deleted state.
_key	False	Returns the key (*a string value*) value of the item in the collection.
collection	Yes	Returns parent collection. (*DbSet, List, Dictionary*)
isUpdating	Yes	Returns true if the collection in the updating state.

isEditing	Yes	Returns true if the item in the editing state.
		It is part of implementation of the IEditable interface:
		export interface IEditable {
		beginEdit(): boolean;
		endEdit(): boolean;
		cancelEdit(): boolean;
		isEditing: boolean;
		}

isCanSubmit	Yes	Returns true if the item supports submitting changes to the server.
		It is part of implementation of the ISubmittable interface:
		export interface ISubmittable {
		submitChanges(): IVoidPromise;
		isCanSubmit: boolean;
		}

status	Yes	Returns the status of changes for the item *export enum STATUS { NONE= 0, ADDED= 1, UPDATED= 2, DELETED= 3 }*

ItemAspect's methods:

Property	Description
getFieldInfo	Returns a *IFieldInfo* by the field's name.
getFieldNames	Returns an array of field names.
beginEdit	Starts items's editing. Returns true if the editing started successfully.
endEdit	Ends (*commits*) items's editing. Returns true if the editing ended successfully.
cancelEdit	Cancels current items's editing. Returns true if the editing canceled successfully.
deleteItem	Deletes the item. Returns true if the item is really deleted.
addOnItemErrorsChanged	Adds an event handler for the errors change event

getFieldErrors	Returns an array of *IValidationInfo* for the field by field's name. If field's name is asterisk * , then it returns a result of the full item's (*all the fields*) validation.
	export interface IValidationInfo {
	fieldName: string;
	errors: string[];
	}

getAllErrors	Returns an array of *IValidationInfo* for the item.
getErrorString	Returns the errors in a string form.
submitChanges	Submits changes to the server if the collection supports it.
getIsHasErrors	Returns true if the item has any validation errors.

ItemAspect's events:

Event	Description
errors_changed	Occurs when the item's errors collection is changed.

The descendants of the collection type:

BaseList – is a simple base collection for all lists.
BaseDictionary – a base collection for all dictionaries. It has also a *keyName* parameter in constructor arguments, so the items are indexed by the key and can be found by their keys.
The *BaseList* and *BaseDictionary* have a toArray method which returns an array of simple objects instead of collection items.

The strongly typed lists and dictionaries are usually generated by using the DataService's code generation feature. It is used to generate strongly typed collections definitions from the server side types and it also generates client side domain models from its server side counterpart.

an example of creation of a Dictionary instance and filling its items:

```
this._months = new DEMODB.KeyValDictionary();
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);
```

3.2 *Working with the data obtained from the DataService* (*DomainService*)

In order to work with the data originated from the server side in a consistent and safe way there must be a set of components which implement a protocol of communication between the server and the client side. For the server side there is a *DomainService* which provides the data, accepts the updates originated on the client side, checks permissions for the clients to execute certain operations on the server, validates the updates, and then it provides the result of the operations back to the clients.
For the updates on the server (*CRUD operations*) there is often a need to make these updates in the sequence order of the relationship between the entities. The entities can also have generated on the server fields' values which must be propagated back to the client after updates. The server and client sides must have all the metadata which describes entities, relationship between them. The metadata is also used for validation purposes.

In the client side, the components that work with the DomainService are implemented in the db.ts file. The main component which communicates with the DomainService is the DbContext class.

The DbContext:

The instance of the DbContext is used to communicate with the data service.
The DbContext stores the data in collections (*which are classes derived from a DbSet class*).

The DbContext prevents repeated loading of the same entities into the DbSet. If the entity is loaded repeatedly then it does'nt replace the entity in the collection but only refreshes its data. The DbContext checks entities' uniqueness by comparing their primary keys. Each entity in the DbSet must have a primary key.

DbContext's properties:

Property	Is readonly	Description
isBusy	Yes	Boolean property, which indicates that the DbContext is doing some work (*typically, asynchronous*).
isSubmiting	Yes	Boolean property, which indicates that the DbContext is submitting updates to the data service.
serverTimezone	Yes	The timezone on the server from which the page was loaded.
dbSets	Yes	Exposes the DbSets by their names.
serviceMethods	Yes	A map (*indexed by the names*) of the service methods (*invocable from the client methods exposed from the data service*). A 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 is failed.
isHasChanges	Yes	Boolean property which indicates that the DbContext has pending changes on the client side.
serviceUrl	Yes	Returns the data service's url.
isInitialized	Yes	Returns true after the DbContext is initialized with the metadata - it indicates that the DbContext is ready to work with. That means the *initialize* method was called.

DbContext's methods:

Property	Description
initialize	Initializes the DbContext, with a data service's url and permissions. If you don't provide permissions in the method's argument, then the DbContext will load them from the data service.
getDbSet	Returns the DbSet by its name.
submitChanges	Submits changes to the data service. It is an asynchronous operation and returns a promise.
load	Loads the data from the data service. It is an asynchronous operation and accepts a query object. It returns a promise *IPromise<IQueryResult<IEntityItem>>*.
acceptChanges	Accepts all the changes made on the clients side. It changes entities' statuses to the *STATUS.None*. It is automatically invoked when the DbContext successfully submits the changes.
rejectChanges	Rejects all the changes. It changes entities' statuses to the *STATUS.None* and the values are restored to the original ones.
getAssociation	Returns a parent-child association object instance by its name.

DbContext's events:

Event	Description
submit_error	Triggered when submitting the changes is not successful. It allows to handle the submit error without rejecting all the changes (*if isHandled was set to true*).

The DbContext that is defined in the *db.ts* file is not used directly. Instead the class is used as base class for a concrete DbContext class. The DataService's *code generation* feature can create a script with the strongly typed entity classes and the strongly typed DbContext. It exposes strongly typed *dbSets* , *serviceMethods*, and *associations* properties.

an example of a concrete DbContext class:

```
 export class DbContext extends RIAPP.MOD.db.DbContext {
          protected _initDbSets() {
             super._initDbSets();
             this._dbSets = new DbSets(this);
             var associations = [association info here ...];
             this._initAssociations(associations);
             var methods = [imethos info here  ...];
             this._initMethods(methods);
          }
          get associations() { return <IAssocs>this._assoc; }
          get dbSets() { return <DbSets>this._dbSets; }
          get serviceMethods() { return <ISvcMethods>this._svcMethods; }
       }
```

an example of creation of a strongly typed DbContext:

```
this._dbContext = new SVMDB.DbContext();
this._dbContext.initialize({ serviceUrl: options.service_url, permissions: options.permissionInfo });
```

an example of loading data from the server using a query and using filtering and sorting criteria:

```
load() {
            //you can create several methods on the service which return the same entity type
            //but they must have different names (no overloads)
            //the query's service method can accept additional parameters which you can supply with query
            var query = this.dbSet.createReadProductQuery({ param1: [10, 11, 12, 13, 14], param2: 'Test' });
            query.pageSize = 50;
          //load 20 pages at once (but only one will be visible, the others will be in the local cache)
            query.loadPageCount = 20;
          //clear the local cache when a new batch of data is loaded from the server
            query.isClearCacheOnEveryLoad = true;

            addTextQuery(query, 'ProductNumber', this._filter.prodNumber);
            addTextQuery(query, 'Name', this._filter.name);
            if (!utils.check.isNt(this._filter.childCategoryID)) {
               query.where('ProductCategoryID', MOD.collection.FILTER_TYPE.Equals, [this._filter.childCategoryID]);
            }
            if (!utils.check.isNt(this._filter.modelID)) {
               query.where('ProductModelID', MOD.collection.FILTER_TYPE.Equals, [this._filter.modelID]);
            }

            if (!utils.check.isNt(this._filter.saleStart1) && !utils.check.isNt(this._filter.saleStart2)) {
               query.where('SellStartDate', MOD.collection.FILTER_TYPE.Between, [this._filter.saleStart1,
this._filter.saleStart2]);
            }
            else if (!utils.check.isNt(this._filter.saleStart1))
               query.where('SellStartDate', MOD.collection.FILTER_TYPE.GtEq, [this._filter.saleStart1]);
            else if (!utils.check.isNt(this._filter.saleStart2))
               query.where('SellStartDate', MOD.collection.FILTER_TYPE.LtEq, [this._filter.saleStart2]);

            query.orderBy('Name').thenBy('SellStartDate', MOD.collection.SORT_ORDER.DESC);
            return query.load();
}
```

Where the *addTextQuery* is a helper function defined in the same module as:

```
function addTextQuery(query: MOD.db.DataQuery, fldName: string, val) {
            var tmp;
            if (!!val) {
                if (utils.str.startsWith(val, '%') && utils.str.endsWith(val, '%')) {
                    tmp = utils.str.trim(val, '% ');
                    query.where(fldName, collMod.FILTER_TYPE.Contains, [tmp])
                }
                else if (utils.str.startsWith(val, '%')) {
                    tmp = utils.str.trim(val, '% ');
                    query.where(fldName, collMod.FILTER_TYPE.EndsWith, [tmp])
                }
                else if (utils.str.endsWith(val, '%')) {
                    tmp = utils.str.trim(val, '% ');
                    query.where(fldName, collMod.FILTER_TYPE.StartsWith, [tmp])
                }
                else {
                    tmp = utils.str.trim(val);
                    query.where(fldName, collMod.FILTER_TYPE.Equals, [tmp])
                }
            }
            return query;
};
```

Note: *You can can define several query methods on the data service which return the same entity type but they must have different names (no overloads). These methods can accept arguments which can be used in querying of the data.*

Note: *Generally it is better to use the query's load method instead of dbContext's load method. You can use a Promise returned by the load method to wait for the loading is completed.*

The DbContext class also allows to execute methods on the DataService annotated with the *Invoke* attribute. They are useful to execute some arbitrary code on the server side. They can accept arguments (*complex types is also allowed*) and can return result (*can be also a complex type*). The DataService's code generation feature generates strongly typed versions of those methods, which are easy to use from the client side code.

an example of two generated call signatures for the service methods:

```
export interface ISvcMethods {
            TestInvoke: (args: {
                param1: number[];
                param2: string;
            }) => IPromise<string>;
            TestComplexInvoke: (args: {
                info: IAddressInfo2;
                keys: IKeyVal[];
            }) => IVoidPromise;
 }
```

an example of a service method invocation from the client side code:

```
self.invokeResult = null;
var promise = self.dbContext.serviceMethods.TestInvoke({ param1: [10, 11, 12, 13, 14], param2: param.Name });
     promise.done(function (res) {
                        self.invokeResult = res;
                        self.showDialog('testDialog');
                    });

      promise.fail(function () {
                        //do something on fail if you need
                        //but the error message is shown always in this case
                    });
```

DataCache:

The DataCache is used to cache the data on the client side. You can set the query's *loadPageCount* property to the value more than 1. If the *loadPageCount* value is more than 1 then the loading operation returns several pages of the data (*the number you set on the loadPageCount or less if there's less than needed data*).
Those extra pages of the data rows are cached inside the query's instance with the help of a DataCache class instance.
If a DbSet's *pageIndex* property value is changed (*when going to another data page in the data grid*), then before loading the data from the server it is checked in the local cache for availability. If it exists then the page's data is served from the local cache.

Note: *the local data caching is very useful for returning more data rows than can be displayed in the DataGrid (several pages at once). If the query execution is slow (for complex queries), and if navigation from one page to another takes a considerable time, then it is better to preload several pages of the data to the client in one query operation.*

an example of a query to return several pages at once

```
  //the query has includeNav parameter on the server side
 var query = this._dbSet.createReadCustomerQuery({ includeNav: false });

 query.pageSize = 50;
  //so we load 20 pages at once -- 1000 rows
 query.loadPageCount = 20;
 //clear the previous cache data for each loading data from the server
 query.isClearCacheOnEveryLoad = true;
 query.orderBy('LastName').thenBy('MiddleName').thenBy('FirstName');
 var promise =  query.load();

promise.done(function (res) {
            //although we loaded 1000 rows, fetchedItems have only rows for a one data page - 50 rows
            //the caching is transparent, you work as if you loaded one page
            var customers = res.fetchedItems;
});
```

Note: *The DbSet class has a **fill** event (you can use an* addOnFill *method which is used to subscribe to the event) when the dbSet is starting to fill with new data or the filling is completed (*it includes the case when dbSet's pageIndex is changed*). By using this event you can chain load related dbSets. For example, when the filling of a parent dbSet is ended, you can start loading children dbSet, retrieving only entities related to the fetched parent entities.*

an example of chain loading of the related entities

```
 var customerDbSet = this.dbContext.dbSets.Customer,
                  customerAddressDbSet = this.dbContext.dbSets.CustomerAddress,
                  addressDbSet = this.dbContext.dbSets.Address;

                //when the customers filling is ended, then fill the related entities
                customerDbSet.addOnFill(function (sender, args) {
                  if (args.isBegin)
                     return;
                   //notice, fetchedItems don't include all rows fetched from the server
                   //it includes the data for the current data page only
                   var custIDs:number[] = args.fetchedItems.map(function (item) {
                      return item.CustomerID;
                   });

                   var query = customerAddressDbSet.createReadAddressForCustomersQuery({ custIDs: custIDs });
                   query.isClearPrevData = true;
```

```
            var promise = query.load();
            //load related addresses based on what customerAddress items just loaded
            promise.done(function (res) {
                var addressIDs = res.fetchedItems.map(function (item) {
                    return item.AddressID;
                });
                var query = addressDbSet.createReadAddressByIdsQuery({ addressIDs: addressIDs });
                query.isClearPrevData = true;
                return query.load();
            });
        });

        //start loading customers
        var query = customerDbSet.createReadCustomerQuery({ includeNav: false });
        query.pageSize = 50;
        query.loadPageCount = 20;
        //clear the previous cached data for each loading data from the server
        //that means when another fetching rows from the server will be,
        //the previously cached data will be replaced with the new one
        query.isClearCacheOnEveryLoad = true;
        query.orderBy('LastName').thenBy('MiddleName').thenBy('FirstName');
        query.load();
```

Note: *you can see an example of chain loading of the related entities in the Many to Many demo example, included in the demo project.*

*The DbSet class also has a **load** event which you can use to get all the items loaded from the server. The **fill** event only gives the items for the current data page, although it can load at once many more and the extra goes to the data cache.*

DataQuery:

The DbSet's *createQuery* method return an instance of the *DataQuery* class, which can be used to modify the query's options and to provide additional query parameters.

