

jRIAppTS, the RIA framework – user's guide

jRIAppTS, the RIA framework

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

The framework's classes

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

Working with the data on the client side

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

- DataView
- Associations and ChildDataViews
- Data validation

Working with the data on the server side

4.1 Data service

- Data service's public interface
- Exposing the data service through ASP.NET MVC controller
- Query methods
- Entity refresh methods
- Custom validation methods
- DataService's methods
- DataService's metadata
- Associations (*foreign keys relationship*)
- Authorization
- Change tracking
- Error logging
- DataService disposal (*cleanup*)
- Code generation

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 – the client and the server parts. The client part was written in typescript language. The server part was written in C# and the demo application was implemented in ASP.NET MVC (*it can also be written in other languages, for example Ruby or Java, but you have to roll up your sleeves and prepare to write them*).

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

The Client part resembles Microsoft Silverlight client development, only it is based on HTML (*not XAML*), and uses 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 a server side dataservice, datatemplates, 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 datacontents inside of this region. It also provides for summary error display.

A [DbContext](#) – the control used as a data manager to store the data (*DbSets*) and to cache changes on the client for submitting them 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 single page applications.

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

1.2 Licensing

The MIT License

Copyright (c) 2013 Maxim V. Tsapov

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

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

OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

1.3 *The Framework's files and deployment*

The framework is written in typescript. So, the typescript code is compiled first to javascript before the 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: `tsc --out $(ProjectDir)\jriapp.js --d --target ES5 $(ProjectDir)\main\app.ts`

At the end of the compilation in the Project directory will appear two files: jriapp.js and jriapp.d.ts.

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

The has the next references:

```
/// <reference path=".\\jquery\\jquery.d.ts" />
/// <reference path="app_en.ts"/>
/// <reference path="baseobj.ts"/>
/// <reference path="globalobj.ts"/>
/// <reference path="..\\modules\\consts.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\\datepicker.ts"/>
/// <reference path="..\\modules\\mvvm.ts"/>
/// <reference path="..\\modules\\baseElView.ts"/>
/// <reference path="..\\modules\\binding.ts"/>
/// <reference path="..\\modules\\collection.ts"/>
/// <reference path="..\\modules\\template.ts"/>
/// <reference path="..\\modules\\baseContent.ts"/>
/// <reference path="..\\modules\\dataform.ts"/>
/// *** the rest are optional modules, which can be removed if not needed ***
/// <reference path="..\\modules\\db.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"/>
```

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 **jriapp.css** (*the css styles for the frameworks's UI controls*) and the **img** folder – which contains the images used by the framework's controls.

The demo application which demonstrates the capabilities of the framework was created using ASP.NET MVC web site project. The project uses the layout page (*_LayoutDemo.cshtml*), which is used by all pages included in the demo web site (*in the RIAppDemo solution*).

The *demoTS* typescript project contains user modules, and on compilation it produces the javascript files which are used in the demo ASP.NET MVC web site.

This layout page includes the core files of the framework (*jriapp.js* and *jriapp.css*) and some javascript and css files which are used on every demo page (jquery.js, bootstrap.js, qtip.js, moment.js).

```
<head>
  <meta charset="utf-8" />
  <title>@ViewBag.Title</title>
  <link href="@Url.Content("~/Content/themes/redmond/jquery-ui-1.9.2.custom.min.css")" rel="stylesheet" type="text/css" />
  <link href="@Url.Content("~/Scripts/bootstrap/css/bootstrap.min.css")" rel="stylesheet" type="text/css" />
  <link href="@Url.Content("~/Scripts/qtip/jquery.qtip.min.css")" 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" />

  @RenderSection("CssImport", false)

  <script src="@Url.Content("~/Scripts/jquery/jquery-1.8.3.min.js")" type="text/javascript"></script>
  <script src="@Url.Content("~/Scripts/jquery/jquery-ui-1.9.2.custom.min.js")" type="text/javascript"></script>
  <script src="@Url.Content("~/Scripts/bootstrap/js/bootstrap.min.js")" type="text/javascript"></script>
  <script src="@Url.Content("~/Scripts/qtip/jquery.qtip.min.js")" type="text/javascript"></script>
  <script src="@Url.Content("~/Scripts/moment/moment.js")" type="text/javascript"></script>
  <script src="@Url.Content("~/Scripts/jriapp/jriapp.js",true)" type="text/javascript"></script>

  @RenderSection("JSImport", false)
</head>
```

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 dates formatting*), and qtip (*for tooltips*). Thus, these javascript libraries files must be always referenced on the 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 libraries.

Note: *The bootstrap.js is not required for the framework's functionality. It is included as a library for page's UI creation. You can use any javascript libraries with the framework which you find useful.*

The demo pages besides 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:

```
@section CssImport
{
}

@section JSImport
{
  <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 include the code generated from the *dataservice's GetTypeScript* method, which contains autogenerated classes and interfaces, strongly typed entities and *DbSets* and the *DbContext* class.

The autogenerated classes are used to communicate with the DataService. In addition to the client side checks, the DataService always performs the checks on the server side for the submits from the clients.

For example, in the *DataGridDemo.cshtml* page *~/Scripts/RIAppDemo/demoDB.js* file is autogenerated and contains classes to work with the DataService.

The *~/Scripts/RIAppDemo/gridDemo.js* file contains view models (*used in the MVVM pattern*) and the application class and depends on the classes in the *demoDB.js*.

The framework's classes

2.1 BaseObject

All the types used in the client part of the framework derived from the [RIAPP.BaseObject](#) type. The [RIAPP.BaseObject](#) defined in [baseobj.ts](#) file and this class adds common logic for the object's destruction and adds events support for all the objects derived from it.

BaseObject's methods:

Methods	Description
addHandler	adds an event handler for an event (<i>with optional support for event namespaces like in jQuery</i>)
removeHandler	removes a handler for an event (<i>can be used to remove all handlers registered within a namespace</i>)
removeNSHandlers	removes all handlers for events registered with an event namespace
addOnPropertyChange	adds a handler for a property change notification
removeOnPropertyChange	removes a handler for a property change notification
raisePropertyChanged	triggers registered events handlers for a property change notification
raiseEvent	triggers registered events handlers for an event
destroy	the method is invoked when the object needs to be destroyed for cleaning up resources such as registered event handlers and other resources. It is usually overridden in descendants (<i>but don't forget to always invoke super method!</i>)
_getEventNames	defines event names supported by the object type. BaseObject type supports 'error', and 'destroyed' events. descendants of the BaseObject can override this method to add their own events.
_onError	typically it is invoked by descendants of the BaseObject on error conditions. It triggers the error event and returns boolean value of handled the error or not. Many frameworks object types override this method, for

	example, in BaseElView type it invokes <code>_onError</code> inherited from <code>BaseObject</code> , and if the error was not handled it invokes application's <code>_onError</code> method.
--	---

BaseObject's events:

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

BaseObject's properties:

Property	Description
<code>_isDestroyed</code>	Is set to true in the <code>BaseObject</code> 's destroy method when the destruction of the object is completed. After it was set the <code>destroy</code> event is fired.
<code>_isDestroyCalled</code>	Is set to true (<i>typically in the descendants</i>) when the destroy method was called. The destruction of the object is not completed, but it is in the progress.

The `BaseObject` class has no public properties, although it has a number of protected fields.

For example, each framework's object has `_isDestroyCalled` field which indicates the object's state. When the destroy method is called this field is set to true (*even if full destruction of object is not completed*). You can check this field value in asynchronously invoked methods' callbacks, to be sure that the object is still alive after some asynchronous operation completed (*because in the meantime the object can be disposed*), like in the next example:

```
setTimeout(function () {
    if (self._isDestroyCalled) //the object state is already destroyed or is destroying
        return;
    self._checkQueue(property, self._owner[property]);
}, 0);
```

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

```
destroy() {
    if (this._isDestroyed) //prevents repeated destroys, if destroyed just return and do nothing
        return;

    this._unbindDS();
    this._clearContent();
    this._$el.removeClass(_css.pager);
    this._el = null;
    this._$el = null;
    super.destroy();
}
```

The initialization of 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: MOD.collection.Dictionary;
_format: string;
_formats: MOD.collection.Dictionary;

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 MOD.collection.Dictionary('MonthType', { key: 0, val: " ", 'key'});
    this._months.fillItems([
        { key: 1, val: 'January' }, { key: 2, val: 'February' }, { key: 3, val: 'March' },
        { key: 4, val: 'April' }, { key: 5, val: 'May' }, { key: 6, val: 'June' },
        { key: 7, val: 'July' }, { key: 8, val: 'August' }, { key: 9, val: 'September' }, { key: 10, val: 'October' },
        { key: 11, val: 'November' }, { key: 12, val: 'December' }], true);

    this._format = 'PDF';
    this._formats = new MOD.collection.Dictionary('format', { key: 0, val: " ", 'key'});
    this._formats.fillItems([
        { key: 'PDF', val: 'Acrobat Reader PDF' }, { key: 'WORD', val: 'MS Word DOC' },
        { key: 'EXCEL', val: 'MS Excel XLS' }], true);
}

_onTestCommandExecuted() {
    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 evaluate its availability
        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 fire two predefined events **'error'**, **'destroyed'**.

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

The Error event handler can set `isHandled` to `true`, and then the error handling is successfully finished without showing to the user.

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

The BaseObject class allows derived classes to override **_getEventNames** method in order to define new custom events as in the example:

```

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

```

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

```

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

```

The last argument of the **addHandler** method is the event's namespace, which is an optional parameter and helps to remove subscriptions to the events in this namespace. For this you can use the **removeNSHandlers** method. For removing the subscriptions you can also use **removeHandler** method, and provide to it the event's name and optionally the event's namespace.

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

The events can be triggered by using **raiseEvent** method as in the example:

```

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

```

You also can subscribe to a property change notification using **addOnPropertyChange** method as in the example:

```

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

```

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

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

To unsubscribe from property change notification you can use `removeOnPropertyChange` method, or use `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

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

`RIAPP.global` - is used to hold references for registered types, application instances, loaded modules, and manages subscriptions to some `window.document`'s events, it also subscribes to `window.onerror` event to handle errors. It also dispatches DOM document keydown events to the currently selected on HTML page UI control (*a `DataGrid` or a `StackPanel`*), so that only one instance of the user control can handle keyboard events at a time (*for example, it is used for selecting a row in the `DataGrid` with the help of up or down keyboard keys*). Also the Global object exposes `load` event which is fired 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. The global object also holds references for registered converters (*some converters are registered in the `converter` module*).

The global object has `defaults` property, which exposes an instance of the `Defaults` object type (*it is instantiated in `defaults` global 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 default
global.defaults.imagesPath = '/Scripts/jriapp/img/';
```

The dates formats use moment.js format style. But defaults for the datepicker use datepicker's date format style.

Global object's methods:

Method	Description
<code>findApp</code>	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 registered type by its name.
registerConverter	Registers a converter by its name. 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 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's view by its name. Registered element's views are used directly (<i>using their names</i>) or indirectly in the databindings.
getImagePath	Get common images paths used in the framework by their names. It just appends provided image name to the default path for the images. It is just a helper method.
registerTemplateGroup	Registers a template's group by its name. The template's group can contain several templates. When one template from a group is needed by a user control, then the whole group of template is loaded from the server. P.S.- Typically this is used in complex SPAs, which has a lot of templates and it is difficult to maintain them in one file. And the application class also has this method. Which is mainly used instead of global's one.
loadTemplates	Loads templates as a batch (<i>as a file with templates</i>) from the server. They are later available to all application instances. It should be used only before an application instance is created. Typically in the global.onload event handler.

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	The currently selected control the DataGrid or the StackPanel which accepts keyboard input like Up or Down keys for scrolling them by 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 keyboard input.
defaults	Yes	Instance of the Default object type, to access or change the default values. This property is initialized by the global defaults module.
UC	--	the namespace (<i>empty object instance</i>) for attaching any custom code. You can attach any code to it which can be used globally.
utils	Yes	Object which contains common utility methods. This property is initialized by the global utils module.
modules	Yes	the namespace (<i>object instance</i>) for obtaining global modules' instances, and through the modules instances you can get access to the types. But it is rarely used because global object has all modules exposed through its respective properties, like: <i>utils, consts, defaults</i> .

consts	Yes	Object which contains public globally accessed constants, like: <code>global.consts.KEYS</code> .
isLoading	Yes	Returns true if the application's instance or global object loads templates from the server.

Global object events:

Event name	Description
unload	Raised when the browser's window unloads
load	Raised when the document DOM structure is fully loaded. The same as using <code>jQuery.ready</code> event handler.

2.3 Application

[RIAPP.Application](#) - class which represents the application.

