jRIAppTS, the RIA framework – user's guide

- Authorization
- Change tracking (*auditing*)
- Error logging
- Disposal of resources (*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 (*RIAppDemo*) was implemented in ASP.NET MVC.
The Server part resembles Microsoft WCF RIA services, featuring data services which is consumed by clients.
The Client part resembles the Microsoft Silverlight client development, only it is based on HTML (*not XAML*), and uses the typescript language for coding.
The framework was designed primarily for creating data centric Line of Business (*LOB*) applications which will work natively in browsers without the need for plugins .
The framework supports a wide range of essential features for creating LOB applications, such as, declarative use of *databindings*, integration with the server side dataservice, data templates, client side and server side data validation, localization, authorization, and a set of UI controls, like the *datagrid*, the *stackpanel* , the *dataform* and a lot of utility code.
Unlike many other existing javascript frameworks, which use the 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 a strongly typed environment either on the client or on the server.

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

The framework has a set of controls such as:
A *DataGrid* – the control for displaying and editing the data in the table form. It supports the databinding, the row selection with keyboard keys, sorting by the 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 a vertical list . It uses the data template for the display of the items and also has the support for item selection with the help of keyboard keys and the mouse.
A *ListBox* - the control which encapsulates the HTML select tag and attaches to it the logic to draw the data from the collection type datasource.

A *DataForm* - the control which bounds a datacontext to a region and allows to use special contents elements inside this region. It also provides for summary error display.
A *DbContext* – the data control used as the data manager to store the data (*DbSets*) and to cache changes on the client for submitting them later to the dataservice.

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

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

1.2 *Licensing*

1.3 *The Framework's files and deployment*

The framework is written in typescript language. So, the typescript code is compiled first into the javascript code before the use in HTML5 applications.
The typescript code is contained in the Microsoft Visual Studio Solution *jriappTS* - which includes several projects. On compilation (*each project is compiled separately*), in the built directory will appear several javascript files:

jriapp_shared.js - the bundle with common classes (collection types, utilities, and the other shared types)
jriapp.js - application class and data binding infrastructure (it depends on the **jriapp_shared.js**)
jriapp_ui.js - element views for the User Interface (it depends on the **jriapp.js** and the **jriapp_shared.js**)
jriapp_db.js - client side entity framework (it depends only on the **jriapp_shared.js**)
jriapp_langs.js - local strings, needed for other than english language.

If someone does not want to use the full framework, but needs only the means to work with databases using other frameworks, he (she) can use only the **jriapp_shared.js** and the **jriapp_db.js** bundles (*they are not dependent on JQuey, QTip and the RequireJS*).

The framework depends on JQuery (only in the user interface bundle), the Moment, QTip (only in the user interface bundle), the RequireJS. The Moment, the QTip and the RequireJS are easily replaceable.

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 there are also the css and the img folders, which contain the styles and the images respectively.

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

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

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

```
<link href="@Url.Asset("/Scripts/bootstrap/css/bootstrap.min.css", false)" rel="stylesheet" type="text/css"/>
 <link href="@Url.Asset("/Scripts/jquery/ui/jquery-ui.min.css", false)" rel="stylesheet" type="text/css" />
 <link href="@Url.Asset("/Scripts/jquery/ui/jquery-ui.theme.min.css", false)" rel="stylesheet" type="text/css"/>
 <link href="@Url.Asset("/Content/Site.css", true)" rel="stylesheet" type="text/css"/>
  <link href="@Url.Asset("/Scripts/qtip/jquery.qtip.min.css", false)" rel="stylesheet" type="text/css"/>

    @RenderSection("CssImport", false)

    <script src="@Url.Asset("/Scripts/jquery/jquery-3.1.1.min.js", false)" type="text/javascript"></script>
    <script src="@Url.Asset("/Scripts/bootstrap/js/bootstrap.min.js", false)" type="text/javascript"></script>
    <script src="@Url.Asset("/Scripts/jquery/ui/jquery-ui.min.js", false)" type="text/javascript"></script>
    <script src="@Url.Asset("/Scripts/moment/moment.min.js", false)" type="text/javascript"></script>
    <script src="@Url.Asset("/Scripts/qtip/jquery.qtip.min.js", false)" type="text/javascript"></script>

    @RenderSection("JSImport", false)

    <script src="@Url.Asset("/Scripts/require.min.js", false)" type="text/javascript"></script>
    @Scripts.Render("~/bundles/jriapp")
    @Scripts.Render("~/bundles/shared")

    <script type="text/javascript">
        var jriapp_config = {
            frameworkPath: '@Url.Asset("/Scripts/jriapp/", true)',
            debugLevel: 0
        };

        var config = {
            baseUrl: "@Url.Asset("/Scripts/demo/", true)",
```

```
        waitSeconds: 10
    };
    requirejs.config(config);
</script>
```

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

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

For example, the *ManyToManyDemo.cshtml* page contains script tags:

```
<script src="@Url.Content("~/Scripts/demo/manToManDemo/main.js", true)" type="text/javascript"></script>

<script type="text/javascript">
    require(["manToManDemo/main", "jriapp_ui", "jriapp_ru"],
        function(DEMO, UI, LANG) {
            $("#loading").fadeIn();
            var mainOptions = {
                service_url: null,
                permissionInfo: null
            };

            (function(ops) {
                ops.service_url = '@Url.RouteUrl("Default", new {controller = "RIAppDemoServiceEF", action = ""})';
                ops.permissionInfo = @Html.Action("PermissionsInfo", "RIAppDemoServiceEF");
            })(mainOptions);

            DEMO.start(mainOptions).then(function () {
                $("#demoContent").animate({ opacity: 1 }, 1000);
                $("#loading").fadeOut(1000);
            }, function (err) {
                $("#loading").fadeOut(1000);
            });
        });
</script>
```

The client side code the users write for creating HTML5 applications consists of *view models*, which are javascript objects which expose properties which can be data bound on a HTML page (*using data-bind attribute*). Also, the *view models* can expose commands so that the HTML controls like buttons or anchors could invoke actions (*wrapped in the commands*).
The other important pieces of the web application are the *element views*, which are roughly equivalent to the *AngularJS* components. They allow to attach a javascript code (*usually, UI controls*) to the HTML elements in a declarative way.