DataQuery's properties:

Property	Is readonly	Description
loadPageCount	No	determines, how many pages of the data to load. For example: *If the pageSize is 100 and loadPageCount is 25 then it will try to load 2500 records from the server*.
isClearCacheOnEveryLoad	No	determines if the cached data should be cleared when the DbContext's load method is called explicitly. *If you will set it to true then the already cached data will be cleared. If you will set it to false, then the new data will be appended to the previous data.*
isIncludeTotalCount	No	determines if the query will try to return the total number of the records.
params	No	used to set the query parameters if the data service query method expects arguments.
pageIndex	No	Usually pageIndex property of the query is not used, because it is set automatically set by the DbSet which created the query. But If you set the pageIndex property value on the query to a negative value (-1) then it will

		load the data from the server without using data paging (*all results of the query*).

Applications generally use a strongly typed DataQuery. Every strongly typed DbSet (*generated by the data service*) exposes strongly typed *createQuery* methods.

an example of a strongly typed DbSet generated by the DataService's GetTypeScript method:

```
 export class CustomerDb extends dbMOD.DbSet<Customer, DbContext>
   {
      constructor(dbContext: DbContext) {
         var self = this, opts: dbMOD.IDbSetConstuctorOptions = {
            dbContext: dbContext,
            dbSetInfo: { "fieldInfos": null, "enablePaging": true, "pageSize": 25, "dbSetName": "Customer" },
            childAssoc: ([]),
            parentAssoc: ([{ "name": "CustAddrToCustomer", "parentDbSetName": "Customer", "childDbSetName":
"CustomerAddress", "childToParentName": "Customer", "parentToChildrenName": "CustomerAddresses",
"onDeleteAction": 0, "fieldRels": [{ "parentField": "CustomerID", "childField": "CustomerID" }] }, { "name":
"OrdersToCustomer", "parentDbSetName": "Customer", "childDbSetName": "SalesOrderHeader",
"childToParentName": "Customer", "parentToChildrenName": null, "onDeleteAction": 0, "fieldRels": [{ "parentField":
"CustomerID", "childField": "CustomerID" }] }])
         }, utils = RIAPP.global.utils;
         opts.dbSetInfo.fieldInfos = ([{ "fieldName": "CustomerID", "isPrimaryKey": 1, "dataType": 3, "isNullable": false,
"isReadOnly": true, "isAutoGenerated": true, "isNeedOriginal": true, "maxLength": 4, "dateConversion": 0,
"allowClientDefault": false, "range": "", "regex": "", "fieldType": 0, "dependentOn": "", "nested": null }, { "fieldName":
"NameStyle", "isPrimaryKey": 0, "dataType": 2, "isNullable": false, "isReadOnly": false, "isAutoGenerated": false,
"isNeedOriginal": true, "maxLength": 1, "dateConversion": 0, "allowClientDefault": false, "range": "", "regex": "",
"fieldType": 0, "dependentOn": "", "nested": null }, { "fieldName": "Title", "isPrimaryKey": 0, "dataType": 1, "isNullable":
true, "isReadOnly": false, "isAutoGenerated": false, "isNeedOriginal": true, "maxLength": 8, "dateConversion": 0,
"allowClientDefault": false, "range": "", "regex": "", "fieldType": 0, "dependentOn": "", "nested": null }, { "fieldName":
"Suffix", "isPrimaryKey": 0, "dataType": 1, "isNullable": true, "isReadOnly": false, "isAutoGenerated": false,
"isNeedOriginal": true, "maxLength": 10, "dateConversion": 0, "allowClientDefault": false, "range": "", "regex": "",
"fieldType": 0, "dependentOn": "", "nested": null }, { "fieldName": "CompanyName", "isPrimaryKey": 0, "dataType": 1,
"isNullable": true, "isReadOnly": false, "isAutoGenerated": false, "isNeedOriginal": true, "maxLength": 128,
"dateConversion": 0, "allowClientDefault": false, "range": "", "regex": "", "fieldType": 0, "dependentOn": "", "nested":
null }, { "fieldName": "SalesPerson", "isPrimaryKey": 0, "dataType": 1, "isNullable": true, "isReadOnly": false,
"isAutoGenerated": false, "isNeedOriginal": true, "maxLength": 256, "dateConversion": 0, "allowClientDefault": false,
"range": "", "regex": "", "fieldType": 0, "dependentOn": "", "nested": null }, { "fieldName": "PasswordHash",
"isPrimaryKey": 0, "dataType": 1, "isNullable": false, "isReadOnly": true, "isAutoGenerated": true, "isNeedOriginal":
true, "maxLength": 128, "dateConversion": 0, "allowClientDefault": false, "range": "", "regex": "", "fieldType": 0,
"dependentOn": "", "nested": null }, { "fieldName": "PasswordSalt", "isPrimaryKey": 0, "dataType": 1, "isNullable": false,
"isReadOnly": true, "isAutoGenerated": true, "isNeedOriginal": true, "maxLength": 10, "dateConversion": 0,
"allowClientDefault": false, "range": "", "regex": "", "fieldType": 0, "dependentOn": "", "nested": null }, { "fieldName":
"rowguid", "isPrimaryKey": 0, "dataType": 9, "isNullable": false, "isReadOnly": true, "isAutoGenerated": true,
"isNeedOriginal": true, "maxLength": 16, "dateConversion": 0, "allowClientDefault": false, "range": "", "regex": "",
"fieldType": 4, "dependentOn": "", "nested": null }, { "fieldName": "ModifiedDate", "isPrimaryKey": 0, "dataType": 6,
"isNullable": false, "isReadOnly": true, "isAutoGenerated": true, "isNeedOriginal": true, "maxLength": 8,
"dateConversion": 0, "allowClientDefault": false, "range": "", "regex": "", "fieldType": 0, "dependentOn": "", "nested":
null }, { "fieldName": "ComplexProp", "isPrimaryKey": 0, "dataType": 0, "isNullable": true, "isReadOnly": false,
"isAutoGenerated": false, "isNeedOriginal": true, "maxLength": -1, "dateConversion": 0, "allowClientDefault": false,
"range": "", "regex": "", "fieldType": 5, "dependentOn": "", "nested": [{ "fieldName": "FirstName", "isPrimaryKey": 0,
"dataType": 1, "isNullable": false, "isReadOnly": false, "isAutoGenerated": false, "isNeedOriginal": true, "maxLength":
50, "dateConversion": 0, "allowClientDefault": false, "range": "", "regex": "", "fieldType": 0, "dependentOn": "", "nested":
null }, { "fieldName": "MiddleName", "isPrimaryKey": 0, "dataType": 1, "isNullable": true, "isReadOnly": false,
"isAutoGenerated": false, "isNeedOriginal": true, "maxLength": 50, "dateConversion": 0, "allowClientDefault": false,
"range": "", "regex": "", "fieldType": 0, "dependentOn": "", "nested": null }, { "fieldName": "LastName", "isPrimaryKey":
0, "dataType": 1, "isNullable": false, "isReadOnly": false, "isAutoGenerated": false, "isNeedOriginal": true,
"maxLength": 50, "dateConversion": 0, "allowClientDefault": false, "range": "", "regex": "", "fieldType": 0,
"dependentOn": "", "nested": null }, { "fieldName": "Name", "isPrimaryKey": 0, "dataType": 1, "isNullable": true,
"isReadOnly": false, "isAutoGenerated": false, "isNeedOriginal": true, "maxLength": -1, "dateConversion": 0,
"allowClientDefault": false, "range": "", "regex": "", "fieldType": 2, "dependentOn":
"ComplexProp.FirstName,ComplexProp.MiddleName,ComplexProp.LastName", "nested": null }, { "fieldName":
"ComplexProp", "isPrimaryKey": 0, "dataType": 0, "isNullable": true, "isReadOnly": false, "isAutoGenerated": false,
"isNeedOriginal": true, "maxLength": -1, "dateConversion": 0, "allowClientDefault": false, "range": "", "regex": "",
"fieldType": 5, "dependentOn": "", "nested": [{ "fieldName": "EmailAddress", "isPrimaryKey": 0, "dataType": 1,
"isNullable": true, "isReadOnly": false, "isAutoGenerated": false, "isNeedOriginal": true, "maxLength": 50,
"dateConversion": 0, "allowClientDefault": false, "range": "", "regex": "^[_a-z0-9-]+(\\.[_a-z0-9-]+)*@[a-z0-9-]+(\\.[a-z0-9-]+)*(\\.[a-z]{2,4})$", "fieldType": 0, "dependentOn": "", "nested": null }, { "fieldName": "Phone", "isPrimaryKey": 0, "dataType": 1, "isNullable": true, "isReadOnly": false, "isAutoGenerated": false, "isNeedOriginal": true, "maxLength": 25, "dateConversion": 0, "allowClientDefault": false, "range": "", "regex": "", "fieldType": 0, "dependentOn": "", "nested": null }] }] }, { "fieldName": "AddressCount", "isPrimaryKey": 0, "dataType": 3, "isNullable": true, "isReadOnly": false, "isAutoGenerated": false, "isNeedOriginal": true, "maxLength": -1, "dateConversion": 0, "allowClientDefault": false, "range": "", "regex": "", "fieldType": 6, "dependentOn": "", "nested": null }, { "fieldName": "CustomerAddresses", "isPrimaryKey": 0, "dataType": 0, "isNullable": true, "isReadOnly": false, "isAutoGenerated": false, "isNeedOriginal": true, "maxLength": -1, "dateConversion": 0, "allowClientDefault": false, "range": "", "regex": "", "fieldType": 3, "dependentOn": "", "nested": null }]);
            super(opts, Customer);
        }
        findEntity(customerID: number): Customer {
            return this.findByPK(RIAPP.ArrayHelper.fromList(arguments));
        }
        toString() {
            return 'CustomerDb';
        }
        createReadCustomerQuery(args?: {
            includeNav?: boolean;
        }) {
            var query = this.createQuery('ReadCustomer');
            query.params = args;
            return query;
        }

     defineComplexProp_NameField(getFunc: (item: Customer) => string)
          {  this._defineCalculatedField('ComplexProp.Name', getFunc); }

   }
```

DbSet:

The DbSet class is derived from the *Collection* class so it supports all its methods and properties. But it is used to store items (*entities*) loaded from the data service.

It can be also filled directly with the data using its *fillItems* method.
The direct data loading is useful when you wish that the data will be present in the DbSet at the time when the web page is loaded. It reduces a number of data round trips to the server.

Note: *You can see how it is done in the GridDemo example (Models and Categories in the filter are loaded using this method).*

The DbSet also adds or overloads implementation of some methods and properties inherited from its base CollectionBase type:

DbSet's specific methods:

Property	Description
fillItems	Loads the DbSet with locally stored data. (*Useful to fill static data for lookups*)
acceptChanges	Accepts pending changes.
rejectChanges	Rejects pending changes.
deleteOnSubmit	Marks an entity for deletion on submitting updates.
createQuery	Creates an instance of the DataQuery (*by query's name*).
clearCache	Explicitly clears local cache.
_defineCalculatedField	Defines a calculated field. You need to provide a field name and a function which performs calculations.

DbSet's specific properties:

Property	is readonly	Description
dbContext	Yes	Returns its DbContext
dbSetName	Yes	Returns the DbSet's name (*as it was defined in the metadata*).
entityType	Yes	Returns a type of the entities which the DbSet contains.
isSubmitOnDelete	No	If it is set to true, then after an entity is submitted for delete, the changes will be submitted to the server automatically.
query	Yes	Returns the current query which is used for loading the DbSet
isHasChanges	Yes	Returns true if there are pending changes.
cacheSize	Yes	Returns the count of records currently stored in the local cache.