On creation of the application's instance you can provide application's options 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?: { querySelectorAll: (selectors: string) => NodeList; };
}
```

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 types for user modules initialization. The [initFn](#) is invoked when a 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 page is the scope and the application root refers to `window.document` property. But if you have several applications in one HTML page you can provide different scopes (*typically, a div element*) for each application.

The main method of the application is the [startUp](#), which is used to trigger execution of the callback function (*which is provided as part of the parameter*) and also perform the databinding.

The callback function used as a sandbox environment, in which can be created instances of the view models (*they are usually defined in custom user modules*) and any other user defined objects. After executing the callback function, the application invokes its [onStartup](#) method (*which can be overridden in derived application classes*) and then the application processes the databindings on the HTML page.

Usually each 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 extended options.

For example, the grid 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;
}

```

The demo uses a new specialized application's class which exposes instances of view models (*which are defined in that or other modules*) through its new properties and also exposes the DbContext's instance (*to allow communication with the data service*).

//strongly typed application class

```

export class DemoApplication extends Application {
    _dbContext: DEMODB.DbContext;
    _errorVM: COMMON.ErrorViewModel;
    _headerVM: HEADER.HeaderVM;
    _productVM: ProductViewModel;
    _uploadVM: UploadThumbnailVM;

    constructor(options: IMainOptions) {
        super(options);
        var self = this;
        this._dbContext = null;
        this._errorVM = null;
        this._headerVM = null;
        this._productVM = null;
        this._uploadVM = 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 () {
            return !this.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);

        //adding event handler for our custom event
        this._uploadVM.addOnFilesUploaded(function (s, a) {
            //need to update ThumbnailPhotoFileName
            a.product.refresh();
        });
        this._productVM.filter.modelData = options.modelData;
        this._productVM.filter.categoryData = options.categoryData;
        this._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();
        } 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; }
}

```

The application's instance is usually created in *global.onload* handler (*which has the semantics of JQuery's ready method*) and then the application is started by invoking 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 invoking the **startUp** method you can register template groups, or start loading the data templates from the server.

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

Application's methods:

Method name	Description
registerElView	Registers element views in the application type system
getElementView	Returns the element view which is already attached to the element or creates new element view and attaches it to the DOM element and then returns this new instance.

_getElViewType	Returns the element view type by its registered name. Typically it is used internally by the framework.
registerType	Registers an object type 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, only this method is used to register object's instances, instead of types. 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 when the application is started.
registerTemplateLoader	Registers function which loads individual template asynchronously (<i>on as needed basis</i>) - returns a promise which resolves with a template as a html string. P.S.- <i>See the DataGrid demo, for an example how it is used.</i>
getTemplateLoader	Returns registered template's loader by its name.
loadTemplates	The same as the global's loadTemplates , only it loads templates in the application's scope. They are available only to this application. This method should be used only before the application's startUp method is invoked.
loadTemplatesAsync	Loads templates using a provided loader function, which returns a promise which resolves with the loaded templates as a html string. This method is used internally by loadTemplates method. You can use it in special cases, when templates are obtained from some custom place.
registerTemplateGroup	Registers a group of templates to load 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. - The templates names must be unique between different groups. (<i>See the Single Page Application demo for an example</i>)
registerContentFactory	Registers a factory class for a new custom content. (<i>For example, in listbox.ts module new content factory is registered in its initModule function</i>). The default content factory is registered in baseContent.ts module. P.S.- <i>data grid and data form use content factory to create a specialized content class for each type (string, bool, integer,.. etc) .</i>

Applications's properties:

Property	Read Only	Description
options	Yes	Exposes application's options. Typically it is not used directly.
appRoot	Yes	Exposes the root (<i>element or window.document</i>) of the application.
appName	Yes	the application's unique name. If it is not provided on creation with the options, it will have the default value

		'default'.
modules	--	the namespace (<i>object instance</i>) for obtaining application modules' instances. But it is hardly ever is used in custom code.
global	Yes	exposes an instance of RIAPP.Global object for easier access to the global object.
contentFactory	Yes	exposes the content factory which is used by the application.
UC	---	a namespace (<i>empty object instance</i>) for attaching any custom user code. We can attach any code to it.
VM	---	a namespace (<i>empty object instance</i>) for attaching user defined view models. We can attach view model's instances to this namespace.
app	Yes	<p>Returns self reference. Can be used to assign application's instance as a source for the databindings. It can be helpful in some cases, because databinding's source expression (<i>if we use fixed source</i>) is evaluated from the application instance and we can not leave it empty in that case.</p> <pre>{this.dataContext,to=VM.viewModel,source=app}</pre> <p>We can not use empty source like this (<i>invalid usage</i>)</p> <pre>{this.dataContext,to=VM.viewModel,source=}</pre> <p>If we don't use the source at all like in the next expression</p> <pre>{this.dataContext,to=VM.viewModel}</pre> <p>Then the source is not fixed and is defined by the current data context and is volatile as the data context can change.</p>

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

```
//properties with null values must be initialized on the 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,
  user_modules: [{ name: "COMMON", initFn: COMMON.initModule },
    { name: "HEADER", initFn: HEADER.initModule },
    { name: "GRIDDEMO", initFn: initModule }],
};
```

In a user module provided to the application (*in the options*) , you must define a function (*conventionally named initModule*), 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 the modules. In this function you can register converters, object instances and etc, like this:

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

2.4 Binding

The framework's [Binding](#) class has 7 properties:

Binding's properties:

Property	Description
targetPath	the path for the property which is updated when the source property's value changes
sourcePath	the path for the property which provides a value to the target property
mode	the mode of binding ('OneTime', 'OneWay', 'TwoWay')
source	the source of the data (<i>must be a descendant of the BaseObject</i>)
target	the target of the data (<i>must be a descendant of the BaseObject</i>)
converter	converts the data when it flows between the source and the target (<i>for example, object value to string value and backward</i>)
converterParam	the converter can use the parameter to adjust data conversion (<i>for example, formatting style</i>)
isSourceFixed	Returns true if we set the source in the databinding's expression explicitly.
isDisabled	Is 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's](#) type is created in the code. The application's type 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: 'OneWay',  
    target: this._uploadVM, converter: null, converterParam: null  
});
```

When databindings 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 get real object instances, then it creates instances of the Binding.

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 the real databinding targets (*in place of the raw DOM elements*). The selection of which

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

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

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 the 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 an element view by using the options in the `data-view` attribute value, as in the example:

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

or in

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

or in

```
<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 Databinding's expressions are contained inside a custom `data-bind` attribute's value. The `data-bind` attribute's value can contain multiple binding expressions. Each binding 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. The data binding can be only done to the properties exposed by the element view (*the wrapper of the HTML DOM element*), and not directly to the HTML DOM element's properties.

For example, 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 of the databinding (*an instance of the `SendListVM` view model in this case, which is exposed through the application's property*) in the `OneWay` mode (*which is default value, and can be omitted here*).

When databindings are used **not inside** data templates and data forms, without explicitly providing the source, then the source is assumed to be the application's

instance, but when they are used inside templates, the implicit source is a template's current `datacontext` (*data templates have datacontexts, as well as data forms*).

When the `source` is explicitly provided by the databinding expression, the databinding 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 the data binding's expression (*using the implicit source, not fixed one*), and can write previous binding expression as:

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

But then you should pay attention to where this databinding is used! If you use this 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 dataContext property*), and the datacontext's value can change when the program runs. (*the same applies to dataform's datacontext*)

A 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 null value (*effectively changing the binding's implicit source property value*).

Otherwise, if you explicitly name the source in the databinding's expression, then even if it was used inside a data template, the source will be fixed, and will not change for this binding's instance even if the template's dataContext property is changed (*and the path's evaluation in that case always starts from an application's instance*).

In the above examples, there were the shortcut style of the data binding expression, but you can write a databinding expression in another (*expanded, and rarely used*) 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`.

In the databinding expressions, instead of `=` separator (*which separates the name and the value*), you can equally use `:` separator, such as:

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

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

The data binding instances are created not only on the startup of the application, they also can be created when the application runs. It can happen when the controls on the page create data templates during their life cycle. The 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 the `DataGrid` control instance is databound to the 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 data bindings on the

template elements are evaluated. Later, when the template instances are destroyed (*when a row in the DataGrid is removed*), instances of the databindings are also destroyed with the template's instance.

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

an example of a converter definition (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 when a method of the converter (*any of it*) return **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 **undefined** value when a value from the target is **false**. This prevents it from updating the property value on the source in this case. The scenario is very helpful to bind several radio buttons or check boxes to one source property (*see collections demo, only one radio button updates the source - the one which is checked*).

2.5 Command

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:

```
<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 a command's action (*typically, 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._onError(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 the *this* context for the command's action (*inside the action, this will be 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 in a disabled state.

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

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

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

an example of a HTML markup inside a 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>
```

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

Using a command parameter in the command's action (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._onError(ex, this);
    }
}, self, function (sender, param) {
    return true;
});

```

2.6 Element views

An Element view is a wrapper around a HTML DOM element's and can also wrap other controls which you wish to use in a declarative way. It exposes properties which can be databound. Element views help to use databindings declaratively. They are created when databinding'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. 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 options. Without the options the element view uses its default values.

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

```

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

```

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

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

```

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

```

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

Registration of element views in the baseElView.ts:

```

global.registerElView('template', TemplateElView);
global.registerElView('busy_indicator', BusyElView);
global.registerElView(global.consts.ELVIEW_NM.DYNACONT, DynaContentElView);

```



```

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);
global.registerElView('tabs', TabsElView);
global.registerElView('datepicker', DatePickerElView);

```

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

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

[The 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 **tip** and **css** options. The first option sets a tooltip to the wrapped 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 (<i>in the constructor and methods</i>)
isVisible	No	boolean value, determines DOM element visibility on the page
propChangedCommand	No	A command (<i>typically, defined in a view model</i>) for property change notification. For example, The 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. Databindings can set this property, for the error display.

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

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

[TextAreaElView](#) – a wrapper around the `<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 the `<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 the `<input type="radio"/>` element. It exposes [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 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 the internal `invokeCommand` method to trigger the command's action (*typically, when a button or a link is clicked*). It also exposes [isEnabled](#) property.

[ButtonElView](#) - a descendant of the [CommandElView](#). It is a wrapper around the `<button/>` or `<input type="button" />` DOM element. It exposes [value](#), [text](#), [html](#) properties of a button element. It also exposes a boolean [preventDefault](#) property, so to choose between if the button should trigger its default action or not. This element view's command property can be databound 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>
```

P.S.- ***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 the needed elements. It is an alternative to the **name** attribute, because in HTML5 some elements can not have the **name** attribute, but **data-name** can be used universally.*

AnchorEiView - a descendant of the **CommandEiView**. It is a wrapper around the `<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>
```

ExpanderEiView - a descendant of the **AnchorEiView**. 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);
```

TemplateEiView - a descendant of the **CommandEiView**. It is a special element view. It has no other properties besides the inherited from the base type. One property which is important for this element view is the **command** property. This view is used only inside the data templates to subscribe to notifications when the 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 loaded or is starting to unload. This behavior can be used, to access DOM elements (*you can assign some event handlers to them or to add some attributes*) inside the template. The element view is registered by the name **template**.

an example of the data template which uses **TemplateEiView** (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">
    <!--dummy form action to satisfy HTML5 specification-->
    <form data-name="uploadForm" action="#">
        <div data-name="uploadBlock">
            <input data-name="files-to-upload" type="file" style="visibility: hidden;" />
            <div class="input-append">
                <input data-name="files-input" class="span4" type="text">
                <a data-name="btn-input" class="btn btn-info btn-small"><i class="icon-folder-open">
                </i></a><a data-name="btn-load" class="btn btn-info btn-small"
                data-bind="{this.command,to=uploadCommand}"
                data-view="options={tip='Click to upload a file'}">Upload</a>
            </div>
            <span>File info:</span><text>&nbsp;</text>
            <div style="display: inline-block" data-bind="{this.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 the command definition databound to the TemplateElView's command property (see *UploadThumbnailVM in gridDemo.ts*):

//executed when template is loaded or unloading

```

this._templateCommand = new MOD.baseElView.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._onError(ex, this);
  }
}, self, function (sender, param) {
  return true;
});

```

SpanElView - a wrapper around the `` element. It is used to databind some string property (*text or html*) 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 an element. It should not be used to display the user input without first checking the content to prevent XSS attacks!

```

<span data-bind="{this.value,to=testProperty1,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/>`. Besides the properties inherited from the **SpanElView**, it adds **borderColor**, **borderStyle**, **width**, **height** properties, which can be data bound. It is registered also by the name **block**, which can be provided in 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 `` 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, the `div` element*).

It exposes the `isBusy` and the `delay` property.

It is used to display an animated loader gif image above the content of a 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 Collection derived source which is data bound to the `dataSource` property.

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.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}}}">
  <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>
```

PagerEIView - a wrapper around a block HTML DOM element (*typically*, `<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**.