This only scraches the surface of what you can do with the help of the framework. There will be a more detailed explanation further in this guide.

Note: *the good thing to learn how the framework works is to see how the demo web application works and to look at the code.*

The framework's classes

2.1 *BaseObject class*

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

The BaseObject implements the IBaseObject interface:

```
export interface IBaseObject extends IErrorHandler, IDisposable {
    getIsStateDirty(): boolean;
    isHasProp(prop: string): boolean;
    readonly objEvents: IObjectEvents;
}

export interface IDisposable {
    dispose(): void;
    getIsDisposed(): boolean;
}

export interface IErrorHandler {
    handleError(error: any, source: any): boolean;
}
```

The *BaseObject* class has the objEvents property, which exposes an object which implements the *IEvents* interface for subscribing to the object events:

```
export interface IEvents<T = any> {
    canRaise(name: string): boolean;
    on(name: string, handler: TEventHandler<T, any>, nmspace?: string, context?: object, priority?: TPriority): void;
    off(name?: string, nmspace?: string): void;
    // remove event handlers by their namespace
    offNS(nmspace?: string): void;
    raise(name: string, args: any): void;
    raiseProp(name: keyof T | "*" | "[*]"): void;
    // to subscribe for changes on all properties, pass in the prop parameter: '*'
    onProp(prop: keyof T | "*" | "[*]", handler: TPropChangedHandler<T>, nmspace?: string, context?: object, priority?:
TPriority): void;
    offProp(prop?: keyof T | "*" | "[*]", nmspace?: string): void;
}

export interface IObjectEvents<T = any> extends IEvents<T> {
    addOnError(handler: TErrorHandler<T>, nmspace?: string, context?: object, priority?: TPriority): void;
    offOnError(nmspace?: string): void;
    addOnDisposed(handler: TEventHandler<T, any>, nmspace?: string, context?: object, priority?: TPriority): void;
    offOnDisposed(nmspace?: string): void;
    readonly owner: T;
}
```

Each object has *getIsStateDirty* method which indicates if the object is an invalid state. You can check the method return value, to be sure that the object is still in the valid state after some asynchronous operation is completed (*since in the meantime the object can be disposed*), as in the next example:

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

The *BaseObject* class has a *getIsDisposed* method, which returns true after the object is completely disposed. This method return value is usually checked in the overridden dispose method, so if the object have been disposed then to exit the method immediately, like in the next example:

```
dispose(): void {
        if (this.getIsDisposed()) // checks to see, if already disposed
            return;
        this.setDisposing(); //  informs that the destruction is started
        this._unbindDS();
        this._clearContent();
        this._$el.removeClass(_css.pager);
        this._el = null;
        this._$el = null;
        super.dispose();
}
```

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

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

```
export class TestObject extends RIAPP.BaseObject {
    private _testProperty1: string;
    private _testProperty2: string;
    private _testProperty3: string;

    constructor(initPropValue: string) {
        super();
        this._testProperty1 = initPropValue;
        this._testProperty2 = null;
        this._testProperty3 = null;
    }
    get testProperty1(): string { return this._testProperty1; }
    set testProperty1(v: string) {
        if (this._testProperty1 != v) {
            this._testProperty1 = v;
            this.objEvents.raiseProp('testProperty1');
            this.objEvents.raiseProp('isEnabled');

            //let the command to evaluate its availability
            this._testCommand.raiseCanExecuteChanged();
        }
    }
    get testProperty2(): string { return this._testProperty2; }
    set testProperty2(v: string) {
        if (this._testProperty2 != v) {
            this._testProperty2 = v;
            this.objEvents.raiseProp('testProperty2');
        }
    }
    get testProperty3(): string { return this._testProperty3; }
    set testProperty3(v: string) {
        if (this._testProperty3 != v) {
            this._testProperty3 = v;
            this.objEvents.raiseProp('testProperty3');
        }
    }
}
```

The object instance can trigger the two predefined events: '*error*', '*disposed*'.

The *error* event is triggered from the BaseObject's *handleError* method.

The error event handler can set *isHandled* to true, and after that the error handling is successfully finished without going up the stack.

The *disposed* event is triggered from the BaseObject *dispose* method. It is used to notify about the object destruction and can be used by subscribers to remove references to the disposed object.

The Users can subscribe to events by using the *on* method, as in the example:

```
addOnNodeSelected(fn: (sender: FolderBrowser, args: { item: FileSystemObject; }) => void, nmspace?: string) {
    this.objEvents.on('node_selected', fn, nmspace);
}
```

Note: *Many framework classes have strongly typed versions of methods to add and remove event handlers to simplify event subscriptions.*

```
 // subscribe for the application's errors
this.objEvents.addOnError(function (_s, data) {
        debugger;
        data.isHandled = true;
        self.errorVM.error = data.error;
        self.errorVM.showDialog();
    });
```

The last argument of the *on* method is the event *namespace*, which is an optional argument and can be used to selectively remove subscriptions. For this purpose you can use the *offNS* method. To remove the subscriptions to the events you can also use the *off* method, and provide the event name to it (*and optionally the event namespace*).

```
// remove the subscription by the event name, plus the event namespace
someObject.objEvents.off('status_changed', this.uniqueID);
```

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

```
this._objEvents.raise(OBJ_EVENTS.disposed, {});
```

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

```
this.objEvents.onProp('testProperty1', ()=> { alert('testProperty1 changed'); });
```

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

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

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

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

2.2 the *Bootstrap class*

All the code in the client side part of the framework is structured into modules. The *Bootstrap* object instance is created by the framework on the time of loading of the *jriapp.js* bundle. It is a singleton object. The Bootstrap class is defined in the *bootstrap.ts* module. The instance of this object can be accessed in the code via the exported *bootstrap* variable.

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

The bootstrap object also holds the references for registered objects (*like converters or services*).