The DataService's *code generation feature* produces a script which contains all the DbSets which are exposed by the DataService. These DbSets are strongly typed and if a DbSet contains a calculated field (*its name is defined in the metadata*) then the strongly typed DbSet will have a strongly typed method to define this calculated field.

an example of a strongly typed DbSet with several methods to define different calculated fields:

```
export class RegistrDb extends RIAPP.MOD.db.DbSet<Registr, DbContext>
{
        constructor(dbContext: DbContext) {
           var self = this, opts: RIAPP.MOD.db.IDbSetConstuctorOptions = {
              dbContext: dbContext,
              dbSetInfo: { "fieldInfos": null, "enablePaging": true, "pageSize": 100, "dbSetName": "Registr" },
              childAssoc: ([]),
              parentAssoc: ([the associations info here...])
           }, utils = RIAPP.global.utils;
           opts.dbSetInfo.fieldInfos = ([the fields info here...]);
           super(opts, Registr);
        }
        findEntity(iNFOMONTH: string, iNFOTYPE: number, mCOD: string, rEGNUM: string): Registr {
           return this.findByPK(RIAPP.ArrayHelper.fromList(arguments));
        }
        toString() {
           return 'RegistrDb';
        }
        createReadRegistrQuery(args?: {
           d1: string;
           d2: string;
           cod: string;
        }) {
           var query = this.createQuery('ReadRegistr');
           query.params = args;
           return query;
        }
        createReadOldRegistrQuery(args?: {
           period: string;
        }) {
           var query = this.createQuery('ReadOldRegistr');
           query.params = args;
           return query;
        }
        defineLPUField(getFunc: (item: Registr) => string) { this._defineCalculatedField('LPU', getFunc); }
        defineDTYPEField(getFunc: (item: Registr) => string) { this._defineCalculatedField('DTYPE', getFunc); }
        defineFIOField(getFunc: (item: Registr) => string) { this._defineCalculatedField('FIO', getFunc); }
        defineS_OPLField(getFunc: (item: Registr) => number) { this._defineCalculatedField('S_OPL', getFunc); }
        defineSMOField(getFunc: (item: Registr) => string) { this._defineCalculatedField('SMO', getFunc); }
        defineADDRESSField(getFunc: (item: Registr) => string) { this._defineCalculatedField('ADDRESS', getFunc); }
        defineerrorsField(getFunc: (item: Registr) => any) { this._defineCalculatedField('errors', getFunc); }
}
```

Calculated fields (*if present*) must be defined after the DbContext's initialize method had been called, and before you load the data into the DbSet. Usually it is done in the Application's *onStartUp* method.

an example of several calculated fields definitions:

```
            this._dbContext.dbSets.Registr.defineerrorsField(function (item) {
                return self.dbContext.associations.getErrorToRegistr().getChildItems(item);
            });

            this._dbContext.dbSets.Registr.defineFIOField(function (item) {
                return item.FAM + " " + item.IM + " " + item.OT;

            });
            this._dbContext.dbSets.Registr.defineADDRESSField(function (item) {
                var res = '', ul = filter.cls.ulDict[item.UL];
                if (!!ul) {
                    res = ul.v;
                }
                if (!!item.DOM) {
                    res = res + ' дом ' + this.DOM;
                }
                if (!!item.KOR) {
                    res = res + ' кор. ' + this.KOR;
                }
                if (!!item.STR) {
                    res = res + ' стр. ' + this.STR;
                }
                if (!!item.KV) {
                    res = res + ' кв. ' + this.KV;
                }
                return res;

            });
            this._dbContext.dbSets.Registr.defineDTYPEField(function (item) {
                var res = filter.cls.dtypeRDict[item.D_TYPE_R];
                return !!res ? res.v : this.D_TYPE_R;
            });
            this._dbContext.dbSets.Registr.defineLPUField(function (item) {
                var res = filter.cls.lpuDict[item.MCOD];
                return !!res ? res.v : null;
            });
            this._dbContext.dbSets.Registr.defineS_OPLField(function (item) {
                var errs = this.errors, sall = item.S_ALL;
                if (!errs)
                    return sall;
                errs = errs.filter(function (e) {
                    return e.SFNUM != '?';
                });
                var udl = 0;
                errs.forEach(function (e) {
                    udl += e.SUM_UDL;
                });
                return sall - udl;
            });
            this._dbContext.dbSets.Registr.defineSMOField(function (item) {
                var res = filter.cls.smoDict[item.Q];
                return !!res ? res.v : item.Q;
```

```
        });
```

Entities:

The *CollectionItem* is a base class for the derived entity classes.
Basicaly, an entity is a collection item which is specific for the DbSet class. The concrete implementations of the entity types have all the properties defined in the metadata for the DbSet.
The entities can also have navigation properties added by the associations.
You can see for example, in the Demo application- in the file RIAppDemo.BLL\Metadata\MainDemo.xml
This is a file in which every DbSet used by the DataService is defined in xml format. It simplifies editing of the metadata, because xml is more readable format than a definition of this information in code. This data is kept on the server and some (*not all*) of this information is available on the client and it is also used to generate strongly typed classes.
The DataService loads this xml file in the *GetMetadata* method:

```
 protected override Metadata GetMetadata(bool isDraft)
 {
        if (isDraft)
        {
           //returns raw (uncorrected) programmatically generated metadata from LinqToSQL classes
           return base.GetMetadata(true);
        }
        else
        {
           //first the uncorrected metadata was saved into xml file and then edited
           return
Metadata.FromXML(ResourceHelper.GetResourceString("RIAppDemo.BLL.Metadata.MainDemo.xml"));
        }
 }
```

Note: *using this information the DataService's GetTypeScript method generates entities and strongly typed DbSets classes.*

Every entity has an *_aspect* property which exposes strongly typed instance of an EntityAspect class. The aspect is used to keep separately methods and properties which are used for specific purposes - like *beginEdit*, *endEdit* and etc.

an example of an entity definition:

```
export class CustomerAddress extends collMOD.CollectionItem<dbMOD.EntityAspect<CustomerAddress,
DbContext>> implements dbMOD.IEntityItem {

      constructor(aspect: dbMOD.EntityAspect<CustomerAddress, DbContext>) {
         super(aspect);

      }
      toString() {
         return 'CustomerAddress';
      }
      get CustomerID(): number { return this._aspect._getFieldVal('CustomerID'); }
      set CustomerID(v: number) { this._aspect._setFieldVal('CustomerID', v); }
      get AddressID(): number { return this._aspect._getFieldVal('AddressID'); }
      set AddressID(v: number) { this._aspect._setFieldVal('AddressID', v); }
      get AddressType(): string { return this._aspect._getFieldVal('AddressType'); }
      set AddressType(v: string) { this._aspect._setFieldVal('AddressType', v); }
      get rowguid(): string { return this._aspect._getFieldVal('rowguid'); }
      get ModifiedDate(): Date { return this._aspect._getFieldVal('ModifiedDate'); }
```

```
        get Customer(): Customer { return this._aspect._getNavFieldVal('Customer'); }
        set Customer(v: Customer) { this._aspect._setNavFieldVal('Customer', v); }
        get Address(): Address { return this._aspect._getNavFieldVal('Address'); }
        set Address(v: Address) { this._aspect._setNavFieldVal('Address', v); }
        get AddressInfo(): AddressInfo { return this._aspect._getNavFieldVal('AddressInfo'); }
        set AddressInfo(v: AddressInfo) { this._aspect._setNavFieldVal('AddressInfo', v); }
}
```

an example of a DbSet's schema definition (in XAML):

```
 <data:DbSetInfo dbSetName="Customer" isTrackChanges="True" insertDataMethod="Insert{0}"
updateDataMethod="Update{0}" deleteDataMethod="Delete{0}" refreshDataMethod="Refresh{0}"
validateDataMethod="Validate{0}" enablePaging="True" pageSize="25" EntityType="{x:Type dal:Customer}">
                <data:DbSetInfo.fieldInfos>
                    <data:Field fieldName="CustomerID" dataType="Integer" maxLength="4" isNullable="False"
isAutoGenerated="True" isReadOnly="True" isPrimaryKey="1" />
                    <data:Field fieldName="NameStyle" dataType="Bool" maxLength="1" isNullable="False"  />
                    <data:Field fieldName="Title" dataType="String" maxLength="8" />
                    <data:Field fieldName="Suffix" dataType="String" maxLength="10" />
                    <data:Field fieldName="CompanyName" dataType="String" maxLength="128" />
                    <data:Field fieldName="SalesPerson" dataType="String" maxLength="256" />
                    <data:Field fieldName="PasswordHash" dataType="String" maxLength="128" isNullable="False"
isAutoGenerated="True" isReadOnly="True" />
                    <data:Field fieldName="PasswordSalt" dataType="String" maxLength="10" isNullable="False"
isAutoGenerated="True" isReadOnly="True" />
                    <data:Field fieldName="rowguid" dataType="Guid" maxLength="16" isNullable="False"
isAutoGenerated="True" isReadOnly="True" fieldType="RowTimeStamp" />
                    <data:Field fieldName="ModifiedDate" dataType="DateTime" maxLength="8" isNullable="False"
isAutoGenerated="True" isReadOnly="True" />
                    <data:Field fieldName="ComplexProp" fieldType="Object" >
                        <data:Field.nested>
                            <data:Field fieldName="FirstName" dataType="String" maxLength="50" isNullable="False"
/>
                            <data:Field fieldName="MiddleName" dataType="String" maxLength="50" />
                            <data:Field fieldName="LastName" dataType="String" maxLength="50" isNullable="False"
/>
                            <data:Field fieldName="Name" dataType="String" fieldType="Calculated"
dependentOn="ComplexProp.FirstName,ComplexProp.MiddleName,ComplexProp.LastName" />
                            <data:Field fieldName="ComplexProp" fieldType="Object" >
                                <data:Field.nested>
                                    <data:Field fieldName="EmailAddress" dataType="String" maxLength="50"
regex="^[_a-z0-9-]+(\.[_a-z0-9-]+)*@[a-z0-9-]+(\.[a-z0-9-]+)*(\.[a-z]{2,4})$" />
                                    <data:Field fieldName="Phone" dataType="String" maxLength="25" />
                                </data:Field.nested>
                            </data:Field>
                        </data:Field.nested>
                    </data:Field>
                </data:DbSetInfo.fieldInfos>
 </data:DbSetInfo>
```

an example of an association definition (in XAML):

```
<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>
```

The EntityAspect type specific methods (besides inherited from the ItemAspect):

Property	Description
deleteOnSubmit	Marks the item for deletion and the item's status (*changetType*) is set to deleted.

deleteItem	The same as deleteOnSubmit. The methods are semantically equivalent.
refresh	Invokes the entity's data refresh. The method is asynchronous, it returns a promise.
acceptChanges	Accepts the changes.
rejectChanges	Reject the changes and restores the values to the original.

The EntityAspect type specific properties (*besides inherited from the ItemAspect*):

Property	is readonly	Description
dbSetName	Yes	Returns the name of the parent DbSet.
status	Yes	Returns the status of the entity. *export enum STATUS { NONE= 0, ADDED= 1, UPDATED= 2, DELETED= 3 }*
serverTimezone	Yes	Returns the time zone on the server.
isRefreshing	Yes	Returns true if the entity is refreshing its values.
isCached	Yes	Returns true if the entity is cached locally. Generally, it is used internally.
isHasChanges	Yes	Returns true if the values are modified and not submitted to the server.
isCanSubmit	Yes	Always returns true.
isNew	Yes	Returns true if the entity was added, but not submitted to the server.
isDeleted	Yes	Returns true if the entity is deleted but not submitted to the server.
dbSet	Yes	Returns the parent DbSet.

When a new entity is added, it exists in the editing state. To commit any modifications the *endEdit* method should be called. To discard the modifications and undo adding the entity the *cancelEdit* method should be called.

An example of adding a new entity and setting its field values:

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

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

//commit the changes on the client using the _aspect
Item._aspect.endEdit();

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

If you assign a new value to a field, and the entity is not in editing state, then its *beginEdit* method is called automatically.
On every call to the beginEdit or endEdit method, if it is completed successfully, there will be triggered 'begin_edit' or 'end_edit' events. In order to prevent triggering those events, for example, when you want to update a large number of DbSet's items, you can use the DbSet’s *isUpdating* property.

an example of updating DbSet's items without triggering editing events:

```
 //prevent implicit calls to beginEdit method
 self._dbSet.isUpdating = true; //mark the update started
  try
  {
         self._dbSet.items.forEach(function(item){
              item.someField= 1;   //set the entity's fields
          });
  }
  finally
  {
        self._dbSet.isUpdating = false; //mark the update ended
  }
```

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

Calculated fields:

The calculated fields calculate their values on the client side in a function. They must not have circular references. Calculated fields are read only. They can depend on other fields (*calculated or not*), and they are automatically refreshed (*recalculated*) when those fields are changed.

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

an example of a calculated field declaration in the metadata (in XAML):

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

Client fields:

Client fields are used only on the client. They don't exist on the server side's entity. So their changes are not submitted to the server and they don't take values from the server (*initially they have null values*).

They are declared on the server in the DbSet's schema.

an example of two client fields declaration in the metadata (in XAML):

```
<data:FieldInfo fieldName="Address" dataType="None" fieldType="ClientOnly"  />
<data:FieldInfo fieldName="Customer" dataType="None" fieldType="ClientOnly"  />
```

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

Server side calculated fields:

Server side calculated fields are used to provide a property on the entity which is calculated on the server and is used on the client for information purposes. Their changes are not submitted to the server (*even if you change them on the client their updates will be ignored*). They can be used for different purposes, when the fields values can be only obtained on the server side (*or if it easier to do*). You can even set real database

fields on the entities to a *ServerCalculated* field type. In that case the fields will function on the client side like *ClientOnly* fields. Only their values will be initially set from the server side.
They are declared on the server in the DbSet's schema.

an example of a server side calculated field declaration in the metadata (in XAML):

```
<data:FieldInfo fieldName="AddressCount" dataType="Integer" fieldType="ServerCalculated"  />
```

Note: *You can see the Customer entity in the Demo application. There was added an AddressCount server side calculated field (for testing purposes). It is used in the Master-detail demo web page.*

Navigation fields:

An entity can also have navigation properties. They are based on foreign key relationship between the entities. These foreign key relationship in the framework are encapsulated in the association type. The relationship (*parent - child*) are defined in these associations in the metadata definition. There can also exist many to many relationships which are defined by using two *parent - child* relationships.
The association definition can set (*optionally*) the names of the navigation fields, and if they were set, then the association definition adds them to the corresponding, generated for the client side, entities. A parent entity can get (*using its navigation field*) an array of child entities and a child entity can get its parent entity.

If you want to insert into the database a parent entity along with a child entity in one transaction you can use navigation fields for that purpose.

Ordinarily (*without using navigation fields*), you insert a parent entity, then submit the changes to the server to obtain the primary key for the entity (*they are usually autogenerated on the server*), then assign this primary key values to the child entities' foreign key fields and then submit them to the server in a second batch.