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>
```

StackPanelEIView - a wrapper around a block HTML DOM element (*typically*, `<div/>`). It is used to attach the logic and the markup of the **StackPanel** control to the div element. It exposes the **dataSource** and the **panel** properties. It is used to display horizontally or vertically stacked panels (*which are templated*) on the page. The element view is registered by the name **stackpanel**.

an example of the markup for the StackPanel (*see the CollectionsDemo 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>
```

SelectEIView - 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 gives data to fill select element 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>
```

DataFormEIView - 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 the editing and viewing purposes. The element view is registered by the name **dataform**.

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:multiline,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>
<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 an `<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**.

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 the JQuery UI Tabs plugin logic to the HTML DOM element for managing the content displayed in tabs. It exposes **tabsEventCommand** property. The element view is registered by the name **tabs**.

Note: The **tabsEventCommand** is used to get notifications on tabs events (like when a tab was selected, added, showed, enabled, disabled, removed, loaded) and handle them in the view model.

an example of the 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.tabsEventCommand,to=tabsEventCommand,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-->
            </div>
        </div>
    </div>

```

```

        <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:

```

this._tabsEventCommand = new MOD.mvvm.Command(function (sender, param) {
    var index = param.args.index, tab = param.args.tab, panel = param.args.panel;
    //alert('event: ' + param.eventName + ' was triggered on tab: '+index);
}, self, null);

```

DynaContentElView - a wrapper around a block HTML DOM element (*typically, the <div/>*).

It exposes the **templateID** and the **dataContext** properties.

It is used to mark a block element as a content region which will contain templated 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 the dataContext to the currently displayed template.

The element view is registered by the name **dynacontent**.

an example of the markup for the dynacontent:

```

<div id="demoDynaContent"
data-bind="{this.templateID,to=viewName,source=customerVM.uiMainView}
{this.dataContext,source=customerVM}" 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 - Besides the core element views, it is easy to add a custom element view.

For example, in the demo application there has been added several custom element views, such as **AutoCompleteElView**, **DownloadLinkElView**, **FileImageElView** – they all have been defined in custom 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" }})"/>

```

Note: The BaseElView has a property **propChangedCommand** which needs more explanation.

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

It is invoked when a property on the element view is changed. So in an action for this command you can get reference to the element view instance and to the name of the changed property.

For example, it can be used, to obtain references to the instance of the datagrid control when a datagrid control instance is created in the element view. Then if you have this instance you can attach event handlers to it and handle these events in the view model.

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

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

//our product's view model defines handler for the command
//we can obtain datagrid instance from the sender - (the sender is the element view - GridElView)
this._propChangeCommand = new MOD.baseElView.PropChangedCommand(function (sender, data) {
    if (data.property == '*' || data.property == 'grid') {
        if (self._dataGrid === sender.grid)
            return;
        self._dataGrid = sender.grid;
    }
    //example of binding to dataGrid events
    if (!!self._dataGrid) {
        self._dataGrid.addOnPageChanged(function (s, a) {
            self._onGridPageChanged();
        }, self.uniqueID);
        self._dataGrid.addOnRowSelected(function (s, a) {
            self._onGridRowSelected(a.row);
        }, self.uniqueID);
        self._dataGrid.addOnRowExpanded(function (s, a) {
            self._onGridRowExpanded(a.old_expandedRow, a.expandedRow, a.isExpanded);
        }, self.uniqueID);
        self._dataGrid.addOnRowStateChanged(function (s, a) {
            if (!a.val) {
                a.css = 'rowInactive';
            }
        }, self.uniqueID);
    }
}, self, null);
```

2.7 Data templates

Data templates are pieces of the HTML markup which can be used by the UI controls for cloning their structure and displaying them the page.

The framework contains several built-in controls, which are aware of the data template: [DataEditDialog](#), [DataGrid](#), [StackPanel](#), [DynaContentElView](#).

The data templates definition (*markup*) must have the `id` attribute for referencing them in the options of the UI controls. The data template aware controls create instances of the data templates and then add the newly created data template's instance to the HTML DOM tree, and can usually set the template's `dataContext` property. The data template's `dataContext` property can be set or reset at any time on the data template's instance.

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

```
_createTemplate(dcxt) {
    var t = new template.Template(this._app, this._templateID);
    //you can set it to the disabled state
    t.isDisabled = true;
    //set template's data context
    t.dataContext = dcxt;
    //return instance
    return t;
}
```

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 loader function*)
- 4) Register a group of templates for loading them on as needed basis (*but they will be loaded as groups of templates*)

The first way - defining them on the page

The templates are defined in a special section on the html page (*but not necessarily one section, there can be several of 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 data-role attribute with the value "template". The templates loaded by this method available to all application's instances (*so they are in the global scope*).

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 file. The rules for the templates definition are the same as in the first way, but the file contains just the templates definition (*no section tag around them*). They are loaded by the application's `loadTemplates` method. (See *DataGrio Demo* for an example). You must invoke the loading before calling 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 (But in the ASP.NET MVC you can do this using partial views). Another difference is that every 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 the template in the form of a html string. The loader function is agnostic of the way of obtaining the template definition, you need only to return 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 excessive network traffic.

Note: *This method of loading of templates is not very efficient (if 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 a loading templates by registering the 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 the 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 every group of templates you register unique group's name and also the names of templates the group includes. The group registration is done before application's **startUp** method is invoked. When the application will need a template, then the whole group (*containing that template*) will be loaded from the server (*See 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 a SPA usually displays many different screen views while not reloading the page. So, it is good to define groups of the templates per each screen view. Groups will be loaded when they are needed and only when they are needed.*

an example of a 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

A **data content** is used for displaying specific content types in a predetermined way. For example, a boolean value can be displayed as a checkbox on the page and a string value as a textbox (*using the input element*). Also these values can have different display if they are not in editing state - a string value can be displayed as a text inside a span element.

This is the exact situation which a data content handles.

The data-content attribute can be used only inside the data forms or the data grids (*to define cell's content*).

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

an example of using data contents in the 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 can be two types of the data content :

- 1) Those which bind directly to a field name.
- 2) Those which use the data templates.

The first option (*bind to a field directly*):

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

The type (*class*) for the creation of an 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 tweaked by explicitly specifying the name of the data content and some data contents may also need the options for their normal initialization.

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

```

<span data-content="fieldName:Name,css:{displayCss:'name-display',editCss:'name-edit'},name:multiline,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 option 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 a template can have more complex display. Inside the template you can use databindings to several fields
The only drawback here - is that it needs more efforts than the first option.

The templated data content is defined by the [TemplateContent](#) type.

an example of markuop for a templated data content:

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

Note: *editID or displayID can be omitted if you need only to display content in one state.*

2.9 Controls

DataGrid:

The [DataGrid](#) is a control for attaching logic to a table HTML element. Without this control the table's content is static. With this UI control the table is turned into the 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 the pager control*)
- 4) Sorting the data on column clicking
- 5) Specialized column types (*like row selector, row actions, row expander*)
- 6) Usage of templates for the data display and editing
- 7) Support for visual row state (*its display*) based on a field's value
- 8) Column headers are fixed like in desktop data grids
- 9) Support for use of the keyboard keys (*up, down, left, right, space*)
- 10) Row selection (*when row selector column is present*) by space key
- 11) Navigation between pages using pageup or pagedown keys.

The DataGrid – is defined in the [datagrid](#) module. 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 declarative use of the control, there's a supplementing element view - [GridElView](#).

[an example of the 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>

```

The columns in the datagrid are defined by adding to the `<th />` tag `data-column` and `data-content` attributes.

A 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" />

```

A `type` option determines which is the type of the data column. If the `type` option is omitted, then it has the default type, which is 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 and specifies the data (*a field value*) 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 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. Also, there can be the need to handle datagrid's events in the view model's code, so you can databind

`propChangedCommand` and then in the command's action you can access the instance of the grid element view, and can add event handlers for the datagrid's events.

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

```
//our product's view model defines handler for the command
//we can obtain datagrid instance from the sender - (the sender is the element view - GridElView)
this._propChangeCommand = new MOD.baseElView.PropChangedCommand(function (sender, data) {
    if (data.property == '*' || data.property == 'grid') {
        if (self._dataGrid === sender.grid)
            return;
        self._dataGrid = sender.grid;
    }
    //example of binding to dataGrid events
    if (!!self._dataGrid) {
        self._dataGrid.addOnPageChanged(function (s, a) {
            self._onGridPageChanged();
        }, self.uniqueID);
        self._dataGrid.addOnRowSelected(function (s, a) {
            self._onGridRowSelected(a.row);
        }, self.uniqueID);
        self._dataGrid.addOnRowExpanded(function (s, a) {
            self._onGridRowExpanded(a.old_expandedRow, a.expandedRow, a.isExpanded);
        }, self.uniqueID);
        self._dataGrid.addOnRowStateChanged(function (s, a) {
            if (!a.val) {
                a.css = 'rowInactive';
            }
        }, self.uniqueID);
    }
}, self, null);
```

The other important thing, which is used in the declarative data grid definition, is the `data-view` attribute which allows to provide the 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}}"
```

One important option is the table's wrap style, using this, you can add the vertical scrolling for the table's data (*the table's body*), and allow the table's header be fixed even when the table's body is scrolled up or down.

an example of an overall markup for the 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>
```

```

<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>

```

DataGrid's options 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, on which value's change the grid's row_state_changed event is invoked. When this event is handled the field's value can be inspected, and based on that value can be selected the css style for the row display.

isCanEdit and **isCanDelete** - if the options values are false. Then the grid will never be in the edit state and it 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.

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 a DataGrid on the 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		<input checked="" type="checkbox"/>	Size:
	CA-1098	AWC Logo Cap		Caps	01.07.2001		<input checked="" type="checkbox"/>	Size:

Tab 1

Tab 2



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

Upload product thumbnail

				CL-9009	Bike Wash - Dissolver		Cleaners	01.07.2003		<input checked="" type="checkbox"/>	Size:
				LO-C400	Cable Lock		Locks	01.07.2002	30.06.2003	<input type="checkbox"/>	Size:
				CH-0234	Chainus		Chains	01.07.2003		<input checked="" type="checkbox"/>	Size:
				VE-C304-L	Classic Vest, L3		Jerseys	01.07.2003		<input checked="" type="checkbox"/>	Size: M
				VE-C304-M	Classic Vest, M		Vests	01.07.2003		<input checked="" type="checkbox"/>	Size: M
				VE-C304-S	Classic Vest, S		Vests	01.07.2003		<input checked="" type="checkbox"/>	Size: S
				FE-6654	Fender Set - Mountain2		Fenders	01.07.2003		<input checked="" type="checkbox"/>	Size:
				FB-9873	Front Brakes	317,00	Brakes	01.07.2003		<input checked="" type="checkbox"/>	Size:
				FD-2342	Front Derailleur	88,00	Derailleurs	01.07.2003		<input checked="" type="checkbox"/>	Size:
				GL-F410-M	Full-Finger-Gloves-M		Gloves	01.07.2002	30.06.2003	<input type="checkbox"/>	Size: M
				GL-F410-S	Full-Finger-Gloves-S		Gloves	01.07.2002	30.06.2003	<input type="checkbox"/>	Size: S

1 2 3 4 5 6 >>

Total: 293, Selected: 0

+ New Product

An example of several DataGrids on the 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

Agcaoili N. Samuel

Ahlerin E. Robert

Ahlerin E. Robert

Alan A. Stanley

Alan A. Stanley

Alberts E. Amy

Alberts E. Amy

Alcorn L. Paul

Alcorn L. Paul

Alderson F. Gregory

Alderson F. Gregory

Alexander Mary

Alexander Mary

Alexander Michelle

Alexander Michelle

Allen N. Marvin

Allen N. Marvin

Allison J. Cecil

Allison J. Cecil

Alpuerto L. Oscar

Alpuerto L. Oscar

Amland J. Maxwell

Amland J. Maxwell

Antrim J. Ramona

Antrim J. Ramona

Armstrong B. Thomas

Customer Info

ID: 582

Title: Ms.

FirstName: Catherine

MiddleName: R.

LastName: Abel

Suffix:

CompanyName: Professional Sales and Service

SalesPerson: adventure-works\Linda3

Email: catherine0@adventure-works.com

Phone: 747-555-0171

+ New Customer

Customer Addresses

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

Manage Addresses

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

An example of datagrid's 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:

Refresh Ok Cancel

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 grid's options the

name of the field to observe, as in [rowStateField.isActive](#), then add handler to the grid's instance to change css style of the row depending on the value.

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

DataPager:

A [Data Pager](#) is a separate control. Like all the controls defined in the framework, in order to use it declaratively, it has a special element view - the [PagerElView](#).

In order to display it on the page, it is needed to add a markup for that, as in the example:

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

DataEditDialog:

A [DataEditDialog](#) is a control used to display modal popup dialogs. This control is used by the DataGrid control to display edit dialogs. But this control can also be used independently from the DataGrid.

The DataEditDialog uses a data template for its visual display. 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 special view model:

```
export class DialogVM extends MOD.mvvm.BaseViewModel {  
    _dialogs: { [name: string]: () => MOD.datadialog.DataEditDialog; };  
  
    constructor(app: Application) {  
        super(app);  
        this._dialogs = {};  
    }  
  
    createDialog(name: string, options: MOD.datadialog.IDialogConstructorOptions) {  
        var self = this;  
        this._dialogs[name] = function () {  
            var dialog = new MOD.datadialog.DataEditDialog(self.app, options);  
            var f = function () {  
                return dialog;  
            };  
        };  
    }  
}
```

```

    };
    self._dialogs[name] = f;
    return f();
  };
  return this._dialogs[name];
}
showDialog(name:string, dataContext) {
  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._dialogs[name];
  if (!factory)
    return null;
  return factory();
}
destroy() {
  if (this._isDestroyed)
    return;
  this._isDestroyCalled = true;
  var keys = Object.keys(this._dialogs);
  keys.forEach(function (key:string) {
    this._dialogs[key].destroy();
  }, this);
  this._dialogs = {};
  super.destroy();
}
}

```

Then, it simplifies creation of 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 a dialog is only needed to invoke the DialogVM's `showDialog` method. The method expects two parameters: the name of the dialog, and the data context.

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

The option's property `fn_OnTemplateCreated` needs more explanation. This option's property is used to provide a function which will be invoked when an instance of the data template used by the dialog is created. By using that 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>

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);
        }
    }
};
this._dialogVM.createDialog('folderBrowser', dialogOptions);

this._dialogCommand = new MOD.mvvm.Command(function (sender, param) {
    try {
        self._dialogVM.showDialog('folderBrowser', self);
    } catch (ex) {
        self._onError(ex, this);
    }
}, self, function (sender, param) {
    return true;
});

```

The dialog also has the next options:

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 (*you can see how it was done in ManyToMany demo's view model*).

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 *MOD.utils.ISubmittable* interface.

an example of a more complex dialog:

add new customer address

Customer: **Abel R. Catherine**

Search existing Address:

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

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

+ New Address

1 2 >> Total: 94

Ok

Cancel

DataForm:

A DataForm is a control that attaches a data context to the 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 allows to use the data contents inside it. They can display the data in editing state and not editing state differently. Also the data form automatically displays summary of validation errors.

The DataForm – 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 - the `DataFormElView`. It is registered by the name `dataform`.

an example of the 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:

A **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 (*typically, the <div/> tag*). Each object from the collection databound to a 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 or 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 the page:

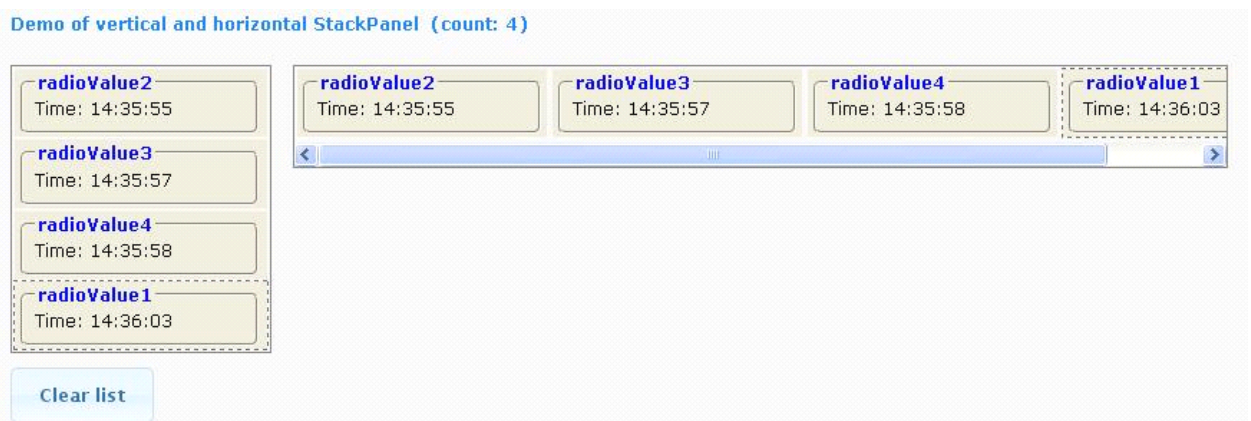
```

<!--example of using stackpanel for vertical and horizontal list view-->
<div style="border: 1px solid gray;float:left;width:180px; min-height:50px; max-height:250px; overflow:auto;"
data-bind="{this.dataSource,to=historyList,source=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 the page:



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

```

<div class="ria-stackpanel"
data-view="name=stackpanel,options:{templateID:stackPanellItemTemplate,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-eltkey0="s_10">
  <div class="stackpanel-item" style="display: inline-block;" data-key="clkey_0">
    <div class="stackPanellItem" style="width: 170px;">
      <fieldset>
        <legend><span data-eltkey0="s_14">radioValue2</span> </legend>Time:&nbsp;<span
          data-eltkey0="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:

A **ListBox** - is a control which is used to attach the logic of a combobox to a `<select/>` tag .

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

The ListBox control is defined in the listbox.ts file. In order to use the control declaratively there is a special element view the **SelectElView**.

This control is also used internally in the **LookupContent** to display lookup fields in editing states.

an example of usage of two ListBoxes on the page:

```

<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>

```

Working with the data on the client side

3.1 Working with the simple collection's data

The framework's collection hierarchy starts from `BaseCollection<TItem extends CollectionItem>` type (*defined in the collection.ts file*), which is a base abstract type for all specialized collections. The **CollectionItem** is the base type for all the item types in these collections. These base classes are defined in the *collection* module, which has also **ListItem**, **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 the events which notify the controls about the changing current position, adding or removing of a collection item, and also about the start and the end of the editing of the item. Only one item in the collection can be in the editing state.

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

Collection's properties:

Property	Is readOnly	Description
permissions	Yes	Exposes the permissions for the updating, inserting and refreshing of the entities in the collection. export interface IPermissions { canAddRow: boolean ; canEditRow: boolean ; canDeleteRow: boolean ; canRefreshRow: boolean ; }
currentItem	No	Current item
count	Yes	Number of the items in the collection
totalCount	No	Total number of the items. Used for the paging support.
pageSize	No	Size of the data page. Used for the paging support.
pageIndex	No	Current page index. Zero based value.
items	Yes	Array of the collection items.
isPagingEnabled	Yes	If true then the collection supports 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 the bulk updates. Used when it is needed to update the items without triggering begin_edit and end_edit events. Which prevents UI controls flickering.
pageCount	Yes	The calculated number of the pages (<i>if the paging is supported</i>)
options	Yes	Exposes the options object, which is created in the constructor. export interface ICollectionOptions { enablePaging: boolean ; pageSize: number ; }

Collection's methods:

Method	Description
getFieldInfo	Returns information about a field by providing a field's name. export interface IFieldInfo { isPrimaryKey: number ; isRowTimeStamp: boolean ; dataType: number ; isNullable: boolean ; maxLength: number ; isReadOnly: boolean ; isAutoGenerated: boolean ;

	<pre> allowClientDefault: boolean; dateConversion: number; isClientOnly: boolean; isCalculated: boolean; isNeedOriginal: boolean; dependentOn: string; range: string; regex: string; isNavigation: boolean; fieldName: string; dependents?: string[]; } </pre>
getFieldNames	Returns an array of the field names.
cancelEdit	Cancels the editing
endEdit	Ends the editing if there are no validation errors, otherwise cancels the editing.
getItemsWithErrors	Returns an array of the items which have validation errors.
addNew	Creates and returns the new item. Leaving the collection in the editing state. You can cancel adding the new item by invoking cancelEdit, or otherwise you can commit it with the endEdit method.
getItemByPos	Returns item (<i>if found</i>) by position or null value.
getItemByKey	Returns an item by the key (or null). The key value is the CollectionItem's property _key which has string type. The Dictionary uses one property on the items as a key value. The DbSet's items are returned from the data service layer and they have the server side generated key (<i>string values of the primary key concantenated by ;</i>). If the item is created on the client side before it is submitted to the server, the _key property contains client side generated value.
findByPK	Returns the item by primary key values (or null). Primary key values supplied 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 (<i>and not submitted</i>) 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 (<i>and not submitted</i>) 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 (<i>and not submitted</i>) 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 (<i>and not submitted</i>) items should be skipped.
goTo	Moves the current item position to the provided in the method argument. Returns a boolean value indicating if the move was successful.
forEach	The same functionality as the array's forEach method.
removeItem	Immediately removes the item from the collection.

sortLocal	Sorts the collection items locally on the client. First parameter must be the array of field names, the second 'ASC' or 'DESC'.
sortLocalByFunc	Sorts the collection items locally on the client. Expects a sorting function for the items.
clear	Removes all the items from the collection.
waitForNotLoading	A utility method which allows to wait for a callback function. It is executed only when the collection is not loading the data, otherwise it waits till that moment. The call is nonblocking.

Collection's events:

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

CollectionItem's properties:

Property	Is readOnly	Description
_isNew	Yes	Returns true if the item is newly created by the addNew method on the collection and before the changes are committed to the server.
_isDeleted	Yes	Returns true if the item in the deleted state.
_key	False	Returns the key (<i>a string value</i>) value of the item in the collection.
_collection	Yes	Returns parent collection. (<i>DbSet, List, Dictionary</i>)
_isUpdating	Yes	Returns true if the collection in the updating state.
isEditing	Yes	<p>Returns true if the item in the editing state.</p> <p>It is part of implementation of the IEditable interface:</p> <pre>export interface IEditable { beginEdit(): boolean; endEdit(): boolean; cancelEdit(): boolean; isEditing: boolean; };</pre>

<code>_isCanSubmit</code>	Yes	Returns true if the item supports submitting changes to the server. It is part of implementation of the <code>ISubmittable</code> interface: <pre>export interface ISubmittable { submitChanges(): IVoidPromise; _isCanSubmit: boolean; }</pre>
<code>_changeType</code>	Yes	Returns a status of changes for the item <pre>export enum STATUS { NONE= 0, ADDED= 1, UPDATED= 2, DELETED= 3 }</pre>

CollectionItem's methods:

Property	Description
<code>getFieldInfo</code>	Returns <code>IFieldInfo</code> value by the name of the field.
<code>getFieldNames</code>	Returns an array of field names.
<code>beginEdit</code>	Starts items's editing. Returns true if the editing started successfully.
<code>endEdit</code>	Ends (commits) items's editing. Returns true if the editing ended successfully.
<code>cancelEdit</code>	Cancels current items's editing. Returns true if the editing canceled successfully.
<code>deleteItem</code>	Deletes the item. Returns true if the item is really deleted.
<code>addOnItemErrorsChanged</code>	Adds an event handler for the errors change event
<code>getFieldErrors</code>	Returns an array of <code>IVValidationInfo</code> for the field by field's name. If field's name is asterisk * , then it returns result of the item's validation. <pre>export interface IValidationInfo { fieldName: string; errors: string[]; }</pre>
<code>getAllErrors</code>	Returns an array of <code>IVValidationInfo</code> for the item.
<code>getErrorString</code>	Returns errors in the string form.
<code>submitChanges</code>	Submits changes to the server if the collection supports it.
<code>getIsHasErrors</code>	Returns true if the item has any validation errors.

CollectionItem's events:

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

The descendants of the collection type:

`BaseList` – is a simple collection of `ListItem` types.