The bootstrap object has the *defaults* property, which exposes an instance of the Defaults object class. 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, path to the images, as in the example:

```
bootstrap.init((bootstrap) => {
    // replace default buttons styles with something custom
    const ButtonsCSS = bootstrap.defaults.ButtonsCSS;
    ButtonsCSS.Edit = 'icon icon-pencil';
    ButtonsCSS.Delete = 'icon icon-trash';
    ButtonsCSS.OK = 'icon icon-ok';
    ButtonsCSS.Cancel = 'icon icon-remove';
});
```

The date formats use the *moment.js* format style.

2.3 the *Application class*

The *Application* - class, is used as the aggregation root for exposing view models.
The Bootstrap class instance is created automatically, but the Application instance is created in the user's code. Usually the application class is inherited to add some properties, which are initialized in the overridden *onStartUp* method.
On creation of the application instance you can provide the options object to the constructor. The options interface is defined as:

```
export interface IAppOptions {
    modulesInits?: IIndexer<(app: IApplication) => void>;
    appRoot?: Document | HTMLElement;
}
```

The earlier version of the framework allowed the creation of several Application instances. The current version allows to create only one Application instance.

The *moduleInits* are used to add a function (*functions*) for innitializing code - for example, to register the element views, the converters, the services.
The *appRoot* is typicaly is left empty (*it is optional*). It is the scope (the *region*) of the application on the HTML page. By default, a whole HTML document is the scope and the application root refers to the window.document property.

The Application instance is usually created in the Bootstrap *startUp* method. This method accepts a callback function which should return the Application instance. It accepts a second optional callback which can be used to register objects. The Bootstrap *startUp* method returns a promise which is resolved when the Application have been started.

```
return bootstrap.startApp(() => {
      return new DemoApplication(options);
   }, (app) => {
      app.registerConverter('sizeConverter', new SizeConverter());

      // an example of how to load a file with templates from the server (for loading group of templates- see spaDEMO.ts)
      app.loadTemplates(options.templates_url);

      // this registered function will be invoked every  time when the template with that name is needed
      // P.S. - but a better way how to load templates is to register templates' groups
      // see the Single Page RIAPP.Application Demo (spaDEMO.ts) how it is done there
      app.registerTemplateLoader('productEditTemplate', coreUtils.memoize(() => {
         return utils.http.getAjax(options.productEditTemplate_url); })
      );
   }).then((app) => {
      if (!!options.modelData && !!options.categoryData) {
         // the data was embedded into HTML page as json, just use it
         app.productVM.filter.modelData = options.modelData;
         app.productVM.filter.categoryData = options.categoryData;
         return null;
      }
      else {
         return app.productVM.filter.load().then(() => {
            return app.productVM.load().then(function (loadRes) {/*alert(loadRes.outOfBandData.test);*/ });
         });
      }
});
```

The Application options are usually extended to add properties which are needed for the application initialization.
For example, the datagrid demo example extends the application options by adding more properties to it.

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

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

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

    constructor(options: IMainOptions) {
        super(options);
        this._dbContext = null;
        this._errorVM = null;
        this._headerVM = null;
        this._productVM = null;
        this._uploadVM = null;
        this._sseVM = null;
        this._websockVM = null;
    }
    onStartUp() {
        const self = this, options: IMainOptions = self.options;
        this._dbContext = new DEMODB.DbContext();
        this._dbContext.addOnDbSetCreating((s, a) => {
            console.log("DbSet: %s is creating", a.name);
        });
        this._dbContext.initialize({ serviceUrl: options.service_url, permissions: options.permissionInfo });
        this._dbContext.dbSets.Product.defineIsActiveField(function (item) {
            return !item.SellEndDate;
        });
        this._errorVM = new COMMON.ErrorViewModel(this);
        this._headerVM = new HEADER.HeaderVM(this);
        this._productVM = new ProductViewModel(this);
        this._uploadVM = new UploadThumbnailVM(this, options.upload_thumb_url);
        function handleError(sender: any, data: any) {
            self._handleError(sender, data);
        };
        //here we could process application's errors
        this.objEvents.addOnError(handleError);
        this._dbContext.objEvents.addOnError(handleError);
        //instead of server side events i added websocket

        if (!!options.sse_url && SSEVENTS.SSEventsVM.isSupported()) {
            this._sseVM = new SSEVENTS.SSEventsVM(options.sse_url, options.sse_clientID);
            this._sseVM.addOnMessage((s, a) => { self._sseMessage = a.data.message;
self.objEvents.raiseProp('sseMessage'); });
            this._sseVM.objEvents.addOnError(handleError);
        }


        if (WEBSOCK.WebSocketsVM.isSupported()) {
            this._websockVM = new WEBSOCK.WebSocketsVM(WEBSOCK.WebSocketsVM.createUrl(81, 'PollingService',
false));
            this._websockVM.addOnMessage(this._onWebsockMsg, this.uniqueID, this);
            this._websockVM.objEvents.addOnError(handleError);
        }

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

        super.onStartUp();
```

```
    }
    private _handleError(sender: any, data: any) {
        debugger;
        data.isHandled = true;
        this.errorVM.error = data.error;
        this.errorVM.showDialog();
    }
    private _onWebsockMsg(sender: WEBSOCK.WebSocketsVM, args: { message: string; data: any; }) {
        this._sseMessage = args.data.message; this.objEvents.raiseProp('sseMessage');
    }
    // the dispose method is redundant here since the application lives while the page lives
    dispose() {
        if (this.getIsDisposed())
            return;
        this.setDisposing();
        const self = this;
        try {
            self._errorVM.dispose();
            self._headerVM.dispose();
            self._productVM.dispose();
            self._uploadVM.dispose();
            self._dbContext.dispose();
            if (!!self._sseVM)
                self._sseVM.dispose();
        } finally {
            super.dispose();
        }
    }
    get options() { return <IMainOptions>this._options; }
    get dbContext() { return this._dbContext; }
    get errorVM() { return this._errorVM; }
    get headerVM() { return this._headerVM; }
    get productVM() { return this._productVM; }
    get uploadVM() { return this._uploadVM; }
    // server side events and websocket
    get sseMessage() { return this._sseMessage; }
    get sseVM() { return this._sseVM; }
    get websockVM() { return this._websockVM; }
}
```

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

The Application class implements the IApplication interface:

```
export interface IApplication extends IErrorHandler, IDataProvider, IBaseObject {
    _getInternal(): IInternalAppMethods;
    addOnStartUp(fn: TEventHandler<IApplication, any>, nmspace?: string, context?:
IBaseObject): void;
    offOnStartUp(nmspace?: string): void;
    registerElView(name: string, vwType: IViewType): void;
    registerConverter(name: string, obj: IConverter): void;
    getConverter(name: string): IConverter;
    registerSvc(name: string, obj: any): void;
    getSvc<T>(name: string): T;
    getSvc(name: string): any;
    registerObject(name: string, obj: any): void;
    getObject<T>(name: string): T;
```

```
    getObject(name: string): any;
    loadTemplates(url: string): IPromise<any>;
    registerTemplateLoader(name: string, loader: () => IPromise<string>): void;
    getTemplateLoader(name: string): () => IPromise<string>;
    registerTemplateGroup(name: string, url: string): void;
    getOptions(name: string): string;
    bind(opts: TBindingOptions): IBinding;
    startUp(onStartUp?: (app: IApplication) => any): IPromise<IApplication>;
    readonly uniqueID: string;
    readonly appName: string;
    readonly appRoot: Document | HTMLElement;
    readonly viewFactory: IElViewFactory;
}
```

2.4 the *Binding class*

The framework *Binding* class is usually not used in the user's code, since the databinding is most often done declaratively.

If it is needed to databind in the code then the application class has the utility method - the *bind* which creates and returns an instance of the *Binding*.

An example of databinding object properties in code (typescript code):

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

The BINDING_MODE determines the direction of the databinding. The default is the OneWay - from the source to the target.

```
export const enum BINDING_MODE {
   OneTime = 0,
   OneWay = 1,
   TwoWay = 2,
   BackWay = 3
}
```

When a declarative databinding expression is parsed, the HTML DOM element is wrapped with a class derived from the *BaseElView* class. The Element View ia a class which exposes properties which can be databound. The element view is the target of the databinding.

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

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

```
<a  data-bind="this.command,to=expanderCommand,source=headerVM"  data-view="expander"></a>
```

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

```
<select id="prodSCat" size="1" class="span2"
                data-bind-1="this.dataSource,to=filter.ChildCategories"
                data-bind-2="this.selectedValue,to=filter.childCategoryID,mode=TwoWay"
                data-bind-3="this.selectedItem,to=filter.selectedCategory,mode=TwoWay"
                data-bind-4="this.toolTip,to=filter.selectedCategory.Name"
                data-bind-5="this.textProvider,to=optionTextProvider"
                data-bind-6="this.stateProvider,to=optionStateProvider"
                data-view-options="valuePath=ProductCategoryID,textPath=Name">
</select>
```

The options can be defined in a script tag (instead of inline definition):

```
<script id="salesDateOptions" type="text/x-options">
   datepicker: { showOn:button,
   yearRange:'-15:c',
   changeMonth: true,
   changeYear:true,
   yearRange:'1998:2005',
   minDate:date(1998-06-01,YYYY-MM-DD),
   maxDate:date(20050707)
   }
</script>

<input type="text" id="saleStart1" placeholder="Enter Date"
                data-bind="this.value,to=filter.saleStart1,mode=TwoWay,converter=dateConverter"
                data-view="datepicker"
                data-view-options="get(salesDateOptions)"/>
```

If you don't provide a data-view attribute then the default element view for this html element will be created.

The data binding expressions are contained inside a custom *data-bind* attribute value (*it is needed only to start with the data-bind word, something like the data-bind-1 will be ok*).
The data-bind attribute value can contain multiple data binding expressions. Each expression is enclosed in the curly braces {} *(you can omit them if you have only one expression*).
If multiple databindings are needed then its better to use separate *data-bind* expression for each of them.

The databind expression: this.dataSource,to=mailDocsVM.dbSet,mode=OneWay,source=sendListVM

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

If the databinding expressions are outside of the data templates or the data forms and **without explicitly providing the source**, then the source will be the **application instance** (*by default*), but if they **are** inside the data templates or data forms, then the implicit source is the current **data context** (*data templates as well as data forms have the data contexts*).

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

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

this.dataSource,to=sendListVM.mailDocsVM.dbSet

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

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

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

targetPath=dataSource,sourcePath=sendListVM.mailDocsVM.dbSet

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

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

targetPath:dataSource,sourcePath:VM.sendListVM.mailDocsVM.dbSet

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

The databinding expressions can be defined externally and can be used the same as the externally defined options. (also)

```
<script id="prodMCatDataBind" type="text/x-options">
    {this.selectedValue,to=filter.parentCategoryID,mode=TwoWay}
    {this.textProvider,to=optionTextProvider}
    {this.stateProvider,to=optionStateProvider}
</script>

 <select id="prodMCat" size="1" class="span3" data-bind="get(prodMCatDataBind)"
  @*
    Providing the datasource through the options - (using the bind built-in function)
    bind(ParentCategories,productVM.filter) takes two arguments - the first is the path,
the second is the source
    the second parameter is optional. Their semantics is the same as in the usual
databinding. You could write it without the source argument, like that -
    data-view-options="valuePath=ProductCategoryID,textPath=Name,
    dataSource=bind(filter.ParentCategories)"></select>
 *@
data-view-options="valuePath=ProductCategoryID,textPath=Name,
dataSource=bind(ParentCategories,productVM.filter)"></select>
```

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

The Converters

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

an example of a converter definition (typescript code):

```
export class UppercaseConverter extends RIAPP.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,
param=radioValue3,source=demoVM}" />
```

A special case is if a method of the converter (*any of it*) returns the *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 the undefined value

```
export class ListTypeConverter extends RIAPP.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 the *undefined* value when the value from the target is *false*. It prevents it from updating the property value on the source in this

case (*see the collections demo, only one radio button updates the source - the one which is checked, although they are databound to the single source property*).