But, by using navigation fields, you can assign the parent entity directly to the *childToParent* type navigation field. On submit, the data service fixes this relationship automatically, and the submit is performed in one database transaction.

an example of assigning parent entity to the navigation field:

```
var cust = this.currentCustomer;
var ca = this.custAdressView.addNew(); //create a 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 can also be new and has no Primary key - the data service fixes this on submit
ca._aspect.endEdit();

//here we can submit changes to the server with dbContext's submitChanges method
//or do more changes on the client, and then submit them in one transaction
```

Warning: *Don't declare navigation fields to a DbSet's fields in the matadata explicitly. When you define an association in the metadata, then the navigation fields will be added automatically to the entities in the client side domain model (generated by dataservice's GetTypeScript method).*

Complex type fields:

An entity can expose a field that has an object type. For example, you can aggregate all address related properties into one field - Address. So, each address property is accessed through the complex property, like: customer.Address.Street.
Complex properties can also have their calculated and client only fields. But, they can not have navigation properties.

They are declared on the server in the DbSet's schema.
These fields (*when declared in the metadata*) have a nested property which is the container for the object's properties.

an example of an object field declaration in the metadata (in XAML):

```
 <data:Field fieldName="ComplexProp" fieldType="Object" >
                        <data:Field.nested>
                           <data:Field fieldName="FirstName" dataType="String" maxLength="50" isNullable="False"  />
                           <data:Field fieldName="MiddleName" dataType="String" maxLength="50" />
                           <data:Field fieldName="LastName" dataType="String" maxLength="50" isNullable="False"  />
                           <data:Field fieldName="Name" dataType="String" fieldType="Calculated"
dependentOn="ComplexProp.FirstName,ComplexProp.MiddleName,ComplexProp.LastName" />
                           <data:Field fieldName="ComplexProp" fieldType="Object" >
                              <data:Field.nested>
                                 <data:Field fieldName="EmailAddress" dataType="String" maxLength="50" regex="^[_a-z0-
9-]+(\.[_a-z0-9-]+)*@[a-z0-9-]+(\.[a-z0-9-]+)*(\.[a-z]{2,4})$" />
                                 <data:Field fieldName="Phone" dataType="String" maxLength="25" />
                              </data:Field.nested>
                           </data:Field>
                        </data:Field.nested>
</data:Field>
```

Entity and fields validations:

An entity validation process is done on the client and is also done on the server.
The client side validation is triggered when a new value is assigned to a field and also when an entity ends editing (*when the endEdit method is invoked implicitly or explicitly*).
For most cases automatic validation is usually enough. The automatic validation is based on the checks and constraints defined in the DbSet's schema. The DbSet's schema can include constraints for nullability (*isNullable*), maximum length (*maxLength*), field writability (*isReadOnly*) , type checking is based on the field's data type (*string, number, bool, date*) and checks defined using a range and a regex expression.

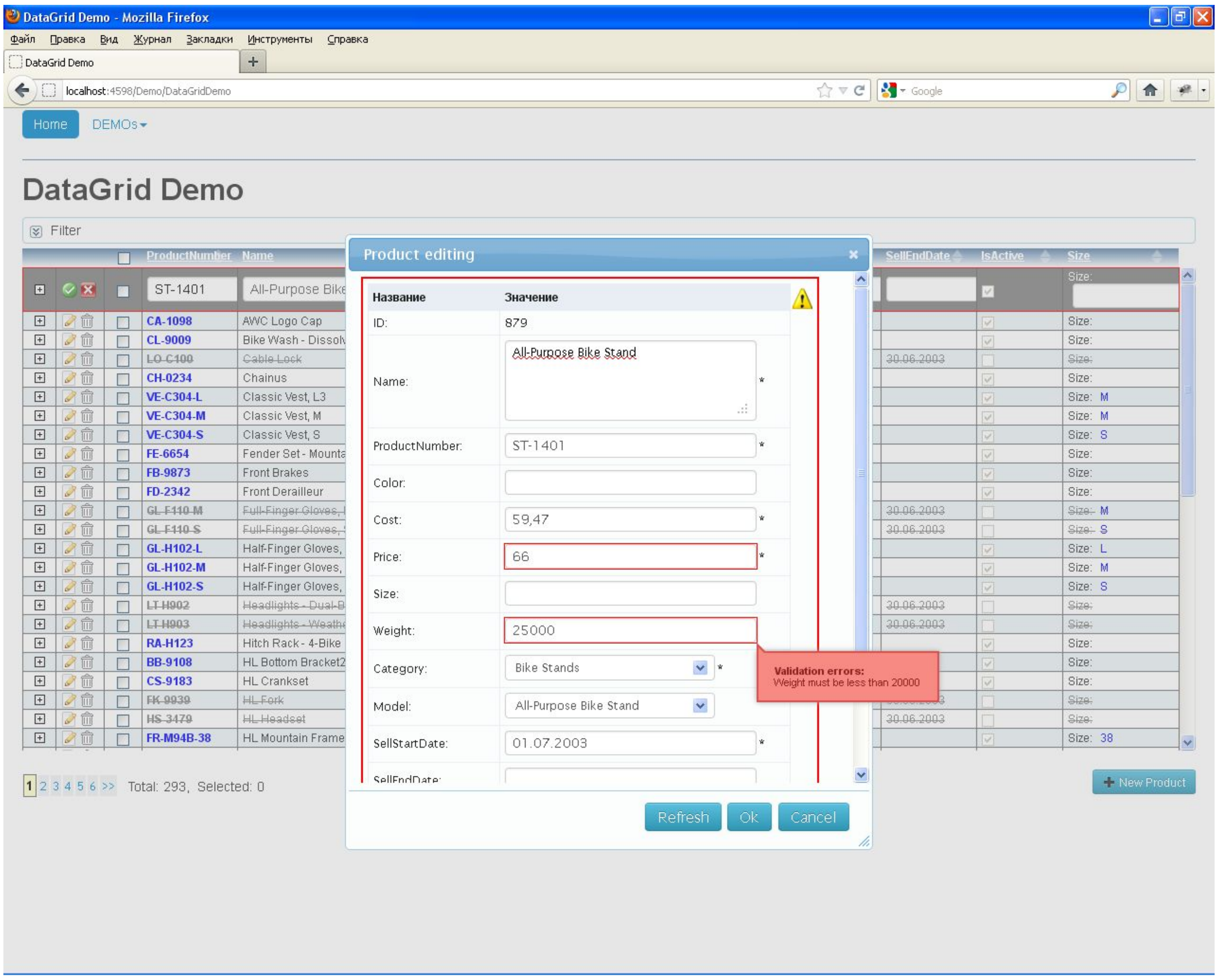
If it is not enough, then you can use a custom validation.
The types derived from the Collection (*List, Dictionary, DbSet*) have a validate event, which is used for a custom client side validation.
An event handler can check custom validation conditions and then can add errors to the error property if it is not validated.

an example of the custom client side data validation:

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

On unsuccessful client side validation, the errors are added to the DbSet's internal array of errors. The entity remains in the editing state and can not end editing while the errors are present.
In order to end editing, the correct values should be assigned which pass the validation, or otherwise the changes must be canceled with the *cancelEdit* method.

Databindings handle validation errors, and update element view targets for their display to the user.
If an entity has errors, then a UI control (*DataGrid, DataForm*) will display error notifications (*red borders and tooltips on mouse hovering*) next to not validated fields and the data form will have a validation summary at the top right corner (*if you hover a mouse over it, a tooltip will be shown*).


DataGrid Demo - Mozilla Firefox
Файл Правка Вид Журнал Закладки Инструменты Справка
DataGrid Demo
localhost:4598/Demo/DataGridDemo
Google
Home
DEMOs
Filter
Refresh
Ok
Cancel
New Product

DataGrid Demo

ProductNumber	Name	SellEndDate	IsActive	Size
ST-1401	All-Purpose Bike		✓	Size:
CA-1098	AWC Logo Cap		✓	Size:
CL-9009	Bike Wash - Dissolv		✓	Size:
~~LO-C100~~	~~Cable Lock~~	~~30.06.2003~~		~~Size:~~
CH-0234	Chainus		✓	Size:
VE-C304-L	Classic Vest, L3		✓	Size: M
VE-C304-M	Classic Vest, M		✓	Size: M
VE-C304-S	Classic Vest, S		✓	Size: S
FE-6654	Fender Set - Mounta		✓	Size:
FB-9873	Front Brakes		✓	Size:
FD-2342	Front Derailleur		✓	Size:
~~GL-F110-M~~	~~Full-Finger Gloves,~~	~~30.06.2003~~		~~Size:~~ M
~~GL-F110-S~~	~~Full-Finger Gloves,~~	~~30.06.2003~~		~~Size:~~ S
GL-H102-L	Half-Finger Gloves,		✓	Size: L
GL-H102-M	Half-Finger Gloves,		✓	Size: M
GL-H102-S	Half-Finger Gloves,		✓	Size: S
~~LT-H902~~	~~Headlights - Dual-B~~	~~30.06.2003~~		~~Size:~~
~~LT-H903~~	~~Headlights - Weathe~~	~~30.06.2003~~		~~Size:~~
RA-H123	Hitch Rack - 4-Bike		✓	Size:
BB-9108	HL Bottom Bracket2		✓	Size:
CS-9183	HL Crankset		✓	Size:
~~FK-9939~~	~~HL Fork~~	[illegible]		~~Size:~~
~~HS-3479~~	~~HL Headset~~	~~30.06.2003~~		~~Size:~~
FR-M94B-38	HL Mountain Frame		✓	Size: 38

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

Product editing

Название	Значение
ID:	879
Name:	All-Purpose Bike Stand *
ProductNumber:	ST-1401 *
Color:	
Cost:	59,47 *
Price:	66 *
Size:	
Weight:	25000
Category:	Bike Stands *
Model:	All-Purpose Bike Stand
SellStartDate:	01.07.2003 *
SellEndDate:	

Validation errors:
Weight must be less than 20000

The errors are cleared when correct values are assigned to the fields or the changes are discarded by using the *cancelEdit* method.

As it was noted above, the validation is done on the client and the server side, and for automatic validation you need to define checks and constraints in the DbSet's schema.

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

DataView:

The DataView – is a descendant of the collection type. It is used to wrap an existing collection, which we want to filter or sort for the display. You can use it to expose only a partial set of the data of the underlying collection.

Note: *For example, you can load all child entities for already loaded parent entities. The child dbSet in the relationship is wrapped with the DataView, and to display only a subset of child entities you can just change the filter condition on the DataView.*
The child items relative to the current parent item will be filtered out of the DbSet's data.

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

By providing a subset of the already prefiltered data through *fn_itemsProvider* and also by filtering the data with *fn_filter* function (*they are not mutually exclusive and can be used together*). The workflow of the data processing is:

If you don't provide *fn_itemsProvider* then the data is taken from the *dataSource* directly, but if you provide *fn_itemsProvider* then the data is taken by executing *fn_itemsProvider.* After that, on the obtained data, if you provide a *fn_filter* then the data is filtered using this function, and then if you provide a *fn_sort* then the data is also sorted.

an example of a DataView initialization:

```
//it filters addresses related to the current customer
 this._addressesView = new MOD.db.DataView<DEMODB.Address>(
                {
                    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 [];
                        var custAdrs = self._currentCustomer.CustomerAddresses;
                        return custAdrs.map(function (m) {
                            return m.Address;
                        }).filter(function (address) {
                            return !!address;
                        });
                    }
                });
```

Another example of a DataView initialization:

```
this._addressInfosView = new MOD.db.DataView<DEMODB.AddressInfo>(
                {
                    dataSource: this._addressInfosDb,
                    fn_sort: function (a: DEMODB.AddressInfo, b: DEMODB.AddressInfo)
                        { return a.AddressID - b.AddressID; },
                    fn_filter: function (item: DEMODB.AddressInfo) {
                        return !item.CustomerAddresses.some(function (CustAdr) {
                            return self._currentCustomer === CustAdr.Customer;
                        });
                    }
                });
```

The only mandatory option's property is the dataSource - which is an instance of the collection which will be wrapped up with the DataView. You can omit the sorting and filtering functions if you don't need to filter or sort the data.

When you want the data in the DataView to be refreshed (*reread, refiltered and resorted*) you can call the DataView's *refresh* method, as in the example:

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

The DataView's specific methods:

Property	Description
refresh	Refilters and resorts the data in the DataView
clear	Overriden collection's method. Clears only the dataview's data, the real data which is in the wrapped DbSet is not touched.
addNew	Overriden collection's method. Adds a new entity (*creates new entity*) to the underlying DbSet.
appendItems	Overriden collection's method. Adds an array of existing in the DbSet entities to the view. The items are not filtered before adding to the view (*they are just appended*).