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

There are also two collections the `List` and the `Dictionary` which have resolved generics arguments. Their items are untyped (*has any type*). They exist for the cases when the `DataService's GetTypeScript` method was not used to generate strongly typed collections definitions from server side types. But this is a rare case. They are defined as:

```

export class List extends BaseList<ListItem, any> {
    constructor(type_name: string, properties: any) {
        var props: IPropInfo[] = getPropInfos(properties);
        var fieldNames: string[] = props.map(function (p) { return p.name; });
        var itemType = getItemType(fieldNames);
        super(itemType, props);
        this._type_name = type_name;
    }
}

export class Dictionary extends BaseDictionary<ListItem, any> {
    constructor(type_name: string, properties: any, keyName: string) {
        var props: IPropInfo[] = getPropInfos(properties);
        var fieldNames: string[] = props.map(function (p) { return p.name; });
        var itemType = getItemType(fieldNames);
        super(itemType, keyName, props);
        this._type_name = type_name;
    }
    _getNewKey(item: ListItem) {
        if (!item) {
            return super._getNewKey(null);
        }
        var key = item[this._keyName];
        if (utils.check.isNt(key))
            throw new Error(utils.format(RIAPP.ERRORS.ERR_DICTKEY_IS_EMPTY, this._keyName));
        return " " + key;
    }
}

```

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

```

//one property in a dictionary must be unique and used as key
this._radioValues = new MOD.collection.Dictionary('RadioValueType', ['key', 'value', 'comment'], 'key');
this._radioValues.fillItems([
    { key: 'radioValue1', value: 'This is some text value #1', comment: 'This is some comment for value #1' },
    { key: 'radioValue2', value: 'This is some text value #2', comment: 'This is some comment for value #2' },
    { key: 'radioValue3', value: 'This is some text value #3', comment: 'This is some comment for value #3' },
    { key: 'radioValue4', value: 'This is some text value #4', comment: 'This is some comment for value #4' }], false);

```

But a better way is to use strongly typed collections generated by DataService's [GetTypeScript](#) method. They don't need any constructor arguments, and therefore it is more simple to create their instances. Their [fillItems](#) method checks at the compile time that you provide values of the expected type.

an example of initialization of a strongly typed Dictionary:

```

this._orderStatuses = new DEMODB.KeyValDictionary();
this._orderStatuses.fillItems([
    { key: 0, val: 'New Order' }, { key: 1, val: 'Status 1' },
    { key: 2, val: 'Status 2' }, { key: 3, val: 'Status 3' },
    { key: 4, val: 'Status 4' }, { key: 5, val: 'Completed Order' }], true);

```

When the [fillItems](#) method is used on the above dictionary it will expect that you will provide an array of values of IKeyVal type.

3.2 Working with the data provided by the DataService

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 interaction between the server and the client side. For the server side there is the [DataService](#) 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 the need to make those updates in the order of the relationship between the entities. The entities can also have autogenerated fields which values should be propagated back to the clients with the results of the operations. The server and client sides must have the metadata which describes entities, relationship between them. It is also used for validation purposes.

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

The DbContext:

An instance of the [DbContext](#) is used to interact with the data service. The DbContext stores the data in the collections with their the type derived from the DbSet.

The DbContext prevents repeated loading of the entities in the DbSet. If the entity is loaded twice then it not replaces the entity in the collection but only refreshes its data. The DbContext checks entities' equality by comparing their keys. Each entity in the DbSet must have the unique key (*primary key*).

DbContext's properties:

Property	Is readonly	Description
isBusy	Yes	Boolean property, which indicates that the DbContext doing some work (<i>typically asynchronous</i>).
isSubmitting	Yes	Boolean property, which indicates that the DbContext submits updates to the service end.
serverTimezone	Yes	The timezone of the server from which the application was loaded.
dbSets	Yes	A special object which contains all the DbSet for the DbContext.
serviceMethods	Yes	A map (<i>indexed by names</i>) of the service methods (<i>methods exposed from the data service</i>). Service method invocation is an asynchronous operation. It returns a promise which will be resolved with the data returned from the method, or rejected if the invocation failed.
hasChanges	Yes	Boolean property which indicates that the DbContext has pending changes (<i>not submitted</i>).
service_url	Yes	Returns the data service's url.
isInitialized	Yes	Returns true when the DbContext is initialized with the metadata.

DbContext's methods:

Property	Description
getDbSet	Returns a DbSet by its name.
submitChanges	Submits the changes to the service. It is the asynchronous

	operation and returns a promise.
load	Loads the data from the data service. It is the asynchronous operation and accepts a query object. It returns a promise <code>IPromise<ILoadResult<Entity>></code> .
acceptChanges	Accepts all the changes turning all entities' statuses to <code>STATUS.None</code> . Typically it is automatically invoked when the <code>DbContext</code> successfully submits the changes.
rejectChanges	Rejects all the changes turning all entities' statuses to <code>STATUS.None</code> and the values are restored to the original ones.
waitForNotBusy	Used internally to wait for all asynchronous operations to complete.
waitForNotSubmitting	Used internally to wait for a submit to complete.
initialize	Initializes the <code>DbContext</code> , with the metadata and the service url.
getAssociation	Returns a parent-child association object instance by its name.

DbContext's events:

Event	Description
submit_error	Raised when a submit of the changes is not successful. It allows to handle the submit error without rejecting all the changes.

The `DbContext` which is defined in the `db.ts` file is generally not used directly. Instead it is used as a class derived from this original `DbContext`. The `DataService's GetTypeScript` method creates a script with strongly typed entity classes and a strongly typed `DbContext`. It exposes strongly typed `dbSets`, `serviceMethods`, and associations properties.

an example of a derived DbContext class:

```
export class DbContext extends RIAPP.MOD.db.DbContext {
  _initDbSets() {
    super._initDbSets();
    this._dbSets = new DbSets(this);
    var associations = [a number of association's infos here];
    this._initAssociations(associations);
    var methods = [a number of method's infos 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 a strongly typed DbContext creation:

```
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:

```
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;
        query.loadPageCount = 20; //load 20 pages at once (but only one will be visible, the others will be in the
local cache)
        query.isClearCacheOnEveryLoad = true; //clear the local cache when a new batch of data is loaded from
the server
        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.DISC);
        return query.load();
    }
}

```

Where 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 create 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 selection of the data.

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

The DbContext class also allows to execute special 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 included*) and can return result (*can be also of complex type*). The DataService's *GetTypeScript* method generates strongly typed versions of those methods, which are easy to use from the client 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._dialogVM.showDialog('testDialog', self);  
});  
  
promise.fail(function () {  
    //do something on fail if you need  
    //but the error message display is automatically shown  
});
```

DataCache:

The DataCache is used to cache the data on the client. You can set the query's `loadPageCount` property to a 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 data*).

Those extra pages of the data rows are cached inside the query's instance with the help of **internally** used a `DataCache` class instance.

If the `DbSet`'s `pageIndex` property value is changed (*going to another 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 (it is often for complex queries), and if navigation from one page to the next takes considerable time, then it is better to preload several pages of the data to the client in one query operation.*

DataQuery:

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

DataQuery's properties:

Property	Is readonly	Description
<code>loadPageCount</code>	No	It determines, how many pages to load. If the <code>pageSize</code> is 100 and <code>loadPageCount</code> is 25

		then the loading will try to load 2500 records from the server.
<code>isClearCacheOnEveryLoad</code>	No	It determines if the cached data is cleared when the DbContext's load method is called explicitly. If you will set it to true then the already cached data would be cleared. If you will set to false, then the new data would be appended to the previous data.
<code>isIncludeTotalCount</code>	No	It determines if the query will try to return the total count of the records.
<code>params</code>	No	It used to set the query parameters if the data service query method expects arguments.

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

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

```
export class CustomerDb extends RIAPP.MOD.db.DbSet<Customer>
{
    constructor(dbContext: DbContext) {
        var self = this, opts: RIAPP.MOD.db.IDbSetConstructorOptions = {
            dbContext: dbContext,
            dbSetInfo: { a dbSet info here }
        }, utils = RIAPP.global.utils;
        super(opts);
        self._entityType = Customer;

        opts.dbSetInfo.fieldInfos.forEach(function (f) {
            f.dependents = [];
            self._fieldMap[f.fieldName] = f;
        });

        opts.dbSetInfo.fieldInfos.forEach(function (f) {
            if (!f.isNavigation) {
                self._navfldMap[f.fieldName] = self._doNavigationField(opts, f);
            }
            else if (!f.isCalculated) {
                self._calcfldMap[f.fieldName] = self._doCalculatedField(opts, f);
            }
        });

        self._mapAssocFields();
    }
    createReadCustomerQuery(args?: {
        includeNav?: boolean;
    }) {
        var query = this.createQuery('ReadCustomer');
        query.params = args;
        return query;
    }

    defineNameField(getFunc: () => string) { this.defineCalculatedField('Name', getFunc); }

    get items2() { return <ICustomerEntity[]>this.items; }
}
```

DbSet:

A [DbSet](#) is derived from a Collection class so it supports all its methods and properties. But it is usually used to store items (*entities*) loaded from the server.

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 when a HTML page is loaded. It reduces a number of data roundtrips 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 the base Collection type:

DbSet's specific methods:

Property	Description
fillItems	Loads the DbSet with locally stored data. (<i>Useful for lookups</i>)
acceptChanges	Accepts pending changes.
rejectChanges	Rejects pending changes.
deleteOnSubmit	Deletes an entity on submit.
createQuery	Creates an instance of the DataQuery's (<i>by query name</i>).
clearCache	Explicitly clears local cache.
defineCalculatedField	Defines calculated field. You need to provide the field name and the function which performs calculations.

DbSet's specific properties:

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

The DataService's *GetTypeScript* method produces the script which contains all the DbSets which are exposed by the DataService. Those DbSets are strongly typed and if the DbSet contains a calculated field (*its name is defined in the metadata*) then the strongly typed DbSet will have a special 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>
{
    constructor(dbContext: DbContext) {
        var self = this, opts: RIAPP.MOD.db.IDbSetConstructorOptions = {
            dbContext: dbContext,
            dbSetInfo: { a dbSet info here },
            childAssoc: [associations infos here],
            parentAssoc: [ associations infos here]
        }, utils = RIAPP.global.utils;
        super(opts);
        self._entityType = Registr;

        opts.dbSetInfo.fieldInfos.forEach(function (f) {
            f.dependents = [];
            self._fieldMap[f.fieldName] = f;
        });

        opts.dbSetInfo.fieldInfos.forEach(function (f) {
            if (!f.isNavigation) {
                self._navfldMap[f.fieldName] = self._doNavigationField(opts, f);
            }
            else if (!f.isCalculated) {
                self._calcfldMap[f.fieldName] = self._doCalculatedField(opts, f);
            }
        });

        self._mapAssocFields();
    }

    //two query methods with different arguments
    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;
    }
    //a set of different methods to define calculated fields

    defineLPUField(getFunc: () => string) { this.defineCalculatedField('LPU', getFunc); }
    defineDTYPEField(getFunc: () => string) { this.defineCalculatedField('DTYPE', getFunc); }
    defineFIOField(getFunc: () => string) { this.defineCalculatedField('FIO', getFunc); }
    defineS_OPLField(getFunc: () => number) { this.defineCalculatedField('S_OPL', getFunc); }
    defineSMOField(getFunc: () => string) { this.defineCalculatedField('SMO', getFunc); }
    defineADDRESSField(getFunc: () => string) { this.defineCalculatedField('ADDRESS', getFunc); }
    defineerrorsField(getFunc: () => any) { this.defineCalculatedField('errors', getFunc); }

    get items2() { return <IRegistrEntity[]>this.items; }
}

```

Calculated fields (*if present*) must be defined after the DbContext's initialize method had been invoked, and before you load any 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(){
    return self.dbContext.associations.getErrorToRegistr().getChildItems(this);
});

this._dbContext.dbSets.Registr.defineFIOField(function () {
    //this function is executed in the context of the entity, so 'this' refers to the entity
    return this.FAM + " " + this.IM + " " + this.OT;
});

this._dbContext.dbSets.Registr.defineDTYPEField(function () {
    var res = filter.cls.dtypeRDict[this.D_TYPE_R];
    return !!res ? res.v : this.D_TYPE_R;
});

this._dbContext.dbSets.Registr.defineLPUField(function () {
    var res = filter.cls.lpuDict[this.MCOD];
    return !!res ? res.v : null;
});

this._dbContext.dbSets.Registr.defineS_OPLField(function () {
    var errs = this.errors, sall = this.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 () {
    var res = filter.cls.smoDict[this.Q];
    return !!res ? res.v : this.Q;
});

```

Entities:

An **Entity** is a base class for the derived entity classes.

Basically, the Entity is a collection item which is specific for the DbSet. 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, the **Demo** - in the file

RIAppDemo.BLL\DataServices\RIAppDemoMetadata.xml

This is a **xml** 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 the 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 is used to generate strongly typed classes. Exactly using this information the DataService's GetTypeScript method generates entities and strongly typed DbSet classes.

an example of a DbSet's schema definition:

```

<data:DbSetInfo dbSetName="Customer" insertDataMethod="Insert{0}" updateDataMethod="Update{0}"
deleteDataMethod="Delete{0}" refreshDataMethod="Refresh{0}" enablePaging="True" pageSize="25"

```