Note: *Yo need to register a converter with the application or the bootstrap.*

```
app.registerConverter('sizeConverter', new SizeConverter());
```

Some predefined converters are already registered with the bootstrap in the converter.ts module

```
boot.registerConverter("BaseConverter", baseConverter);
boot.registerConverter("dateConverter", dateConverter);
boot.registerConverter("dateTimeConverter", dateTimeConverter);
boot.registerConverter("numberConverter", numberConverter);
boot.registerConverter("integerConverter", integerConverter);
boot.registerConverter("smallIntConverter", smallIntConverter);
boot.registerConverter("decimalConverter", decimalConverter);
boot.registerConverter("floatConverter", floatConverter);
boot.registerConverter("notConverter", new NotConverter());
```

Debugging databindings

The *jriapp.js* has a globally available variable *DebugLevel* which has a *DEBUG_LEVEL* type defined as:

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

It can be set in the framework configuration:

```
<script type="text/javascript">
      var jriapp_config = {
         frameworkPath: '@Url.Asset("/Scripts/jriapp/", true)',
         debugLevel: 0     // set the debug level here
      };

      var config = {
         baseUrl: "@Url.Asset("/Scripts/demo/", true)",
         waitSeconds: 10
      };
      requirejs.config(config);
</script>
```

When you want to perform a testing how the application works it is recommended to set the *DebugLevel* variable to the level above *NONE*. When the *DebugLevel* is *NORMAL = 1* than the unresolved databinding paths are outputted to the browser console.

When the *DebugLevel* is *HIGH* = 2 then if the path is unresolved, the javascript debugger kicks in.

2.5 *The Command class*

The Command provide the means for a declarative execution of methods defined on the view models. Element views can expose properties which accept the commands (*for example, the button element view or the anchor element view, for the clicks*).

an example of binding a custom command to the button command property:

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

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

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

```
this._uploadCommand = new RIAPP.Command(() => {
        try {
            self.uploadFiles(self._fileEl.files);
        } catch (ex) {
            self.handleError(ex, this);
        }
    }, () => {
        return self._canUpload();
});
```

The first parameter of the command constructor is a callback function (*the action*), which is invoked when the command is triggered by the UI element (*in this case when the button is clicked*).
The command action function accepts a parameter which can be explicitly provided in the data binding expression.

The second parameter is a callback function which returns a boolean result. It determines if the command is currently in the enabled or disabled state.

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

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

an example of using the command parameter:

```
<a class="btn btn-info btn-small" data-name="upload"
                    data-bind-1="this.command,to=dialogCommand,source=uploadVM"
                    data-bind-2="this.commandParam"
                    data-view-options="tip='click me to upload product thumbnail photo'">
                     Upload product thumbnail </a>
```

Note: In the above example {this.commandParam} expression binds commandParam property on the element view to the current data context, that is a product entity .

Using the command parameter in the command action (typescript code):

```
this._dialogCommand = new RIAPP.Command<DEMODB.Product>(function (product) {
        try {
            //using command parameter to provide the product item
            self._product = product;
            self.id = self._product.ProductID.toString();
            self._dialogVM.showDialog('uploadDialog', self);
        } catch (ex) {
            self.handleError(ex, self);
```

```
        }
    });
```

another example of using the command parameter:

```
<button  data-bind-1="this.command,to=paramCommand,source=testObject"
        data-bind-2="this.commandParam,source={color='Forest Green',r=34,g=139,b=34}">Color #3</button>
```

Using the command parameter in the command action (typescript code):

```
this._paramCommand = new RIAPP.Command<{ color: string; r: number; g: number; b: number; }>((param) => {
        alert(`${param.color}: #${RGBToHex(param.r, param.g, param.b)}`);
});
```

In some cases the command requires its own state or a complex logic which is better incapsulated. In this case the command is implemented in its own class inherited from the BaseCommand.

an example of a command implemented in its own class:

```
export class TestInvokeCommand extends RIAPP.BaseCommand<ProductViewModel, DEMODB.Product>
{
   protected action(param: DEMODB.Product) {
      const viewModel = this.owner;
      viewModel.invokeResult = null;
      const promise = viewModel.dbContext.serviceMethods.TestInvoke({ param1: [10, 11, 12, 13, 14], param2:
param.Name });
      promise.then((res) => {
         viewModel.invokeResult = res;
         viewModel.showDialog();
      });
   }
   protected isCanExecute(param: DEMODB.Product): boolean {
      const viewModel = this.owner;
      //just for the test: this command can be executed only when this condition is true!
      return viewModel.currentItem !== null;
   }
}
```

2.6 the *Element views*

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

Note: *You can not directly databind a HTML DOM element property, because you can only databind properties of an object derived from the framework BaseObject class. The Element views are objects which are all derived from the BaseObject, so they can expose properties which can be databound.*

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

For example, the StackPanelElView uses options to determine how it should be displayed - horizontally or vertically. The *TextBoxElView* uses the *updateOnKeyUp* option,

to decide when to update the databinding source – when the textbox loses the focus (*the default value*) or when the *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-options* attribute expression provides the options. You can also explicitly provide a view name and therefore to select which type of the element view will be created for the native DOM element, as in the example:

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

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

When the data binding expressions are evaluated, the application creates the element view for the html dom element. The explicit name in the *data-view* attribute overrides the default selection which type of the element view to create. Each element view must be registered with the bootstrap or the application before the framework can use it. Some element views are already registered in the *jriapp_ui* bundle and some element views are registered with several alternative names.