The DataView's specific properties:

Property	is readonly	Description
fn_filter	No	the filter function
fn_sort	No	the sort function
fn_itemsProvider	No	the function to provide already prefiltered items
isPagingEnabled	No	Overriden property. If it was set to true then the view splits its data in pages.
permissions	Yes	Returns the wrapped collections's permissions property value.

the Association and the ChildDataView:

The *ChildDataView* class is a descendant of the *DataView* class. It simplifies the task where you need to display details records of a parent record in a master- detail relationship. It uses an association to obtain the child entities from the parent entity.
The association stores and updates a map of the parent-child relationship based on the foreign keys relationship.
The definition of the relationship (*the association*) is done in the metadata on the server side.

an example of the metadata for a DbSet:

```
<data:Metadata x:Key="FolderBrowser">
```

```
        <data:Metadata.DbSets>
            <data:DbSetInfo dbSetName="FileSystemObject" enablePaging="False" EntityType="{x:Type
models:FolderModel}" deleteDataMethod="Delete{0}">
                <data:DbSetInfo.fieldInfos>
                    <data:Field fieldName="Key" dataType="String" maxLength="255" isNullable="False"
isAutoGenerated="True" isReadOnly="True" isPrimaryKey="1" />
                    <data:Field fieldName="ParentKey" dataType="String" maxLength="255" isReadOnly="True" />
                    <data:Field fieldName="Name" dataType="String" maxLength="255" isNullable="False"
isReadOnly="True" />
                    <data:Field fieldName="Level" dataType="Integer" isNullable="False" isReadOnly="True" />
                    <data:Field fieldName="HasSubDirs" dataType="Bool" isNullable="False" isReadOnly="True" />
                    <data:Field fieldName="IsFolder" dataType="Bool" isNullable="False" isReadOnly="True" />
                    <data:Field fieldName="fullPath" dataType="String" fieldType="Calculated" />
                </data:DbSetInfo.fieldInfos>
            </data:DbSetInfo>
        </data:Metadata.DbSets>

        <data:Metadata.Associations>
            <data:Association name="ChildToParent" parentDbSetName="FileSystemObject"
childDbSetName="FileSystemObject" 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 you define an association in the metadata, you define which field in a child entity relates to the key field in a parent entity. You can also give names for navigation properties (*parentToChildrenName and childToParentName*). These navigation properties, with the names you defined, will be added to the generated entity classes.

an example of getting an association and a ChildDataView instantiation:

```
 var custAssoc = self.dbContext.associations.getCustAddrToCustomer();

 //the view to filter CustomerAddresses related to the current customer only
 this._custAdressView = new MOD.db.ChildDataView<DEMODB.CustomerAddress>(
   {
          association: custAssoc,
          fn_sort: function (a: DEMODB.CustomerAddress, b: DEMODB.CustomerAddress) {
                   return a.AddressID - b.AddressID;
                   }
   });
```

When you want to display details for a parent entity, you should assign the ChildDataView's *parentItem* property. It triggers refreshing of the view with the new data. After assigning the *parentItem* property the view will contain only details (*child entities*) for the parent entity.

```
_onCurrentChanged() {
         //set a new parent item to the ChildDataView
          this._custAdressView.parentItem = this._dbSet.currentItem;

          this.raisePropertyChanged('currentItem');
 }
```

Association's methods:

Property	Description
getChildItems	Accepts an entity and returns an array of the child (*details*)

	entities
getParentItem	Accepts an entity and returns the parent (*master*) entity

Association's properties:

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

Working with the data on the server side

4.1 *The Data service*

The data service application is implemented in C# language and requires Microsoft Net Framework 4.5 (*or above*) to be installed on the server computer.

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

```
public interface IDomainService: IDisposable
  {
     //typescript strongly typed implementation of entities, DbSet and DbContext in the text form
     string ServiceGetTypeScript(string comment=null);
     string ServiceGetXAML();
     string ServiceGetCSharp();

    //information about permissions to execute service operations for the client
     Permissions ServiceGetPermissions();
     //information about service methods, DbSets and their fields information
     MetadataResult ServiceGetMetadata();
     QueryResponse ServiceGetData(QueryRequest request);
     ChangeSet ServiceApplyChangeSet(ChangeSet changeSet);
     RefreshInfo ServiceRefreshRow(RefreshInfo rowInfo);
     InvokeResponse ServiceInvokeMethod(InvokeRequest invokeInfo);
  }
```

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

The data service is usually hosted in the ASP.NET MVC framework due to the web framework's convenient design for the invocation of server side methods through Ajax calls.
The server part of the framework consists of 6 projects:

1) RIAPP.DataService - the main implementation of the data service. It implements *RIAPP.DataService.BaseDomainService* class.
2) RIAPP.DataService.EF - implements *RIAPP.DataService.EF.EFDomainService<TDB>* class derived from *BaseDomainService* and which can be used to work with Microsoft Entity Framework 4 domain models.
3) RIAPP.DataService.EF2 - implements *RIAPP.DataService.EF2.EFDomainService<TDB>* class derived from *BaseDomainService* and which can be used to work with Microsoft Entity Framework 6 domain models which use DbContext instead of ObjectContext.
4) RIAPP.DataService.LinqSql - implements *RIAPP.DataService.LinqSql.LinqForSqlDomainService<TDB>* class derived from *BaseDomainService* and which can be used with Microsoft Linq For SQL domain models.
5) RIAPP.DataService.Mvc - implements classes to integrate the dataservice with ASP.NET MVC. Namely it has asp.net MVC controller which exposes the DataService's methods.
6) RIAPP.DataService.WebApi - implements classes to integrate the dataservice with ASP.NET WebApi. It defines base ApiController descendant which exposes the DataService's methods.

RIAPP.DataService.Mvc assembly contains a descendant of *System.Web.Mvc.Controller* class: *DataServiceController*, which incapsulates service methods invocations inside an ASP.NET MVC controller.

an implementation of the DataServiceController:

```
 public abstract class DataServiceController<T> : Controller
       where T : BaseDomainService
   {
      private ISerializer _serializer;

      #region PRIVATE METHODS
      private ActionResult _GetTypeScript()
      {
         string comment = string.Format("\tGenerated from: {0} on {1:yyyy-MM-dd HH:mm} at {1:HH:mm}\r\n\tDon't
make manual changes here, because they will be lost when this db interface will be regenerated!",
this.ControllerContext.HttpContext.Request.RawUrl, DateTime.Now);
         var info = this.DomainService.ServiceGetTypeScript(comment);
         var res = new ContentResult();
         res.ContentEncoding = System.Text.Encoding.UTF8;
         res.ContentType = System.Net.Mime.MediaTypeNames.Text.Plain;
         res.Content = info;
         return res;
      }

      private ActionResult _GetXAML()
      {
         var info = this.DomainService.ServiceGetXAML();
         var res = new ContentResult();
         res.ContentEncoding = System.Text.Encoding.UTF8;
         res.ContentType = System.Net.Mime.MediaTypeNames.Text.Plain;
         res.Content = info;
         return res;
      }
```

```
private ActionResult _GetCSharp()
{
    var info = this.DomainService.ServiceGetCSharp();
    var res = new ContentResult();
    res.ContentEncoding = System.Text.Encoding.UTF8;
    res.ContentType = System.Net.Mime.MediaTypeNames.Text.Plain;
    res.Content = info;
    return res;
}
#endregion

public DataServiceController()
{
    this._serializer = new Serializer();
    this._DomainService = new Lazy<IDomainService>(() => this.CreateDomainService(),true);
}

protected virtual IDomainService CreateDomainService()
{
    ServiceArgs args = new ServiceArgs(this._serializer, this.User);
    var service = (IDomainService)Activator.CreateInstance(typeof(T), args);
    return service;
}

private Lazy<IDomainService> _DomainService;

[ChildActionOnly]
public string PermissionsInfo()
{
    var info = this.DomainService.ServiceGetPermissions();
    return this.Serializer.Serialize(info);
}

[ActionName("code")]
[HttpGet]
public ActionResult GetCode(string lang)
{
    if (lang != null)
    {
        switch (lang.ToLowerInvariant())
        {
            case "ts":
            case "typescript":
                return this._GetTypeScript();
            case "xaml":
                return this._GetXAML();
            case "csharp":
                return this._GetCSharp();
            default:
                throw new Exception(string.Format("Unknown type argument: {0}", lang));
        }
    }
    else
        return this._GetTypeScript();
}

[ActionName("permissions")]
[HttpGet]
public ActionResult GetPermissions()
{
    var res = this.DomainService.ServiceGetPermissions();
    return Json(res, JsonRequestBehavior.AllowGet);
}

[ActionName("query")]
[HttpPost]
public ActionResult PerformQuery([SericeParamsBinder] QueryRequest request)
{
    return new IncrementalResult(this.DomainService.ServiceGetData(request), this.Serializer);
}
```

```
        [ActionName("save")]
        [HttpPost]
        public ActionResult Save([SericeParamsBinder] ChangeSet changeSet)
        {
            var res = this.DomainService.ServiceApplyChangeSet(changeSet);
            return Json(res);
        }

        [ActionName("refresh")]
        [HttpPost]
        public ActionResult Refresh([SericeParamsBinder] RefreshInfo refreshInfo)
        {
            var res = this.DomainService.ServiceRefreshRow(refreshInfo);
            return Json(res);
        }

        [ActionName("invoke")]
        [HttpPost]
        public ActionResult Invoke([SericeParamsBinder] InvokeRequest invokeInfo)
        {
            var res = this.DomainService.ServiceInvokeMethod(invokeInfo);
            return Json(res);
        }

        protected IDomainService DomainService
        {
            get
            {
                return this._DomainService.Value;
            }
        }

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

        public ISerializer Serializer
        {
            get { return this._serializer; }
        }

        protected override void Dispose(bool disposing)
        {
            if (disposing && this._DomainService.IsValueCreated)
            {
                this._DomainService.Value.Dispose();
            }
            this._DomainService = null;
            this._serializer = null;
            base.Dispose(disposing);
        }
    }
```

The base DataService class (*BaseDomainService*) is implemented in the *RIAPP.DataService* assembly. It is an abstract class, it has two abstract methods *GetMetadata* and *ExecuteChangeSet* which must be implemented in the descendant.

For example, in the *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.

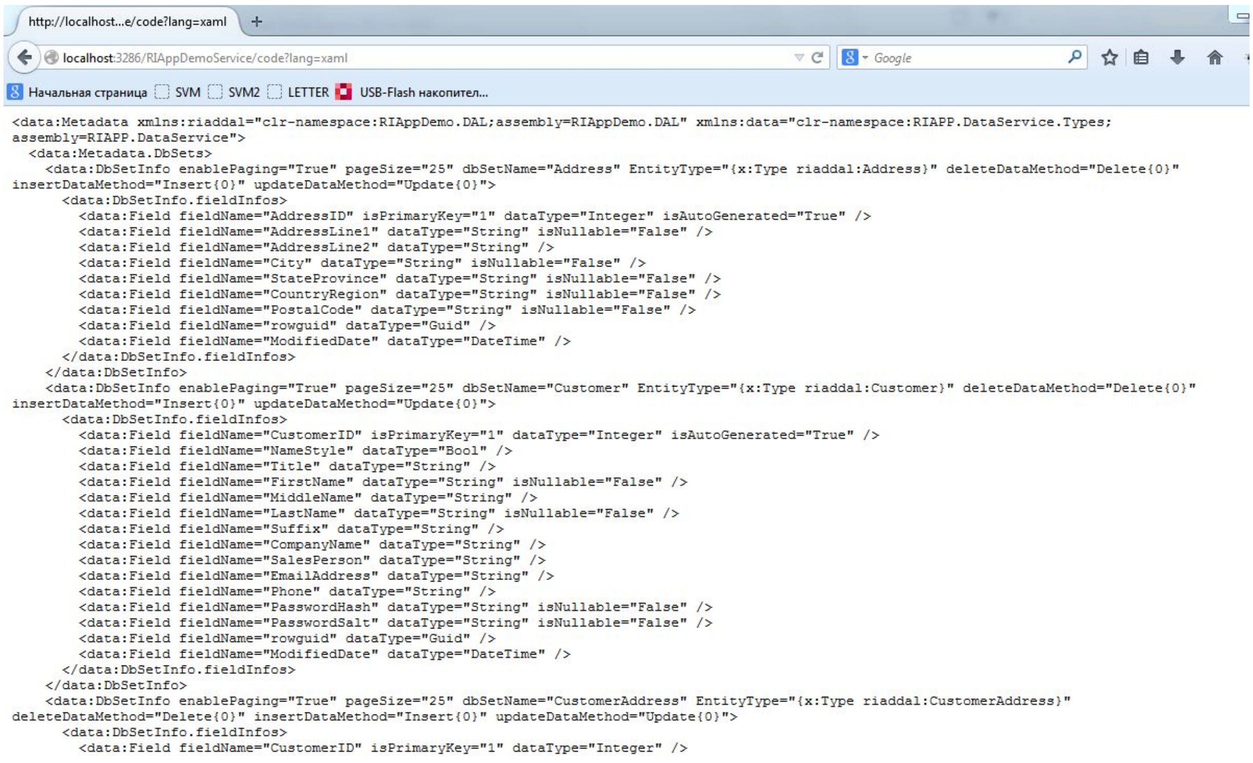
```
 protected override async Task ExecuteChangeSet()
 {
           using (TransactionScope transScope = new TransactionScope(TransactionScopeOption.RequiresNew,
              new TransactionOptions { IsolationLevel = IsolationLevel.ReadCommitted, Timeout =
TimeSpan.FromMinutes(1.0) }, TransactionScopeAsyncFlowOption.Enabled))
            {
                await this.DB.SaveChangesAsync().ConfigureAwait(false);

                transScope.Complete();
            }
            await this.AfterExecuteChangeSet();
  }
```

The *GetMetadata* method should return *RIAPP.DataService.Types.Metadata* class instance. Specialized data services classes which work with Microsoft Linq for SQL and Microsoft Entity Framework (*defined in RIAPP.DataService.Linq and RIAPP.DataService.EF respectively*) override this method. They take the underlying *System.Data.Linq.DataContext* or *System.Data.Objects.ObjectContext* instance respectively and use the metadata information from it. But they produce a raw (*draft*) metadata, which should be edited before exposing it to clients.
The editing of the metadata is best done when it is in a more readable format - for example, XML.

an implementation of the ServiceGetXAML method in the BaseDomainService:

```
  public string ServiceGetXAML(bool isDraft = true)
   {
            if (!this.IsCodeGenEnabled)
                throw new InvalidOperationException(string.Format(ErrorStrings.ERR_CODEGEN_DISABLED,
MethodInfo.GetCurrentMethod().Name, this.GetType().Name));
            Metadata metadata = null;
            MetadataHelper.ExecuteOnSTA((object state) =>
            {
                metadata = this.GetMetadata(isDraft);
            }, this);
            return metadata.ToXML();
  }
```

You can get a XML representation of the metadata. For that you can navigate to *CodeGen* url in the browser like in the example
http://YOURSERVER/RIAppDemoService/code?lang=xaml
Then you can copy and paste the XML.


http://localhost...e/code?lang=xaml
localhost:3286/RIAppDemoService/code?lang=xaml
Google
Начальная страница
SVM
SVM2
LETTER
USB-Flash накопител...