```

EntityType="{x:Type dal:Customer}">
  <data:DbSetInfo.fieldInfos>
    <data:FieldInfo fieldName="CustomerID" dataType="Integer" maxLength="4" nullable="False"
isAutoGenerated="True" readOnly="True" isRowTimeStamp="False" isNeedOriginal="True" primaryKey="1" />
    <data:FieldInfo fieldName="NameStyle" dataType="Bool" maxLength="1" nullable="False"
isAutoGenerated="False" readOnly="False" isRowTimeStamp="False" isNeedOriginal="True" />
    <data:FieldInfo fieldName="Title" dataType="String" maxLength="8" nullable="True"
isAutoGenerated="False" readOnly="False" isRowTimeStamp="False" isNeedOriginal="True" />
    <data:FieldInfo fieldName="FirstName" dataType="String" maxLength="50" nullable="False"
isAutoGenerated="False" readOnly="False" isRowTimeStamp="False" isNeedOriginal="True" />
    <data:FieldInfo fieldName="MiddleName" dataType="String" maxLength="50" nullable="True"
isAutoGenerated="False" readOnly="False" isRowTimeStamp="False" isNeedOriginal="True" />
    <data:FieldInfo fieldName="LastName" dataType="String" maxLength="50" nullable="False"
isAutoGenerated="False" readOnly="False" isRowTimeStamp="False" isNeedOriginal="True" />
    <data:FieldInfo fieldName="Suffix" dataType="String" maxLength="10" nullable="True"
isAutoGenerated="False" readOnly="False" isRowTimeStamp="False" isNeedOriginal="True" />
    <data:FieldInfo fieldName="CompanyName" dataType="String" maxLength="128"
nullable="True" isAutoGenerated="False" readOnly="False" isRowTimeStamp="False" isNeedOriginal="True" />
    <data:FieldInfo fieldName="SalesPerson" dataType="String" maxLength="256" nullable="True"
isAutoGenerated="False" readOnly="False" isRowTimeStamp="False" isNeedOriginal="True" />
    <data:FieldInfo fieldName="EmailAddress" dataType="String" maxLength="50" nullable="True"
regex="^[a-z0-9-]+(\\.[a-z0-9-]+)*@[a-z0-9-]+(\\.[a-z0-9-]+)*(\\.[a-z]{2,4})$" isReadOnly="False"
isRowTimeStamp="False" isNeedOriginal="True" />
    <data:FieldInfo fieldName="Phone" dataType="String" maxLength="25" nullable="True"
isAutoGenerated="False" readOnly="False" isRowTimeStamp="False" isNeedOriginal="True" />
    <data:FieldInfo fieldName="PasswordHash" dataType="String" maxLength="128"
nullable="False" isAutoGenerated="True" readOnly="True" isRowTimeStamp="False" isNeedOriginal="True" />
    <data:FieldInfo fieldName="PasswordSalt" dataType="String" maxLength="10" nullable="False"
isAutoGenerated="True" readOnly="True" isRowTimeStamp="False" isNeedOriginal="True" />
    <data:FieldInfo fieldName="rowguid" dataType="Guid" maxLength="16" nullable="False"
isAutoGenerated="True" readOnly="True" isRowTimeStamp="True" isNeedOriginal="True" />
    <data:FieldInfo fieldName="ModifiedDate" dataType="DateTime" maxLength="8"
nullable="False" isAutoGenerated="True" readOnly="True" isRowTimeStamp="False" isNeedOriginal="True" />
    <data:FieldInfo fieldName="Name" dataType="String" isCalculated="True"
dependentOn="FirstName,MiddleName,LastName" />
  </data:DbSetInfo.fieldInfos>
</data:DbSetInfo>

```

an example of an association definition:

```

<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 Entity type specific methods (besides inherited from CollectionItem):

Property	Description
deleteOnSubmit	Marks the item for deletion and the item status (<i>changeType</i>) is set to deleted.
deleteItem	The same as deleteOnSubmit. The methods are semantically equivalent.
getDbContext	Returns the DbContext's instance.
getDbSet	Returns the DbSet's instance.
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 original.

The Entity type specific properties (besides inherited from CollectionItem):

Property	is readonly	Description
_dbSetName	Yes	Returns the name of the parent DbSet.
_changeType	Yes	Returns the status of the entity. <code>export enum STATUS { NONE= 0, ADDED= 1, UPDATED= 2, DELETED= 3 }</code>
_serverTimezone	Yes	Returns the time zone of the server.
_isRefreshing	Yes	Returns true if the entity 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.
_srvKey	Yes	Returns the key of the entity which is obtained from the server (<i>primary key values concatenated using separator ;</i>).

When a new entity is added, it exists in the editing state. To commit any modifications an `endEdit` method should be called. To discard the modifications and undo adding the entity a `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
item.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 the editing state, then its `beginEdit` method is called internally.

On every call to the `beginEdit` or `endEdit` method, if it completes successfully, will be raised '`begin_edit`' or '`end_edit`' events. In order to prevent triggering those events, for example, when we want to mass update DbSet's items, we 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.pcounts=[]; //sets the entity's field
    });
}
finally
{
    self._dbSet.isUpdating = false; //mark the update ended
}

```

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

Calculated fields:

The calculated fields are just what they are - they calculate their values. They must not have circular references. The calculated fields are read only. They can depend on the other fields (*calculated or not*), and they are automatically refreshed when those fields are changed. The calculated fields are declared in the server side metadata in the DbSet's schema.

```

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

```

Client fields:

Client fields are just what they are - they are used only on the client. They don't exist in the database. 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.

```

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

```

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

Navigation fields:

The 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 entities. The parent entity can get (*using its navigation field*) an array of child entities and the 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 the navigation field 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, using navigation fields, you can assign the parent entity directly to the childToParent navigation field. On submit, the data service fixes this relationship automatically, and the submit is done in one transaction.

an example of assigning parent entity to the navigation field:

```
var cust = this.currentCustomer;
var ca = this.custAddressView.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.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
```

Entity and fields validations:

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

The client side validation is triggered when the new value is assigned to the field and when the entity's editing ends (*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 range and regex.

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 the 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:

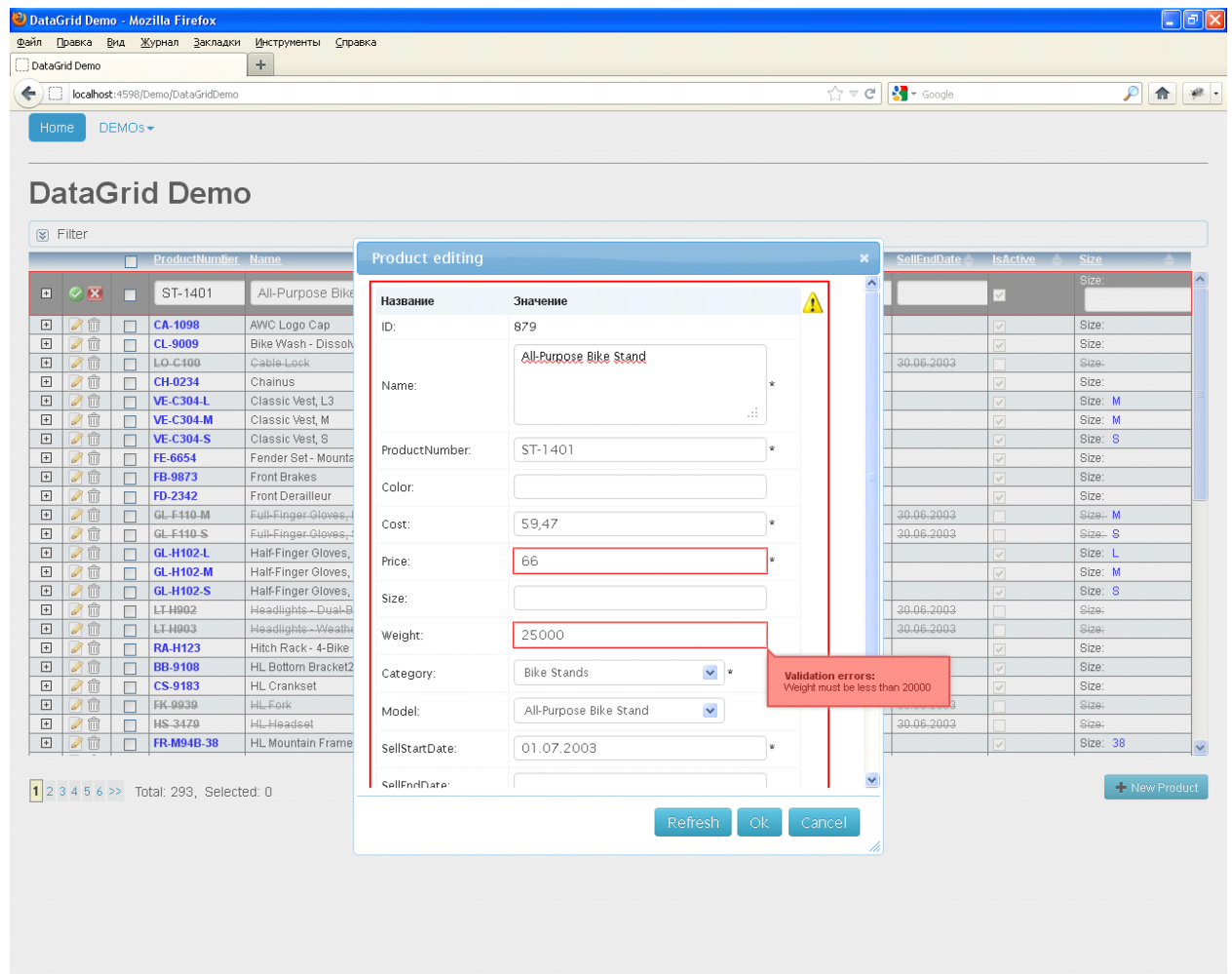
```
this._dbSet.addOnValidate(function (sender, args) {
    var item = args.item;
    if (!args.fieldName) { //full item validation
        if (!!item.SellEndDate) { //check it must be after Start Date
            if (item.SellEndDate < item.SellStartDate) {
                args.errors.push('End Date 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 an unsuccessful client side validation, the errors are added to the DbSet's internal collection of errors. The entity remains in the editing state and can not end editing while the errors exist.

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.

Data bindings automatically handle validation errors.

If the the entity has errors, then a UI control (*DataGrid*, *DataForm*) will display error notifications (*red borders and tooltips on mouse hovering*) near 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*).



The errors are cleared when the correct values are assigned to the fields or the changes are discarded by using a [cancelEdit](#) method.

As it was noted above, the validation is done on the client and the server side, and for the 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 error.

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.*

DataView:

A **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.

For example you can load all child entities for all the 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 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 combined*). 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 if you provide *fn_filter* then the data is filtered using this function, and then if you provide *fn_sort* then the data is sorted.

an example of the 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: DEMODB.Address, b: DEMODB.Address) { return a.AddressID - b.AddressID; },
    fn_filter: function (item: DEMODB.Address) {
        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;
        });
    }
});

```

One more example of the 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 we don't need to filter or sort the data.

When you want the data in the `DataView` to be refreshed (*refiltered and resorted*) you can call the `DataView`'s *refresh* method.

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

The DataView's specific methods:

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

The DataView's specific properties:

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

		value.
--	--	--------

the Association and the ChildDataView types:

The ChildDataView is descendant of the DataView class. It simplifies the task where you need to display details near a parent row in master- detail relationship. It uses an association to obtain child entities from the parent entity.

The association stores and updates a map of the parent-child relationship based on the foreign keys.

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 dbName="FileSystemObject" enablePaging="False" EntityType="{x:Type
models:FolderModel}" deleteDataMethod="Delete{0}">
      <data:DbSetInfo.fieldInfos>
        <data:FieldInfo fieldName="Key" dataType="String" maxLength="255" nullable="False"
isAutoGenerated="True" readOnly="True" primaryKey="1" />
        <data:FieldInfo fieldName="ParentKey" dataType="String" maxLength="255" nullable="True"
isReadOnly="True" />
        <data:FieldInfo fieldName="Name" dataType="String" maxLength="255" nullable="False"
isReadOnly="True" />
        <data:FieldInfo fieldName="Level" dataType="Integer" nullable="False" readOnly="True" />
        <data:FieldInfo fieldName="HasSubDirs" dataType="Bool" nullable="False" readOnly="True" />
        <data:FieldInfo fieldName="IsFolder" dataType="Bool" nullable="False" readOnly="True" />
        <data:FieldInfo fieldName="fullPath" dataType="String" isCalculated="True" />
      </data:DbSetInfo.fieldInfos>
    </data:DbSetInfo>
  </data:Metadata.DbSets>
  <data:Metadata.Associations>
    <!--the association definition -->
    <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 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._custAddrView = new MOD.db.ChildDataView<DEMODB.CustomerAddress>(
{
  association: custAssoc,
  fn_sort: function (a: DEMODB.CustomerAddress, b: DEMODB.CustomerAddress) {
    return a.AddressID - b.AddressID;
  }
});
```