Registration of the element views in the jriapp_ui bundle:

```
boot.registerElView("generic", BaseElView);
boot.registerElView("baseview", BaseElView);
boot.registerElView("a", AnchorElView);
boot.registerElView("abutton", AnchorElView);
boot.registerElView("block", BlockElView);
boot.registerElView("div", BlockElView);
boot.registerElView("section", BlockElView);
boot.registerElView("busy", BusyElView);
boot.registerElView("busy_indicator", BusyElView);
boot.registerElView("input:button", ButtonElView);
boot.registerElView("input:submit", ButtonElView);
boot.registerElView("button", ButtonElView);
boot.registerElView("input:checkbox", CheckBoxElView);
boot.registerElView("checkbox", CheckBoxElView);
boot.registerElView("threeState", CheckBoxThreeStateElView);
boot.registerElView("checkbox3", CheckBoxThreeStateElView);
boot.registerElView(ELVIEW_NM.DataForm, DataFormElView);
boot.registerElView("datepicker", DatePickerElView);
bootstrap.registerElView("dynacontent", DynaContentElView);
bootstrap.registerElView("expander", ExpanderElView);
bootstrap.registerElView("input:hidden", HiddenElView);
bootstrap.registerElView("img", ImgElView);
boot.registerElView("select", ListBoxElView);
boot.registerElView("pager", PagerElView);
bootstrap.registerElView("input:radio", RadioElView);
bootstrap.registerElView("span", SpanElView);
boot.registerElView("stackpanel", StackPanelElView);
boot.registerElView("ul", StackPanelElView);
boot.registerElView("ol", StackPanelElView);
bootstrap.registerElView("tabs", TabsElView);
boot.registerElView("template", TemplateElView);
bootstrap.registerElView("textarea", TextAreaElView);
bootstrap.registerElView("input:text", TextBoxElView);
```

If the registration name is the same as the html tag name then there is no need to add the *data-view* attribute to the html markup. In that case, if the there is a registered element view with that name (*tag name*), then it will be created.

An element view can be simple, exposing only several properties of the HTML DOM element (*as the TextBoxElView*) or can be complex (*as the GridElView*).

All the element views are descendants of the *BaseElView*. The BaseElView accepts the options, which have the following interface:

```
export interface IViewOptions {
    css?: string;
    tip?: string;
    nodelegate?: boolean;
    errorsService?: IViewErrorsService;
}
```

The BaseElView exposes properties which are common to all element views and can be databound.
The *events, props, and classes* returns property bags. The property bag has the capability to provide the property value which it does not have.

Each property bag implements the IPropertyBag interface:

```
export interface IPropertyBag extends IBaseObject {
    getProp(name: string): any;
    setProp(name: string, val: any): void;
    isPropertyBag: boolean;
}
```

To get or set a property on the property bag in the databinding expression, it is needed to use the square brackets (*for simple objects the dot separator between the object and its properties is used*).

```
<a class="btn btn-info btn-small testBtn leftOffset"
        data-bind-1="this.events[click],to=testCommand,source=testObject"
        data-bind-2="this.toolTip,to=testToolTip,source=testObject"
        data-bind-3="this.props[text],to=testProperty1,source=testObject"
        data-bind-4="this.classes[testClass],to=isEnabled,source=testObject,converter=notConverter"
        data-view="generic">Test command 2</a>
```

The most often used property bag property is the *classes*, it can be used to set css styles on the html element. If you use the *classes[*]*, then it sets (*adds or removes*) a group of css styles:

```
<div class="someClass" data-bind-1="this.classes[*],to=Size,converter=sizeConverter"
data-bind-2="this.classes[bold],to=IsActive">
    Size: <span class="leftOffset" data-bind="this.value,to=Size"></span>
</div>

/*
 an example converter, which depending on the size returns
 an array of css classes to add or to remove to the HTML element
*/
const styles = ["lsize", 'msize', 'ssize', 'nsize'];
```

```
export class SizeConverter extends RIAPP.BaseConverter {
    convertToSource(val: any, param: any, dataContext: any): any {
        return undefined;
    }
    convertToTarget(val: any, param: any, dataContext: any): any {
        let size = "" + val, firstLetter: string;
        let res: string[] = undefined, found = false;
        if (!!val) {
            if (utils.check.isNumeric(size))
                firstLetter = 'n';
            else
                firstLetter = size.charAt(0).toLowerCase();
        }

        res = styles.map((style) => {
            // only check if not found (for optimization)
            if (!found && !!firstLetter && utils.str.startsWith(style, firstLetter)) {
                found = true;
                //adds this style to the classes
                return "+" + style;
            }
            else {
                //removes this style from the classes
                return "-" + style;
            }
        });

        /*
         if we return ['-*'] , it will remove all classes from the HTML element
          but here it is undesirable, because we would remove classes which were set elsewhere
        */

        return res;
    }
}
```

BaseElView - is the base class. It has has, besides the mentioned above, the *isVisible* and the *toolTip* properties.

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

TextBoxElView – besides the inherited properties, it exposes a *value* property, which exposes the text value of the DOM element. The default behaviour of this view is to update the value when the textbox loses the focus. It can be changed by using the *updateOnKeyUp* option, so the value is changed on each *keyup* event.

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

TextAreaElView – It has *isEnabled*, *value* (*to get or set text*), *wrap* properties.

```
<textarea data-bind="this.value,to=testProperty,mode=TwoWay,source=testObject1" wrap="soft"></textarea>
```

CheckBoxElView – It exposes the *checked* property of the HTML DOM element.

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

RadioElView – It exposes the *checked* property of the HTML DOM element - the same as *CheckBoxElView*. Usually the databinding expression for the radio element uses a converter, so that only one radio button (*which is checked*) updates the source. It also exposes the read only *name* property.

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

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

ButtonElView - is a descendant of the *CommandElView*. It exposes *value, text, html* properties of the button element. It also exposes a boolean *preventDefault* property. It can be used to choose if the button should trigger its default action or not. This element view command property can be data bound to a command implementation, so as to trigger an action when the button is clicked.

```
<button class="btn btn-info btn-small"
data-bind-1="this.command,to=paramCommand,source=testObject"
data-bind-2="this.commandParam,source={color='Cornflower',r=100,g=149,b=237}">Color #2</button>

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

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

AnchorElView - is a descendant of the *CommandElView*. The link can display a text or an image (*which can be defined 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 style="color: white;" data-name="Product Number Column"
data-bind1="this.command,to=columnCommand,source=productVM"
data-bind2="this.commandParam,to=currentItem,source=productVM"
data-view-options="stopPropagation=true">Product #</a>
```

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

```
<a href="#" data-bind="this.command,to=expanderCommand,source=headerVM"
data-view="expander"></a>
```

```
// an example of the definition of the command in the view model
 this._expanderCommand = new RIAPP.Command<{ isExpanded: boolean; }>((param) => {
         if (param.isExpanded) {
            this.expand();
         } else {
            this.collapse();
         }
 });
```