```xml
<data:Metadata xmlns:riaddal="clr-namespace:RIAppDemo.DAL;assembly=RIAppDemo.DAL" xmlns:data="clr-namespace:RIAPP.DataService.Types;
assembly=RIAPP.DataService">
  <data:Metadata.DbSets>
    <data:DbSetInfo enablePaging="True" pageSize="25" dbSetName="Address" EntityType="{x:Type riaddal:Address}" deleteDataMethod="Delete{0}"
insertDataMethod="Insert{0}" updateDataMethod="Update{0}">
      <data:DbSetInfo.fieldInfos>
        <data:Field fieldName="AddressID" isPrimaryKey="1" dataType="Integer" isAutoGenerated="True" />
        <data:Field fieldName="AddressLine1" dataType="String" isNullable="False" />
        <data:Field fieldName="AddressLine2" dataType="String" />
        <data:Field fieldName="City" dataType="String" isNullable="False" />
        <data:Field fieldName="StateProvince" dataType="String" isNullable="False" />
        <data:Field fieldName="CountryRegion" dataType="String" isNullable="False" />
        <data:Field fieldName="PostalCode" dataType="String" isNullable="False" />
        <data:Field fieldName="rowguid" dataType="Guid" />
        <data:Field fieldName="ModifiedDate" dataType="DateTime" />
      </data:DbSetInfo.fieldInfos>
    </data:DbSetInfo>
    <data:DbSetInfo enablePaging="True" pageSize="25" dbSetName="Customer" EntityType="{x:Type riaddal:Customer}" deleteDataMethod="Delete{0}"
insertDataMethod="Insert{0}" updateDataMethod="Update{0}">
      <data:DbSetInfo.fieldInfos>
        <data:Field fieldName="CustomerID" isPrimaryKey="1" dataType="Integer" isAutoGenerated="True" />
        <data:Field fieldName="NameStyle" dataType="Bool" />
        <data:Field fieldName="Title" dataType="String" />
        <data:Field fieldName="FirstName" dataType="String" isNullable="False" />
        <data:Field fieldName="MiddleName" dataType="String" />
        <data:Field fieldName="LastName" dataType="String" isNullable="False" />
        <data:Field fieldName="Suffix" dataType="String" />
        <data:Field fieldName="CompanyName" dataType="String" />
        <data:Field fieldName="SalesPerson" dataType="String" />
        <data:Field fieldName="EmailAddress" dataType="String" />
        <data:Field fieldName="Phone" dataType="String" />
        <data:Field fieldName="PasswordHash" dataType="String" isNullable="False" />
        <data:Field fieldName="PasswordSalt" dataType="String" isNullable="False" />
        <data:Field fieldName="rowguid" dataType="Guid" />
        <data:Field fieldName="ModifiedDate" dataType="DateTime" />
      </data:DbSetInfo.fieldInfos>
    </data:DbSetInfo>
    <data:DbSetInfo enablePaging="True" pageSize="25" dbSetName="CustomerAddress" EntityType="{x:Type riaddal:CustomerAddress}"
deleteDataMethod="Delete{0}" insertDataMethod="Insert{0}" updateDataMethod="Update{0}">
      <data:DbSetInfo.fieldInfos>
        <data:Field fieldName="CustomerID" isPrimaryKey="1" dataType="Integer" />
```

If you obtain the XML version of the metadata, then you can simplify the *GetMetadata* method of the DataService in order to return the metadata contained in the XML control's resources.

```
 protected override Metadata GetMetadata(bool isDraft)
  {
          if (isDraft)
          {
             //returns raw (uncorrected) programmatically generated metadata from LinqToSQL classes
             return base.GetMetadata(true);
          }
          else
          {
             //first the uncorrected metadata was saved into xml file and then edited
             return
Metadata.FromXML(ResourceHelper.GetResourceString("RIAppDemo.BLL.Metadata.MainDemo.xml"));
          }
 }
```

Note: *The good part of storing metadata in a XAML form, that it is agnostic of the technology with which the DataService works. It can be a Linq for SQL, an Entity framework, an ADO NET or any other technology. The XAML is also type safe and can be generated with existing XML tools.*
It is better to use a WPF control for editing XAML because it is type safe and you have autocomplete for values. The demo solution contains ***WpfMetadataEditor*** *project for editing XAML. It contains MetadataEditor.xaml file which contains the metadata.*

The DataService class usually has some query methods to returns the results of queries. They are distinguished from the other methods by annotating them with a *Query* attribute.

```
 [Query]
 public QueryResult<Product> ReadProduct(int[] param1, string param2)
 {
          int? totalCount = null;
```

```
        var res = this.QueryHelper.PerformQuery(this.DB.Products,this.CurrentQueryInfo, ref
totalCount).AsEnumerable();
        var queryResult = new QueryResult<Product>(res, totalCount);

        //example of adding out of band information to the result and use it on the client (of course, it can be more
useful than this)
        queryResult.extraInfo = new { test = "ReadProduct Extra Info: " + DateTime.Now.ToString("dd.MM.yyyy
HH:mm:ss") };

        return queryResult;
 }
```

Query methods return a *QueryResult* instance. In order to simplify the querying, you can use a *QueryHelper* class instance, which can be used to perform queries in a more generic way. It can take result of *this.CurrentQueryInfo* property, and use it to filter and sort the result of the query. This method also can take arbitrary parameters, which can be used for custom filtering or for executing stored procedures.

A DbSet can have several query methods with different names (*no overloading*). For example the Product DbSet in the DEMO application have another specialized query method which returns products by their ids.

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

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 other purposes.

If a query can return a large number of rows then you can set a *FetchSize* in the metadata - the batch size of the rows fetching (*the default value is 1000, and usually it is ok*). The DataService will send the data to the client in batches, not exceeding the *FetchSize*.

an example of setting a 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}">
```

In addition to the query methods, there are also CRUD methods (*they are optional, if you don't need to allow editing*) to perform inserts, deletes and updates on the entities. Their names are defined in the DbSet's metadata as *insertDataMethod ,updateDataMethod, deleteDataMethod*.

They usually have templated names, such as Insert{0}. The real name is obtained by replacing {0} with a dbSetName's value.

But it is currently better to use attributes in your data service to mark them: [Insert],[Update],[Delete]. You can see the demo how they are used. When you use method attributes then you don't need to use *insertDataMethod ,updateDataMethod, deleteDataMethod* in the XAML!

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

Refresh methods

They are used to refresh an entity with the data from the service. A refresh can be also made by using a query method and returning only one row from it, but the refresh methods are more convenient to use from the client (*just use entity's refresh method*). The refresh method name can be set in the metadata as *refreshDataMethod* property value.

an example of a refresh method implementation:

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

Custom validation methods

They are used to validate an entity when the custom validation is needed. The validation method name can be set in the metadata as *validateDataMethod* property value.

an example of a server side custom validation method:

```
 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;
 }
```

Note: *the modifiedField parameter allows you to validate only modified fields. No sense to validate the data which is already stored in the database.*

Service methods

They are executed from the client, to make some sort of processing on the server and optionally to return a result. These methods are distinguished from the other ones by annotating them with an *Invoke* attribute. They can also have an *Authorize* attribute.

```
[Invoke()]
 public string TestInvoke(byte[] param1, string param2)
 {
           StringBuilder sb = new StringBuilder();

           Array.ForEach(param1, (item) => {
              if (sb.Length > 0)
                 sb.Append(", ");
              sb.Append(item);
           });

           /*
           int rand = (new Random(DateTime.Now.Millisecond)).Next(0, 999);
           if ((rand % 3) == 0)
              throw new Exception("Error generated randomly for testing purposes. Don't worry! Try again.");
           */

           return string.Format("TestInvoke method invoked with<br/><br/><b>param1:</b> {0}<br/> <b>param2:</b>
{1}", sb, param2);
 }

 [Invoke()]
 public void TestComplexInvoke(AddressInfo info, KeyVal[] keys)
  {
           //p.s. do something with info and keys
  }
```

Metadata

The *Metadata* class contains two collections: *DbSets* and *Associations.*
the DbSets collection contains *DbSetInfo* typed items (*which represent information for a DbSet*), which in their turn contain collection of *Field* items (*which represent fields for the DbSet*).
The *Field* class contains attributes of the field: its name, its data type and the other attributes. The *Field* is initialized with default attribute values:

```
   this.isPrimaryKey = 0;
   this.dataType = DataType.None;
   this.isNullable = true;
   this.maxLength = -1;
   this.isReadOnly = false;
   this.isAutoGenerated = false;
   this.allowClientDefault = false;
   this.dateConversion = DateConversion.None;
   this.fieldType = FieldType.None;
   this.isNeedOriginal = true;

   this.range = "";
   this.regex = "";
   this.dependentOn = "";
```

The meaning of the attributes is clear from their names.
But several attributes need more 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 it is 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 (*readonly*). The values for the primary key can be also generated on the client side. In that case you need to allow the field assignment on the client (*for new entities only*) that is done by setting *allowClientDefault=true.*

The *dateConversion* attribute determines how conversion of dates values between the server and the client is done. You can choose between three values:

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

The first option means that no conversion is performed. The remaining two options take the server and the client timezones into consideration.
For example, if you choose *ServerLocalToClientLocal* then the date values will be converted from the server local time to the client local time (*and vice versa*).
If you choose *UtcToClientLocal* (*first make sure the dates on the server are really UTC)*, then the dates values will be converted from UTC to the client local time zone (*and vice versa*).

Note: *This is helpful for distributed applications, because clients and servers can be in different time zones, and the dates will be displayed to the client in its own timezone.*

isNeedOriginal - the default is true. By setting it to false can conserve a little of bandwidth when submitting changes. But it can be done carefully , because if you set *isNeedOriginal* attribute to false for the fields which needs original values on submit (*for optimistic concurency check*) - the the update will fail, saying that the row was modified before you applied the updates.

isAutoGenerated - prevents the field to accept updates from the client (*it ignores the updates for them.*).
range - the attribute is used to set accepted range of values for automatic validation. For example, range="100,5000" or for dates, range="2000-01-01,2015-01-01"
regex - the attribute is used to set regular expression for automatic validation.
For example, regex="^[_a-z0-9-]+(\.[_a-z0-9-]+)*@[a-z0-9-]+(\.[a-z0-9-]+)*(\.[a-z]{2,4})$"
fieldType - can have a value from the *FieldType* enumeration:

```
public enum FieldType : int
 {
        None = 0, ClientOnly = 1, Calculated = 2, Navigation = 3, RowTimeStamp = 4, Object = 5
 }
```

Associations

The association defines foreign key references and navigation fields names. For example, in the Demo there are defined the next associations:

```
<data:Metadata.Associations>
            <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>
            <data:Association name="CustAddrToAddress2" parentDbSetName="AddressInfo"
childDbSetName="CustomerAddress" childToParentName="AddressInfo"
parentToChildrenName="CustomerAddresses">
```

```
                <data:Association.fieldRels>
                    <data:FieldRel parentField="AddressID" childField="AddressID"></data:FieldRel>
                </data:Association.fieldRels>
            </data:Association>
            <data:Association name="OrdDetailsToOrder" parentDbSetName="SalesOrderHeader"
childDbSetName="SalesOrderDetail" childToParentName="SalesOrderHeader"
parentToChildrenName="SalesOrderDetails" onDeleteAction="Cascade">
                <data:Association.fieldRels>
                    <data:FieldRel parentField="SalesOrderID" childField="SalesOrderID"></data:FieldRel>
                </data:Association.fieldRels>
            </data:Association>
            <data:Association name="OrdDetailsToProduct" parentDbSetName="Product"
childDbSetName="SalesOrderDetail" childToParentName="Product" parentToChildrenName="SalesOrderDetails">
                <data:Association.fieldRels>
                    <data:FieldRel parentField="ProductID" childField="ProductID"></data:FieldRel>
                </data:Association.fieldRels>
            </data:Association>
            <data:Association name="OrdersToCustomer" parentDbSetName="Customer"
childDbSetName="SalesOrderHeader" childToParentName="Customer">
                <data:Association.fieldRels>
                    <data:FieldRel parentField="CustomerID" childField="CustomerID"></data:FieldRel>
                </data:Association.fieldRels>
            </data:Association>
            <data:Association name="OrdersToShipAddr" parentDbSetName="Address"
childDbSetName="SalesOrderHeader" childToParentName="Address">
                <data:Association.fieldRels>
                    <data:FieldRel parentField="AddressID" childField="ShipToAddressID"></data:FieldRel>
                </data:Association.fieldRels>
            </data:Association>
            <data:Association name="OrdersToBillAddr" parentDbSetName="Address"
childDbSetName="SalesOrderHeader" childToParentName="Address1">
                <data:Association.fieldRels>
                    <data:FieldRel parentField="AddressID" childField="BillToAddressID"></data:FieldRel>
                </data:Association.fieldRels>
            </data:Association>
 </data:Metadata.Associations>
```

If you set the *childToParentName* property on the association it will add a navigation field to the child entity, by which can be accessed the parent entity.
If you set the *parentToChildrenName* property on the association it will add a navigation field to the parent entity, by which can be accessed the child entities.

Note: *You can not set* childToParentName *and* parentToChildrenName *properties on the association. If not set, then the respective navigation fields won't be added, but the associations can be accessed on the client as usual from the DbContext's* getAssociation *method.*

On the client side you can obtain related entities through the associations or more conveniently through the navigation fields.

There are two ways of loading related entities onto the client:

The first one is to include related entities into the result of a query method by using includeNavigations property on the QueryResult.