When you need to display details for a parent entity, you will assign the ChildDataView's parentItem property. It triggers refreshing of the view with 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 (<i>details</i>) entities
getParentItem	Accepts an entity and returns the parent (<i>master</i>) 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 (*or above*) be installed on the server side 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
PermissionsInfo ServiceGetPermissions();
//information about service methods, DbSets and their fields information
MetadataInfo ServiceGetMetadata();
GetDataResult ServiceGetData(GetDataInfo getInfo);
ChangeSet ServiceApplyChangeSet(ChangeSet changeSet);
RefreshRowInfo ServiceRefreshRow(RefreshRowInfo getInfo);
InvokeResult ServiceInvokeMethod(InvokeInfo parameters);
}

```

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

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

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

an implementation of the DataServiceController:

```

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

    public DataServiceController()
    {
        this._serializer = new Serializer();
    }

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

    private IDomainService _DomainService;

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

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

    [HttpGet]
    public 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;
}
```

```
[HttpGet]
public 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;
}
```

```
[HttpGet]
public 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;
}
```

```
[HttpPost]
public ActionResult GetPermissions()
{
    var info = this.DomainService.ServiceGetPermissions();
    return Json(info);
}
```

```
public ActionResult GetMetadata()
{
    var info = this.DomainService.ServiceGetMetadata();
    return Json(info, JsonRequestBehavior.AllowGet);
}
```

```
[HttpPost]
public ActionResult GetItems(GetDataInfo getInfo)
{
    return new IncrementalResult(this.DomainService.ServiceGetData(getInfo));
}
```

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

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

```
[HttpPost]
public ActionResult InvokeMethod(InvokeInfo invokeInfo)
```

```

{
    var res = this.DomainService.ServiceInvokeMethod(invokeInfo);
    return Json(res);
}

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

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

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

protected override void Dispose(bool disposing)
{
    if (disposing && this._DomainService != null)
    {
        this._DomainService.Dispose();
        this._DomainService = 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 are needed to 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 void ExecuteChangeSet()
{
    using (TransactionScope transScope = new TransactionScope(TransactionScopeOption.RequiresNew,
        new TransactionOptions { IsolationLevel = IsolationLevel.ReadCommitted, Timeout =
        TimeSpan.FromMinutes(1.0) }))
    {
        this.DB.SaveChanges();

        transScope.Complete();
    }
}

```

The *GetMetadata* method should return *RIAPP.DataService.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 an 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 need to 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. For this purpose you can override *GetXAML* method of the *BaseDomainService* class. This method is designed to provide a XAML version of the metadata. The *DataService (RIAppDemoService)* in the demo application provides an example of this method implementation.

an implementation of the ServiceGetXAML method in the DEMO:

```
public override string GetXAML()
{
    var metadata = base.GetMetadata();
    var xaml = System.Windows.Markup.XamlWriter.Save(metadata);
    XNamespace data = "clr-namespace:RIAPP.DataService;assembly=RIAPP.DataService";
    XNamespace dal = "clr-namespace:RIAppDemo.DAL;assembly=RIAppDemo.DAL";

    XElement xtree = XElement.Parse(xaml);
    foreach (XElement el in xtree.DescendantsAndSelf())
    {
        el.Name = data.GetName(el.Name.LocalName);
        if (el.Name.LocalName == "Metadata")
        {
            List<XAttribute> atList = el.Attributes().ToList();
            el.Attributes().Remove();
        }
        else if (el.Name.LocalName == "DbSetInfo")
        {
            XAttribute entityTypeAttr = el.Attributes().Where(a => a.Name.LocalName == "EntityType").First();
            entityTypeAttr.Value = string.Format("{{x:Type {0}}}", entityTypeAttr.Value);
        }
    }
    xtree.Add(new XAttribute(XNamespace.Xmlns + "data", "clr-namespace:RIAPP.DataService;assembly=RIAPP.DataService"));
    return xtree.ToString();
}
```

After the *GetXAML* method is implemented you can get XAML representation of the metadata. You can navigate to GetXAML url in the browser like in the example <http://YOURSERVER/RIAppDemoService/GetXAML>. Then you can copy and paste the XAML into the resource of a WPF user control (*for example, in the DEMO, it is in the file RIAppDemo.BLL\DataServices\RIAppDemoMetadata.xaml*).

```

<data:Metadata xmlns:data="clr-namespace:RIAPP.DataService;assembly=RIAPP.DataService">
  <data:Metadata.DbSets>
    <data:DbSetInfo dbSetName="Address" insertDataMethod="Insert{0}" updateDataMethod="Update{0}"
deleteDataMethod="Delete{0}" enablePaging="True" pageSize="25" EntityType="{x:Type riaddal:Address}">
      <data:DbSetInfo.fieldInfos>
        <data:FieldInfo fieldName="AddressID" dataType="Integer" isAutoGenerated="True" isPrimaryKey="1" />
        <data:FieldInfo fieldName="AddressLine1" dataType="String" isNullable="False" />
        <data:FieldInfo fieldName="AddressLine2" dataType="String" />
        <data:FieldInfo fieldName="City" dataType="String" isNullable="False" />
        <data:FieldInfo fieldName="StateProvince" dataType="String" isNullable="False" />
        <data:FieldInfo fieldName="CountryRegion" dataType="String" isNullable="False" />
        <data:FieldInfo fieldName="PostalCode" dataType="String" isNullable="False" />
        <data:FieldInfo fieldName="rowguid" dataType="Guid" />
        <data:FieldInfo fieldName="ModifiedDate" dataType="DateTime" />
      </data:DbSetInfo.fieldInfos>
    </data:DbSetInfo>
    <data:DbSetInfo dbSetName="Customer" insertDataMethod="Insert{0}" updateDataMethod="Update{0}"
deleteDataMethod="Delete{0}" enablePaging="True" pageSize="25" EntityType="{x:Type riaddal:Customer}">
      <data:DbSetInfo.fieldInfos>
        <data:FieldInfo fieldName="CustomerID" dataType="Integer" isAutoGenerated="True" isPrimaryKey="1" />
        <data:FieldInfo fieldName="NameStyle" dataType="Bool" />
        <data:FieldInfo fieldName="Title" dataType="String" />
        <data:FieldInfo fieldName="FirstName" dataType="String" isNullable="False" />
        <data:FieldInfo fieldName="MiddleName" dataType="String" />
        <data:FieldInfo fieldName="LastName" dataType="String" isNullable="False" />
        <data:FieldInfo fieldName="Suffix" dataType="String" />
        <data:FieldInfo fieldName="CompanyName" dataType="String" />
        <data:FieldInfo fieldName="SalesPerson" dataType="String" />
        <data:FieldInfo fieldName="EmailAddress" dataType="String" />
        <data:FieldInfo fieldName="Phone" dataType="String" />
        <data:FieldInfo fieldName="PasswordHash" dataType="String" isNullable="False" />
        <data:FieldInfo fieldName="PasswordSalt" dataType="String" isNullable="False" />
        <data:FieldInfo fieldName="rowguid" dataType="Guid" />
        <data:FieldInfo fieldName="ModifiedDate" dataType="DateTime" />
      </data:DbSetInfo.fieldInfos>
    </data:DbSetInfo>
    <data:DbSetInfo dbSetName="CustomerAddress" insertDataMethod="Insert{0}" updateDataMethod="Update{0}"
deleteDataMethod="Delete{0}" enablePaging="True" pageSize="25" EntityType="{x:Type riaddal:CustomerAddress}">
      <data:DbSetInfo.fieldInfos>
        <data:FieldInfo fieldName="CustomerID" dataType="Integer" isPrimaryKey="1" />
        <data:FieldInfo fieldName="AddressID" dataType="Integer" isPrimaryKey="2" />
        <data:FieldInfo fieldName="AddressType" dataType="String" isNullable="False" />
        <data:FieldInfo fieldName="rowguid" dataType="Guid" />
        <data:FieldInfo fieldName="ModifiedDate" dataType="DateTime" />
      </data:DbSetInfo.fieldInfos>
    </data:DbSetInfo>
    <data:DbSetInfo dbSetName="Product" insertDataMethod="Insert{0}" updateDataMethod="Update{0}"
deleteDataMethod="Delete{0}" enablePaging="True" pageSize="25" EntityType="{x:Type riaddal:Product}">
      <data:DbSetInfo.fieldInfos>
        <data:FieldInfo fieldName="ProductID" dataType="Integer" isAutoGenerated="True" isPrimaryKey="1" />
        <data:FieldInfo fieldName="Name" dataType="String" isNullable="False" />
        <data:FieldInfo fieldName="ProductNumber" dataType="String" isNullable="False" />
        <data:FieldInfo fieldName="Color" dataType="String" />
        <data:FieldInfo fieldName="StandardCost" dataType="Decimal" />
        <data:FieldInfo fieldName="ListPrice" dataType="Decimal" />
        <data:FieldInfo fieldName="Size" dataType="String" />
      </data:DbSetInfo.fieldInfos>
    </data:DbSetInfo>
  </data:Metadata.DbSets>
</data:Metadata>

```

If you obtain a XAML version of the metadata and use it inside a WPF user control, you then can simplify the GetMetadata method of the DataService by returning the metadata contained in the WPF control's resources.

```

protected override Metadata GetMetadata()
{
    //returns raw (unedited) metadata from the base class implementation
    //return base.GetMetadata();

    //returns corrected metadata from WPF control
    return (Metadata)(new RIAppDemoMetadata().Resources["MainDemo"]);
}

```

Note: *The good part of storing metadata in the 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.*

DataService classes usually have 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**'s 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.

The DbSet can have several query methods with different names (*no overloading*). For example the Product DbSet in the DEMO application have another 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 the FetchSize in the metadata - the batch size of the rows fetching. The DataService will send the data 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 the query methods, there are also CRUD methods (*they are optional*) to perform inserts, deletes and updates of 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 got by replacing {0} with dbSetName value.

But there's more, in addition to the query and CRUD methods there are 3 more special methods types which can be used in the data service: The [refresh](#) methods, the [custom validation](#) methods and the [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. The refresh can be also made by using a query method, but the refresh methods are more convenient to use from the client (*just use entity's [refresh](#) method*). The refresh method (*for a DbSet*) can be set in the metadata as [refreshDataMethod](#) property value.

an example of the refresh method implementation:

```
public Product RefreshProduct(RefreshRowInfo 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 (*for a DbSet*) can be set in the metadata as [validateDataMethod](#) property value.

Service methods

They are used to be 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 the [Invoke](#) attribute. They can also have the [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
}
```

Data Service's Metadata

The metadata is usually defined in the form of XAML in a WPF user control's resources. It is better to use a separate assembly (*class library*) in the solution for this.

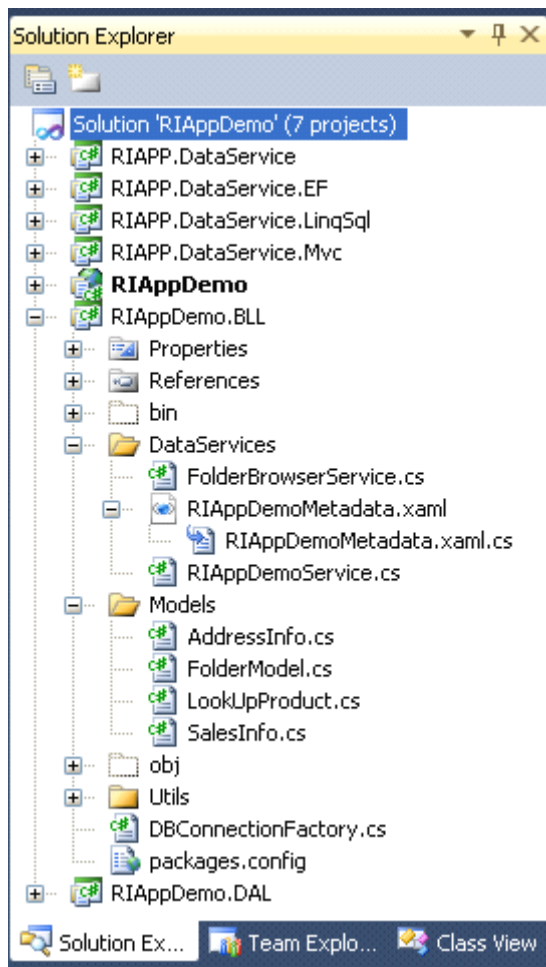
In the demo application they were put in *RIAppDemo.BLL* class library project.

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

RIAppDemoService.cs - file contains the DataService for demo project.

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

A view of the demo application from the VS2012 solution explorer (*with RIAppDemoMetadata.xaml* file visible):



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.

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);
    }
}