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

an example of the data template which uses the TemplateElView:

```
<script id="mailSendedTemplate" type="text/x-template">
   @*mail sended report template*@
   <div data-bind="this.command,to=templateCommand,source=sendListVM" data-view="template">
      <table>
         <tr>
            <td>
               <fieldset>
                  <legend>Report period</legend>
                  <label class="radio">
                     <input type="radio" name="period"
data-bind="this.checked,to=reportPeriod,mode=TwoWay,converter=channelConverter,param=all" />
                     <span>ALL</span>
                  </label>
                  <label class="radio">
                     <input type="radio" name="period"
data-bind="this.checked,to=reportPeriod,mode=TwoWay,converter=channelConverter,param=today" />
                     <span>TODAY</span>
                  </label>
               </fieldset>
            </td>
            <td>
               <fieldset>
                  <legend>The User</legend>
                  <label class="radio">
                     <input type="radio" name="reportuser"
data-bind="this.checked,to=reportUser,mode=TwoWay,converter=channelConverter,param=all" />
                     <span>ALL</span>
                  </label>
                  <label class="radio">
                     <input type="radio" name="reportuser"
data-bind="this.checked,to=reportUser,mode=TwoWay,converter=channelConverter,param=current" />
                     <span>CURRENT</span>
                  </label>
               </fieldset>
            </td>
         </tr>
      </table>


      @using (Html.BeginForm("MailSendedReport", "Report", FormMethod.Post, new { data_name = "form_Report",
target = "_blank" }))
      {
         <input type="hidden" name="d1" value="" />
         <input type="hidden" name="d2" value="" />
         <input type="hidden" name="user" value="" />
         <input type="hidden" name="sendListID" value="" />
      }
```

```
    </div>
</script>
```

an example of the command implementation:

```
// executed when template is loaded or unloading
this._templateCommand = new RIAPP.Command<uiMOD.TemplateCommandParam>((param) => {
        try {
            let t = param.template, templEl = t.el, formEl;
            if (t.templateID == "mailSendedTemplate") {
                formEl = $('form[data-name="form_Report"]', templEl);
                if (param.isLoaded) {
                    if (formEl.length == 0) {
                        this.handleError("formEl is null", self);
                        return;
                    }
                    self._reportForm = formEl;
                }
                else {
                    self._reportForm = null;
                }
            }
        } catch (ex) {
            self.handleError(ex, this);
        }

    }, () => {
        return !!self.selectedSendListID;
});
```

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

Warning: Data binding to the html property should be used carefully, because it inserts a HTML content inside the element. It should not be used to display the user input without first checking the content to prevent XSS attacks!

```
<span data-bind="this.value,to=testProperty1,source=testObject1"></span>
```

BlockElView - in addition to the properties inherited from the *SpanElView*, it adds *width, height* properties, which can be data bound. It is also registered by the name *block*, which can be provided in a data-view attribute. But for the <section/> and the 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 - It exposes the image's *src* property.

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

BusyElView - it exposes the *isBusy* and the *delay* property.
It is used to display an animated loader gif image above the content of the HTML DOM element to which it is attached.
The element view is registered by the name *busy_indicator*.

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

GridElView - is used to attach the logic and the markup of the *DataGrid* control to the HTML DOM element. It exposes the *dataSource* and the *grid* property. It is used to display the datagrid with the data obtained through the data source. All the properties are usually set using the element view options.

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*):

```
<script id="gridProductsOptions" type="text/x-options">
    resizeMode:overflow,
    wrapCss:productTableWrap,
    containerCss:productTableContainer,
    headerCss:productTableHeader,
    rowStateField:IsActive,
    isHandleAddNew:true,
    isCanEdit:true,
    editor:bind(dialogOptions,productVM),
    details:{templateID:productDetailsTemplate}
</script>

<table data-name="gridProducts" data-bind1="this.dataSource,to=dbSet,source=productVM"
          data-bind2="this.grid,to=grid,mode=BackWay,source=productVM"
          data-bind3="this.stateProvider,to=rowStateProvider,source=productVM"
          data-view="resizable_grid"
          data-view-options="get(gridProductsOptions)">
        <thead>
        <tr>
          <th data-column="width:35px,type:row_expander,tip='Row expander'"></th>
          <th data-column="width:50px,type:row_actions,tip='Row actions'"></th>
          <th data-column="width:40px,type:row_selector,tip='Column selector'"></th>
          <th data-column="width:100px,sortable:false,templateID:productNumberTemplate,tip='Product number column
implemented using template!!!'"
data-content="fieldName:ProductNumber,css:{readCss:'number-display',editCss:'number-edit'}"></th>
          <th data-column="width:25%,sortable:true,title:Name,tip='Product name'" data-
content="fieldName:Name"></th>
          <th data-
column="width:90px,title:'Weight',sortable:true,rowCellCss:weightCell,colCellCss:weightCol,tip='Example of a tooltip in
the datagrid column's header'" data-content="fieldName:Weight"></th>
          <th data-column="width:15%,title=CategoryID,sortable:true,sortMemberName=ProductCategoryID"
data-
content="fieldName=ProductCategoryID,name:lookup,options:{dataSource=dbContext.dbSets.ProductCategory,valuePat
h=ProductCategoryID,textPath=Name}"></th>
          <th data-column="width:100px,sortable:true,rowCellCss:dateCell,title='SellStartDate',tip='Start of the sales'"
data-content="fieldName=SellStartDate"></th>
          <th data-column="width:100px,sortable:true,rowCellCss:dateCell,title='SellEndDate',tip='End of the sales'"
data-content="fieldName=SellEndDate"></th>
          <th data-
column="width:90px,sortable:true,rowCellCss:activeCell,title='IsActive',sortMemberName=SellEndDate" data-
content="fieldName=IsActive"></th>
          <th data-column="width:10%,title=Size,sortable:true,sortMemberName=Size" data-
content="template={readID=sizeDisplayTemplate}"></th>
        </tr>
        </thead>
        <tbody></tbody>
</table>
```