An example of inclusion of related entities in the result using navigation hierarchy:

```
[Query]
public QueryResult<Customer> ReadCustomer()
{
        DataLoadOptions opt = new DataLoadOptions();
        opt.LoadWith<Customer>(m => m.CustomerAddresses);
        opt.LoadWith<CustomerAddress>(m => m.Address);
        this.DB.LoadOptions = opt;
        //CustomerAddresses is parentToChildrenName property value, and Address is childToParentName value
        //that is Customer.CustomerAddresses.Address
        string[] includeHierarchy =  new string[] { "CustomerAddresses.Address" };
        int? totalCount = null;
        var res = this.QueryHelper.PerformQuery(this.DB.Customers, this.CurrentQueryInfo, ref
                totalCount).AsEnumerable();
       return new QueryResult<Customer>(result: res, totalCount: totalCount, includeNavigations: includeHierarchy);
}
```

Another option is to include related entities into the result of a query method by explicitly adding them to the result.

An example of explicit inclusion of related entities in the result:

```
 [Query]
 public QueryResult<Product> ReadProduct(int[] param1, string param2)
 {
        int? totalCount = null;
        var res = this.QueryHelper.PerformQuery(this.DB.Products,this.CurrentQueryInfo, ref totalCount).ToArray();
        var productIDs = res.Select(p => p.ProductID).Distinct().ToArray();

        var queryResult = new QueryResult<Product>(res, totalCount);
        //include related SalesOrderDetails with the products in the same query result
        queryResult.subResults.Add(new SubResult() { dbSetName = "SalesOrderDetail", Result =
this.DB.SalesOrderDetails.Where(sod => productIDs.Contains(sod.ProductID)) });

        //example of returning out of band information and use it on the client (of 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;
 }
```

Note: *But the above two options are not always the best when the parent entities are retrieved by slow query. In these cases it is better to load child and parent entities from the client using separate queries. The separate dbSet loading allows you to preload many pages (**data pages , not HTML pages**) of parent (master) entities at once (**setting the loadPageCount on a query to more than 1**), and then to load the details (**only for the current page in the datagrid**). When the user goes to another data page the application retrieves only the details entities for the page (and clears them for the previous one).*
This will improve the user experience when the master entities are retrieved by slow query and user needs to wait a long time when she goes from one page to another.
The details are usually selected by their keys, so they are always retrieved fast.

For example, you can first load 20 data pages of the Customer entities, and then you can load CustomerAddress entities for the current page of the loaded customers.
You can see an example of loading related entities in this way in the ManyToMany Demo.

Examples of two server side query mehods which accept parameters:

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

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

Authorization

The authorization can be applied on two levels - the data service class level and a method's level.
To make it work you need to annotate a data service class or a method with an *Authorize* attribute. The Authorize attribute can include user roles. Without including the roles it is simply checks 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.

First the access is checked at the data service level, and if it is not allowed, there are no further checks except if the a method is annotated with the *AllowAnonymous* attribute. In that case the access to the method is allowed.

At the lower level (*method's level*) which encompasses query methods, CRUD methods, refresh and service (*invoke*) methods, the authorization checks the method level permissions.

```
[Authorize()]
public class RIAppDemoService : LinqForSqlDomainService<RIAppDemoDataContext>
{
  private const string USERS_ROLE = "Users";
  private const string ADMINS_ROLE = "Admins";

 [Query]
  public QueryResult<Customer> ReadCustomer()
  {
         int? totalCount = null;
        var res = this.QueryHelper.PerformQuery(this.DB.Customers, this.CurrentQueryInfo,
          ref totalCount).AsEnumerable();
        return new QueryResult<Customer>(res, totalCount);
 }

 [Authorize(Roles = new string[] { ADMINS_ROLE })]
 public void InsertCustomer(Customer customer)
  {
        customer.PasswordHash = "";
        customer.PasswordSalt = "";
        customer.ModifiedDate = DateTime.Now;
        customer.rowguid = Guid.NewGuid();
        this.DB.Customers.InsertOnSubmit(customer);
 }

 [Authorize(Roles = new string[] { ADMINS_ROLE })]
 public void UpdateCustomer(Customer customer)
 {
        Customer orig = this.GetOriginal<Customer>();
        this.DB.Customers.Attach(customer, orig);
 }

 [Authorize(Roles = new string[] { ADMINS_ROLE })]
 public void DeleteCustomer(Customer customer)
 {
        this.DB.Customers.Attach(customer);
        this.DB.Customers.DeleteOnSubmit(customer);
 }

 public Customer RefreshCustomer(RefreshRowInfo refreshInfo)
 {
        return this.QueryHelper.GetRefreshedEntity<Customer>(this.DB.Customers, refreshInfo);
 }

[AllowAnonymous()]
[Query]
public QueryResult<ProductCategory> ReadProductCategory()
{
        int? totalCount = null;
        var res = this.QueryHelper.PerformQuery(this.DB.ProductCategories, this.CurrentQueryInfo,
          ref totalCount).AsEnumerable();
        return new QueryResult<ProductCategory>(res, totalCount);
}
```

The authorization behaviour can be extended (*or replaced*) by creating a custom authorizer which implements the *IAuthorizer* interface.

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

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

```
 protected virtual IAuthorizer CreateAuthorizer()
 {
        return new AuthorizerClass(this._dataServiceType, this._principal);
 }
```

Change Tracking (auditing)

The BaseDomainService has a virtual method *OnTrackChange* which can be overridden in the DataService. This method provides three arguments which can be used to get information about the entity values changes. The diffgram parameter contains a map of changes in xml form. You must set in the metadata for the DbSet *isTrackChanges* attribute to true, so this type of the entity should be tracked.

```
/// <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)
 {
          /// can log changes here
 }
```

an example of a diffgram for the Product entity:

```
<diffgram>
  <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" />
</diffgram>
```

Error logging in the data service

An Error logging can be implemented in the data service by overriding *OnError* method. Each unhandled error can be seen in this method.

```
protected override void OnError(Exception ex)
{
   //Error logging could be implemented here
}
```

Disposal of resources used by the data service (cleanup)

The data service has a *Dispose* method which can be overridden to clean up additional resources.

an example of an overridden Dispose method:

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

         base.Dispose(isDisposing);
   }
```

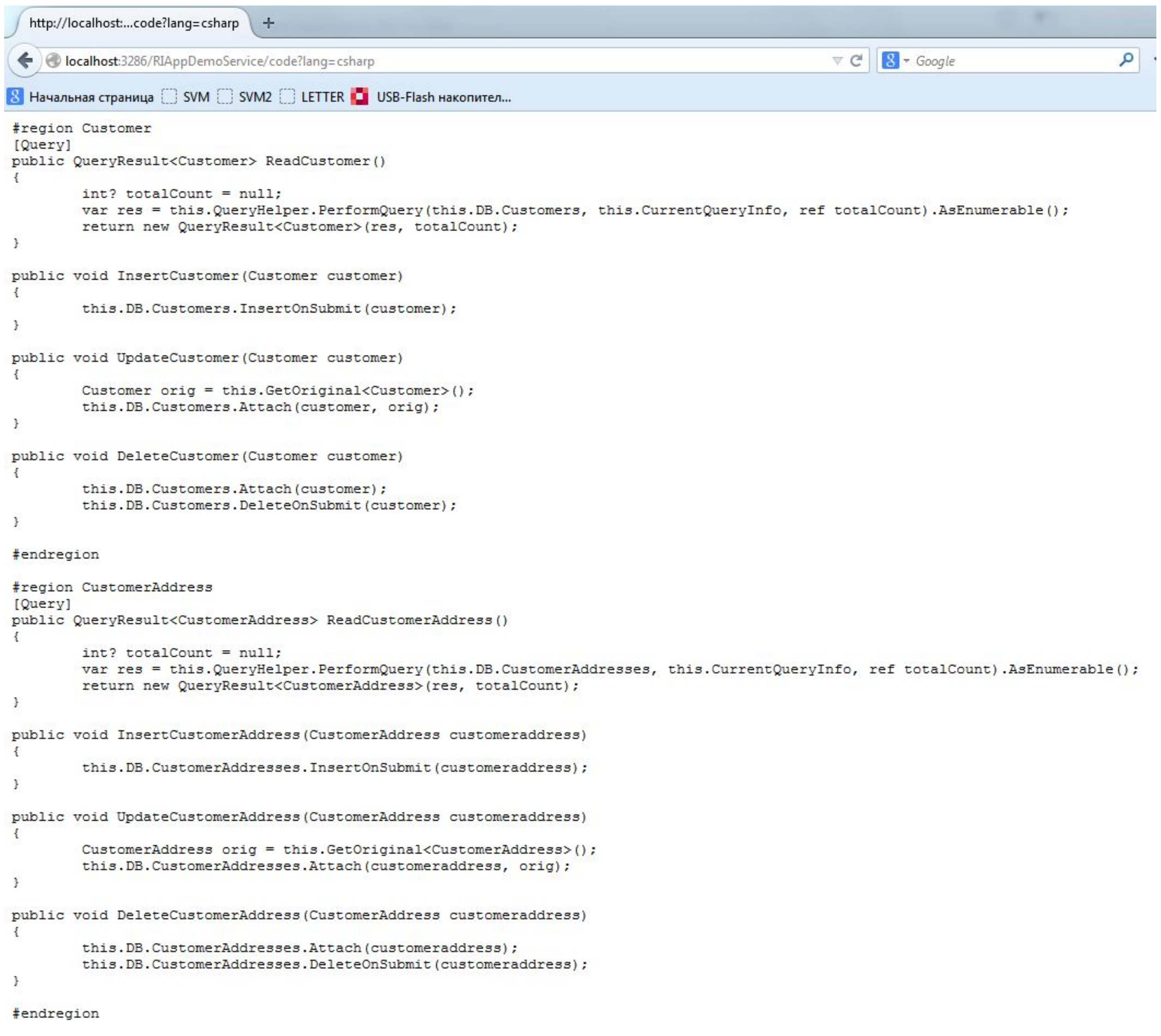
Code generation- obtaining the raw implementation of the data service's methods

The data service has a protected *GetCSharp* method. It is not implemented in the *BaseDomainService* class. The demo's Linq for SQL and Entity Framework dataservices implement this method as an example.

```
protected override string GetCSharp()
{
       var metadata =  this.ServiceGetMetadata();
       return RIAPP.DataService.LinqSql.Utils.DataServiceMethodsHelper.CreateMethods(metadata, this.DB);
}
```

Navigating in the internet browser (*for the demo application*) to
http://YOURSERVER/RIAppDemoService/code?lang=csharp
will return crud methods implementation for the data service.

Note: *The Entity framework version of the DataService has a similar helper class in the RIAPP.DataService.EF.Utils namespace.*


http://localhost:...code?lang=csharp
localhost:3286/RIAppDemoService/code?lang=csharp
Google
Начальная страница
SVM
SVM2
LETTER
USB-Flash накопител...

```
#region Customer
[Query]
public QueryResult<Customer> ReadCustomer()
{
        int? totalCount = null;
        var res = this.QueryHelper.PerformQuery(this.DB.Customers, this.CurrentQueryInfo, ref totalCount).AsEnumerable();
        return new QueryResult<Customer>(res, totalCount);
}

public void InsertCustomer(Customer customer)
{
        this.DB.Customers.InsertOnSubmit(customer);
}

public void UpdateCustomer(Customer customer)
{
        Customer orig = this.GetOriginal<Customer>();
        this.DB.Customers.Attach(customer, orig);
}

public void DeleteCustomer(Customer customer)
{
        this.DB.Customers.Attach(customer);
        this.DB.Customers.DeleteOnSubmit(customer);
}

#endregion

#region CustomerAddress
[Query]
public QueryResult<CustomerAddress> ReadCustomerAddress()
{
        int? totalCount = null;
        var res = this.QueryHelper.PerformQuery(this.DB.CustomerAddresses, this.CurrentQueryInfo, ref totalCount).AsEnumerable();
        return new QueryResult<CustomerAddress>(res, totalCount);
}

public void InsertCustomerAddress(CustomerAddress customeraddress)
{
        this.DB.CustomerAddresses.InsertOnSubmit(customeraddress);
}

public void UpdateCustomerAddress(CustomerAddress customeraddress)
{
        CustomerAddress orig = this.GetOriginal<CustomerAddress>();
        this.DB.CustomerAddresses.Attach(customeraddress, orig);
}

public void DeleteCustomerAddress(CustomerAddress customeraddress)
{
        this.DB.CustomerAddresses.Attach(customeraddress);
        this.DB.CustomerAddresses.DeleteOnSubmit(customeraddress);
}