```

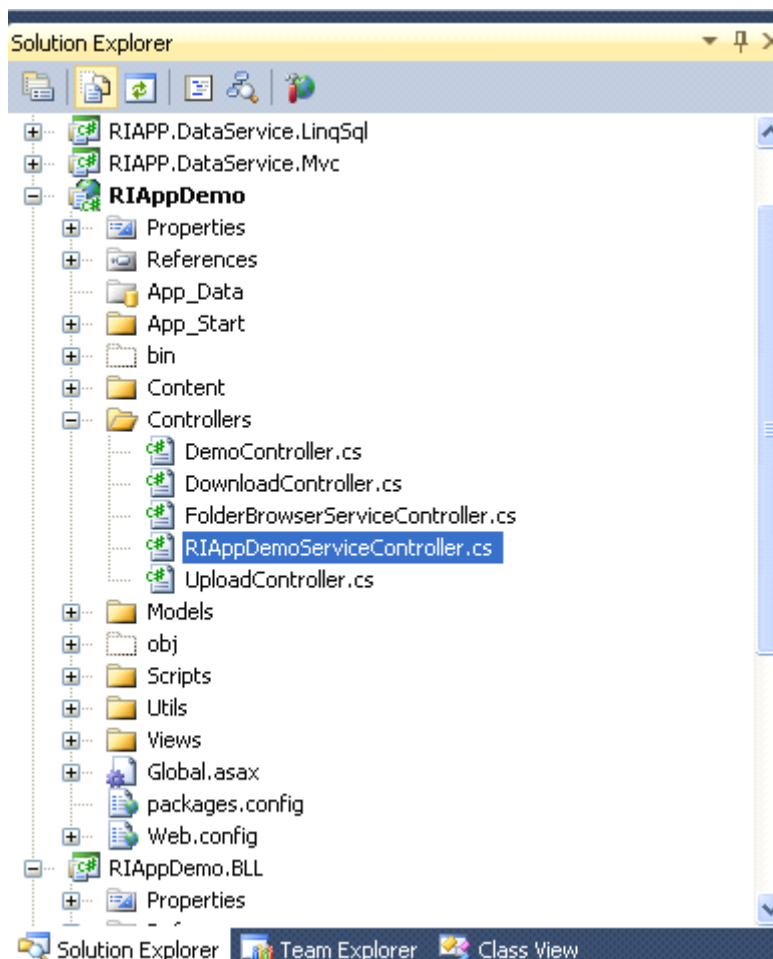
Note: You can add to the controller any methods you need. For example the above controller includes two custom methods which are used on the page. They are used to return the lookup data and it is embedded into the page in the javascript section:

```

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

```

The RIAppDemoService controller in the RIAppDemo MVC project:



The metadata definition for every data service has a key (*unique name*) by which it is taken from the control in the DataService's *GetMetadata* method.

```

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

```

Note: *The above method is executed in the STA thread by the DataService, and the metadata is cached inside the data service so this method is executed only on the first request.*

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

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

Their meaning are clear from their names.
But several attributes need more explanation:

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*), and should be generated on the server when a new entity is added. The other way is generate values for the primary key 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 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* (*Note: 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*).

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. 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 concurrency 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 them*).

isRowTimeStamp - This attribute is used to mark the field that it always needs original values for updates. The field's original value always returns to the client on submit operation. The field is usually *readOnly* on the client.

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})\$"

The Associations

An association defines foreign key references and navigation fields names.

There are two ways of loading related entities to the client.

The first one is to include related entities to the result of a query method.

An example of inclusion of related entities in the result:

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

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

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

But this option is not always the best. This usually complicates queries and slows the query execution in real world applications. In many cases it is better to load child and parent entities from the client using separate queries.

For example, you can load first the Customer entities, then you can execute another query to load the CustomerAddress entities for the loaded customers.

[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);
}

```

and then you can load the address entities by their keys.

```

[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);
}

```

Note: *This allows to load many pages (**pages of the data in the datagrid, not HTML pages**) of master rows at once, and then to load the details (only for the current page in the datagrid). When the user goes to another page the application retrieves the details for the new page and discard for the old one.*

This will improve user experience when the master rows 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.

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 the data service class or a method with the [Authorize](#) attribute. The Authorize attribute can include roles. Without including the roles it is simply checked that the user is authenticated. the Authorize attribute is optional, and when it is not applied then it is assumed that the access is allowed on that level.

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 next level (*method's level*) which encompasses query methods, CRUD methods, refresh and service (*invoke*) methods attributes checks, the authorization checks 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[] { ADMIN_ROLE })]
    public void UpdateCustomer(Customer customer)
    {
        Customer orig = this.GetOriginal<Customer>();
        this.DB.Customers.Attach(customer, orig);
    }

    [Authorize(Roles = new string[] { ADMIN_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 BaseDomainService class has a virtual method which creates and returns an authorizer instance. This method can be overridden in the descendants.

```

protected virtual IAuthorizer CreateAuthorizer()
{
    return new AuthorizerClass(this.GetType(), this.CurrentPrincipal);
}

```

Change Tracking

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:

```

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

```

Error logging 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
}

```

Disposing 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 a raw implementation of the data service's methods

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

```

protected override string GetCSharp()
{

```

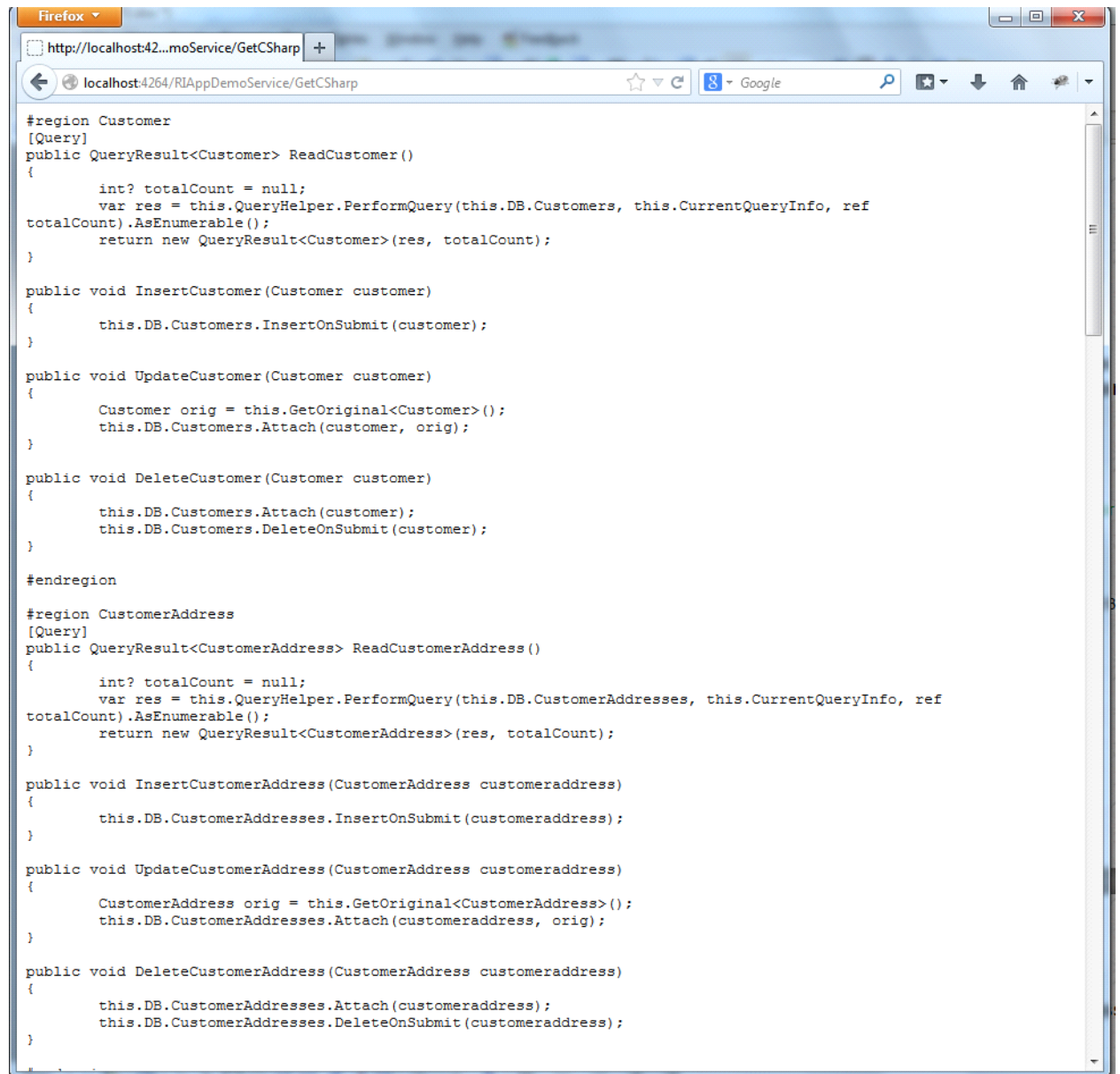
```

    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/GetCSharp> 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.



```

#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);
}

```

Warning: For security reasons you need to set DataService's `IsGetXXXMethodsEnabled` property value to true, in order GetTypeScript, GetCSharp and GetXAML methods can be executed from an internet browser. Without it you will get an error! On deployment to the production site you should disable this feature by setting this property to false.

[an example setting IsGetXXXMethodsEnabled 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.IsGetXXXMethodsEnabled = 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. Those classes include inside them the metadata information. You can obtain those implementation by navigating in the internet browser to the <http://YOURSERVER/RIAppDemoService/GetTypeScript> url.

```

/*
    Generated from: /RIAppDemoService/GetTypeScript on 2013-12-05 13:05 at 13:05
    Don't make manual changes here, because they will be lost when this db interface will be regenerated!
*/

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;
}

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

```

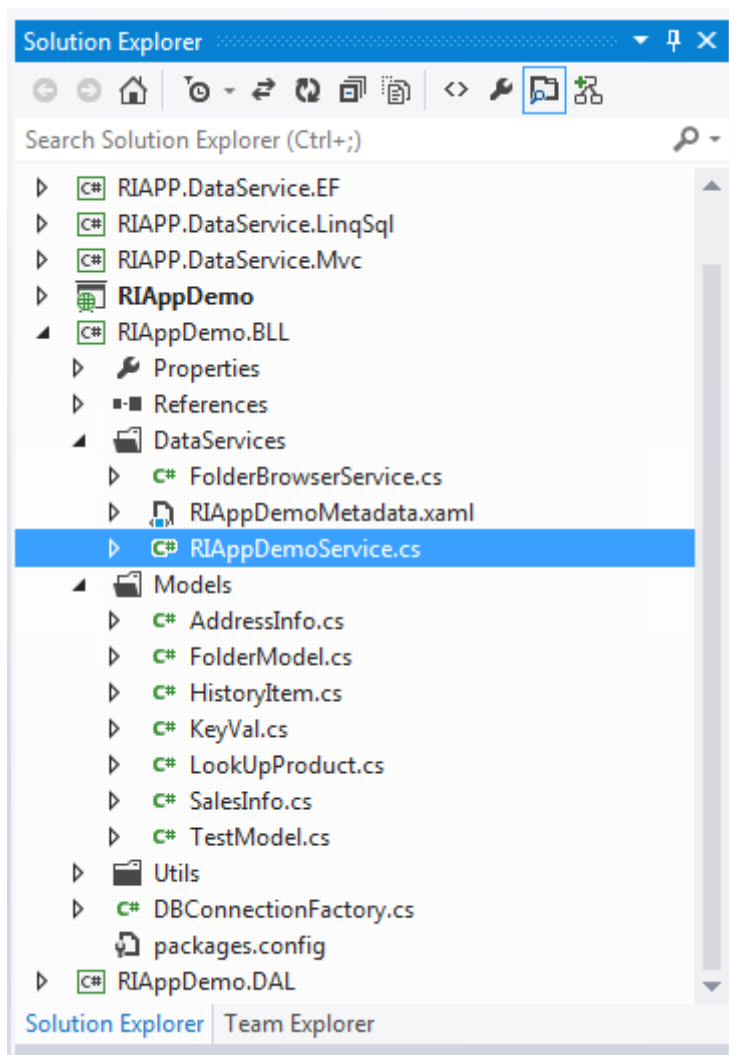
You can include any class from your server side code to be included into this generated typescript code.

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 return 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 a Models folder. It contains classes that are needed on the client side in the generated typescript code.



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.MOD.utils.IEditable"})]
public class HistoryItem
{
    public string radioValue
    {
        get;
        set;
    }

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

will have the output as:

```
/*
    Generated from C# HistoryItem model
*/
export interface IHistoryItem extends RIAPP.MOD.utils.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(HistoryListItem, [{ 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 say to 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 the typescript's enums from the c# enums.