#endregion
```

Warning: *To allow this work you need to set DataService's IsCodeGenEnabled property value to true, in order code generation methods can be executed from the internet browser. Without it you will get an error!*
For security reasons it is recommended on deployment to the production that you disable this feature by setting this property to false!

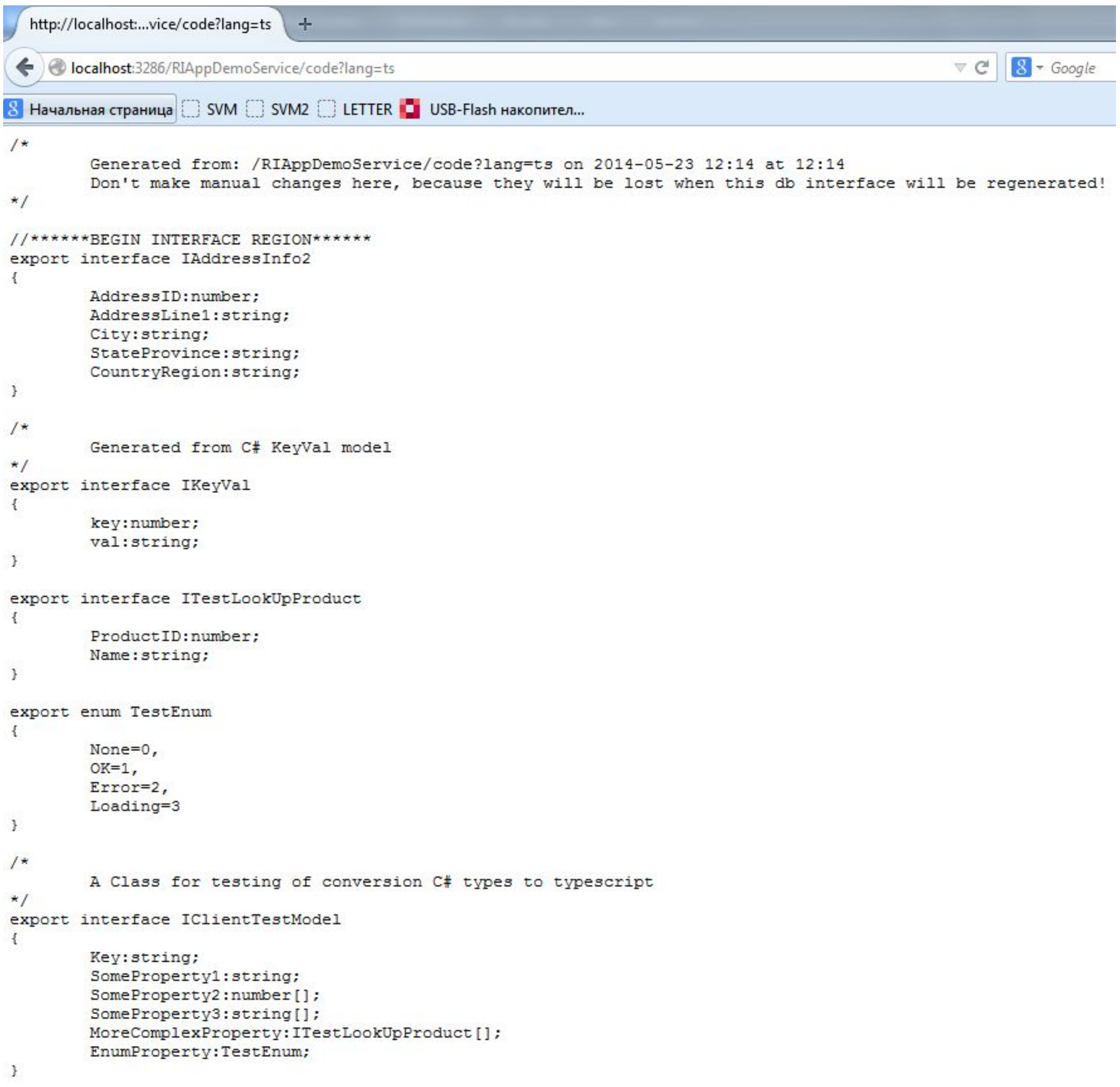
an example of setting IsCodeGenEnabled property to true

```
public RIAppDemoService(IServiceArgs args)
        : base(args)
{
        //it allows getting information via GetCSharp, GetXAML, GetTypeScript
        //it should be set to false in release version
        //allow it only at development time
        this.IsCodeGenEnabled = true;
}
```

Generating a typescript code for the data service classes

The data service exposes a *GetTypeScript* method which returns generated code for the strongly typed entities, DbSets, and DbContext classes. You can obtain this script by

navigating in the internet browser to the
http://YOURSERVER/RIAppDemoService/code?lang=ts url.


http://localhost:...vice/code?lang=ts
localhost:3286/RIAppDemoService/code?lang=ts
Google
Начальная страница
SVM
SVM2
LETTER
USB-Flash накопител...

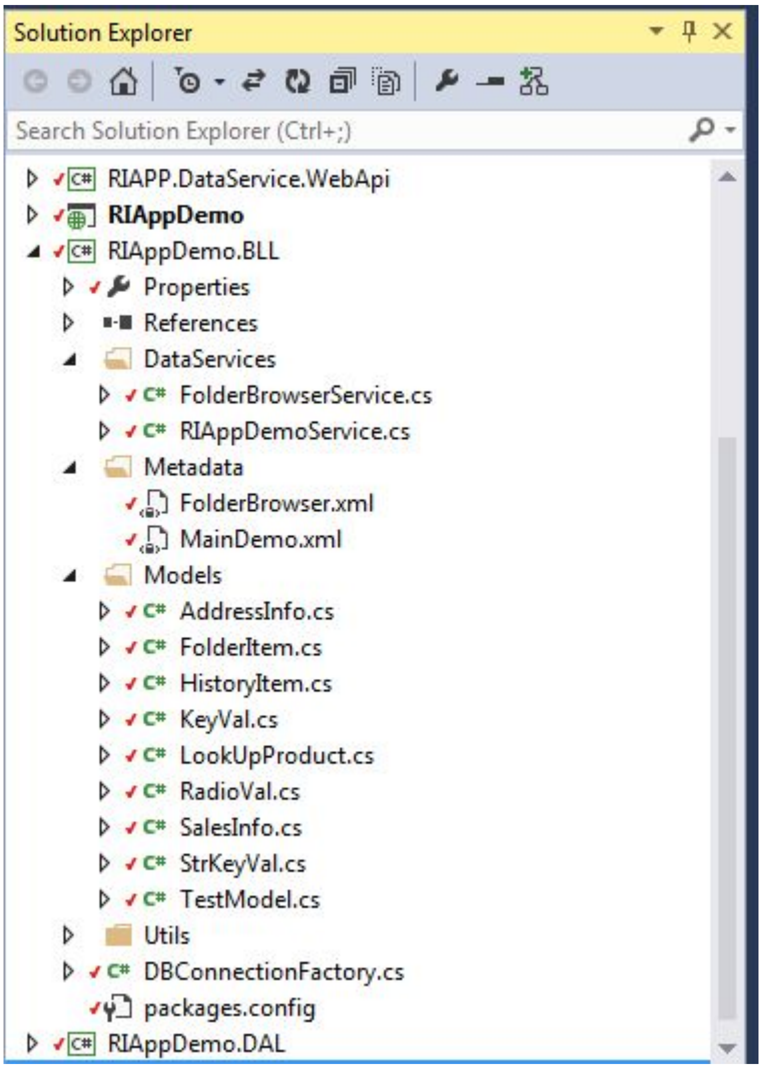
```
/*
        Generated from: /RIAppDemoService/code?lang=ts on 2014-05-23 12:14 at 12:14
        Don't make manual changes here, because they will be lost when this db interface will be regenerated!
*/

//******BEGIN INTERFACE REGION******
export interface IAddressInfo2
{
        AddressID:number;
        AddressLine1:string;
        City:string;
        StateProvince:string;
        CountryRegion:string;
}

/*
        Generated from C# KeyVal model
*/
export interface IKeyVal
{
        key:number;
        val:string;
}

export interface ITestLookUpProduct
{
        ProductID:number;
        Name:string;
}

export enum TestEnum
{
        None=0,
        OK=1,
        Error=2,
        Loading=3
}

/*
        A Class for testing of conversion C# types to typescript
*/
export interface IClientTestModel
{
        Key:string;
        SomeProperty1:string;
        SomeProperty2:number[];
        SomeProperty3:string[];
        MoreComplexProperty:ITestLookUpProduct[];
        EnumProperty:TestEnum;
}
```

You can force any class from your server side code to be included into this generated typescript code.
The DataService can override base class GetClientTypes method to return an array of types which should be included in the result typescript code.

```
/// <summary>
/// this types will be autogenerated in typescript when clients will use GetTypeScript method of the data service
/// </summary>
/// <returns></returns>
protected override IEnumerable<Type> GetClientTypes()
{
    return new Type[] { typeof(TestModel), typeof(KeyVal), typeof(HistoryItem), typeof(TestEnum2) };
}
```

Note: *The types which are used as parameters and results of the service methods are automatically included into the code generation. So, you don't need to return them from GetClientTypes method (but it won't hurt if you do).*

The demo project in RIAppDemo.BLL contains the Models folder. It contains classes that are needed on the client side in the generated typescript code.


```
Solution Explorer
Search Solution Explorer (Ctrl+;)
RIAPP.DataService.WebApi
RIAppDemo
RIAppDemo.BLL
    Properties
    References
    DataServices
        FolderBrowserService.cs
        RIAppDemoService.cs
    Metadata
        FolderBrowser.xml
        MainDemo.xml
    Models
        AddressInfo.cs
        FolderItem.cs
        HistoryItem.cs
        KeyVal.cs
        LookUpProduct.cs
        RadioVal.cs
        SalesInfo.cs
        StrKeyVal.cs
        TestModel.cs
    Utils
    DBConnectionFactory.cs
    packages.config
RIAppDemo.DAL
```

Some classes in the Models folder are annotated with attributes which adjust code generation for those classes.

Applying a *TypeName* attribute on the class changes the name of the generated interface.

A class:

```
[TypeName("IAddressInfo2")]
 public class AddressInfo
 {
     public int AddressID { get; set; }
     public string AddressLine1 { get; set; }
     public string City { get; set; }
     public string StateProvince { get; set; }
     public string CountryRegion { get; set; }
 }
```

will be outputted as:

```
export interface IAddressInfo2 {
        AddressID: number;
        AddressLine1: string;
```

```
          City: string;
          StateProvince: string;
          CountryRegion: string;
 }
```

A more heavily annotated class:

```
   [List(ListName="HistoryList")]
   [Comment(Text = "Generated from C# HistoryItem model")]
   [TypeName("IHistoryItem")]
   [Extends(InterfaceNames= new string[]{"RIAPP.IEditable"})]
   public class HistoryItem
   {
      public string radioValue
      {
         get;
         set;
      }

      public DateTime time
      {
         get;
         set;
      }
     }
```

will provide the output as:

```
/*
     Generated from C# HistoryItem model
*/
export interface IHistoryItem extends RIAPP.IEditable {
       radioValue: string;
       time: Date;
}

export class HistoryItemListItem extends RIAPP.MOD.collection.ListItem implements IHistoryItem {
          constructor(coll: RIAPP.MOD.collection.BaseList<HistoryItemListItem, IHistoryItem>, obj?: IHistoryItem)
           {
             super(coll, obj);
           }
          get radioValue(): string { return <string>this._getProp('radioValue'); }
          set radioValue(v: string) { this._setProp('radioValue', v); }
          get time(): Date { return <Date>this._getProp('time'); }
          set time(v: Date) { this._setProp('time', v); }
          asInterface() { return <IHistoryItem>this; }
 }

export class HistoryList extends RIAPP.MOD.collection.BaseList<HistoryItemListItem, IHistoryItem> {
          constructor() {
             super(HistoryItemListItem, [{ name: 'radioValue', dtype: 1 }, { name: 'time', dtype: 6 }]);
             this._type_name = 'HistoryList';
          }
          get items2() { return <IHistoryItem[]>this.items; }
}
```

The applied *List* attribute says that the code generation should produce a strongly typed list class for this type.

The next annotated class has a *Dictionary* attribute which is used to tell the code generation that the strongly typed dictionary class should be generated.

```
[Dictionary(KeyName="key", DictionaryName="KeyValDictionary")]
[Comment(Text="Generated from C# KeyVal model")]
[TypeName("IKeyVal")]
public class KeyVal
```

```
{
      public int key
      {
         get;
         set;
      }

      public string val
      {
         get;
         set;
      }
}
```

The result of the code generation from the above KeyVal class:

```
 /*
       Generated from C# KeyVal model
 */
 export interface IKeyVal {
          key: number;
          val: string;
 }

 export class KeyValListItem extends RIAPP.MOD.collection.ListItem implements IKeyVal {
          constructor(coll: RIAPP.MOD.collection.BaseList<KeyValListItem, IKeyVal>, obj?: IKeyVal) {
             super(coll, obj);
          }
          get key(): number { return <number>this._getProp('key'); }
          set key(v: number) { this._setProp('key', v); }
          get val(): string { return <string>this._getProp('val'); }
          set val(v: string) { this._setProp('val', v); }
          asInterface() { return <IKeyVal>this; }
 }

 export class KeyValDictionary extends RIAPP.MOD.collection.BaseDictionary<KeyValListItem, IKeyVal> {
          constructor() {
             super(KeyValListItem, 'key', [{ name: 'key', dtype: 3 }, { name: 'val', dtype: 1 }]);
             this._type_name = 'KeyValDictionary';
          }
          get items2() { return <IKeyVal[]>this.items; }
 }
```

The code generation also produces typescript's *enums* from the c# *enums*.

ASP.Net MVC

The ASP.Net MVC project *RIAppDemo* references the *RIAppDemo.BLL* library.
In the ASP.NET MVC project the DataServices are exposed through MVC controllers.
The web request from the client first hits the controller and then the controller relays the request to the DataService's instance which it encapsulates.

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

```
public class RIAppDemoServiceController : DataServiceController<RIAppDemoService>
{
        [ChildActionOnly]
        public string ProductModelData()
        {
            var info = this.GetDomainService().GetQueryData("ProductModel",
"ReadProductModel");
            return this.Serializer.Serialize(info);
        }
        [ChildActionOnly]
        public string ProductCategoryData()
        {
            var info = this.GetDomainService().GetQueryData("ProductCategory",
"ReadProductCategory");
            return this.Serializer.Serialize(info);
        }

}
```

Note: *You can add to the controller any methods you need. For example, the above controller adds two custom methods which are used on the HTML page. They are used to return the lookup data and it is embedded into the page in the javascript section:*

```
ops.modelData = @Html.Action("ProductModelData", "RIAppDemoService");
ops.categoryData = @Html.Action("ProductCategoryData", "RIAppDemoService");
```