PagerElView - 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 style="float: left;"
        data-bind-1="this.dataSource,to=dbSet,source=productVM"
@*when the pager is selected then the datagrid gets keydown and keyup events*@
        data-bind-2="this.parentControl,to=grid,source=productVM"
        data-view="pager"
        data-view-options="sliderSize:5,showTip:false,showInfo:true"></div>
```

StackPanelElView - is used to attach the logic and the markup of the *StackPanel* control to a block element. It exposes a *dataSource* and a *panel* properties. It is used to display horizontally or vertically stacked panels (*which use data templates*) on the page. The element view is registered by the name *stackpanel*.

an example of the markup for two StackPanels (*see the CollectionsDemo*):

```
<!--example of using stackpanel for vertical and horizontal list view-->
<div class="stackPanelV"
      data-bind="this.dataSource,to=historyList"
      data-view="stackpanel"
      data-view-options="templateID:stackPanelItemTemplateV,orientation:vertical"></div>

 <div class="stackPanelH"
      data-bind="this.dataSource,to=historyList"
      data-view="stackpanel"
      data-view-options="templateID:stackPanelItemTemplateH,orientation:horizontal"></div>
```

SelectElView - is used to attach the logic of the *ListBox* control to the HTML DOM element (*usually, the select html element*).
It exposes *isEnabled, dataSource, selectedValue, selectedItem*, and *listBox* properties. The data source of the element view provides the data to fill the list items (*the options*). The textProvider property alllows to attach a text converter for the options. The stateProvider property allows to attach a provider which returns the css style for the options.

an example of the HTML markup for the select elements:

```
<select   data-bind-1="this.dataSource,to=filter.ChildCategories"
                  data-bind-2="this.selectedValue,to=filter.childCategoryID,mode=TwoWay"
                  data-bind-3="this.selectedItem,to=filter.selectedCategory,mode=TwoWay"
                  data-bind-4="this.toolTip,to=filter.selectedCategory.Name"
                  data-bind-5="this.textProvider,to=optionTextProvider"
                  data-bind-6="this.stateProvider,to=optionStateProvider"
                  data-view-options="valuePath=ProductCategoryID,textPath=Name"></select>
```

DataFormElView - is used to attach the logic of the *DataForm* control for managing the data context (*see more info in the DataForm control section of this guide*). It exposes the *dataContext* and the *form* properties. It is used to display the data for editing and viewing purposes. The element view is registered by the name *dataform*. Inside the dataform you can use the data content attributes.

an example of the markup for the dataform:

```
 <div style="width: 100%" data-bind="this.dataContext" data-view="dataform">
     <table style="background-color: transparent; border: none; table-layout: fixed; width: 95%;">
       <colgroup>
         <col style="border: none; text-align: left; width: 225px;" />
         <col style="border: none; text-align: left; width: 100%;" />
       </colgroup>
       <tbody>
         <tr>
```

```
                <td>
                    ID:
                </td>
                <td>
                    <span data-content="fieldName:SalesOrderDetailID"></span>
                </td>
            </tr>
            <tr>
                <td>
                    OrderQty:
                </td>
                <td>
                    <span data-content="fieldName:OrderQty,css:{editCss:'qtyEdit'}"></span>
                </td>
            </tr>
            <tr>
                <td>
                    UnitPrice:
                </td>
                <td>
                    <span data-content="fieldName:UnitPrice"></span>
                </td>
            </tr>
            <tr>
                <td>
                    UnitPriceDiscount:
                </td>
                <td>
                    <span data-content="fieldName:UnitPriceDiscount"></span>
                </td>
            </tr>
            <tr>
                <td>
                    LineTotal:
                </td>
                <td>
                    <span data-content="fieldName:LineTotal,readOnly:true"></span>
                </td>
            </tr>
        </tbody>
    </table>
</div>
```

DatePickerElView - 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:

```
<script id="salesDateOptions" type="text/x-options">
   datepicker: { showOn:button,
   yearRange:'-15:c',
   changeMonth: true,
   changeYear:true,
   yearRange:'1998:2005',
   minDate:date(1998-06-01,YYYY-MM-DD),
   maxDate:date(20050707)
   }
</script>

 <input type="text" id="saleStart2" placeholder="Enter Date"
                    data-bind="this.value,to=filter.saleStart2,mode=TwoWay,converter=dateConverter"
                    data-view="datepicker"
                    data-view-options="get(salesDateOptions)"/>
```

TabsElView - attaches the JQuery UI Tabs plugin logic to the HTML DOM element. It exposes *tabsEvents* property. The element view is registered by the name *tabs*.

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

an example of the HTML markup for the tabs:

```
<div  data-bind="this.tabsEvents,to=tabsEvents,source=productVM" data-view="tabs">
    <div id="myTabs">
            <ul>
              <li>
                 <a href="#a">Some Info</a>
              </li>
              <li>
                 <a href="#b">Photo</a>
              </li>
              <li>
                 <a href="#c">Sales Order Details</a>
              </li>
            </ul>
            <div id="a">
                    <!--tab content here-->
            </div>
            <div id="b">
                    <!--tab content here-->
            </div>
            <div id="c">
                    <!--tab content here-->
            </div>
     </div>
</div>
```

an example of handling tabs events in the view model:

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

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

an example of the markup for the dynacontent:

```
<div  data-bind-1="this.templateID,to=viewName,source=customerVM.uiMainView"
data-bind-2="this.dataContext,source=customerVM"
data-bind-3="this.animation,to=animation,source=customerVM.uiMainView"
data-view="dynacontent"></div>
```

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

The Inject and the bind functions:

Sometimes there's a need to provide a service or a DbContext to the element view constructor. It is usually done declaratively and there are two options how to do it. In the declaration of an element view options you can use the *inject* or the *bind* functions.

The *inject* gets a registered service by the name. If you register a function instead of an object, then this function will be invoked to get the objects which it returns.

For example, the autocomplete uses the inject to get the dbContext which have been registered as:

```
// register the service in the code with the name $dbContext
this.registerSvc("$dbContext", this._dbContext);

<!--using the inject in the options-->
<script id="custInfoGroup.salespersonAutocompleteOptions" type="text/x-options">
   dbContext:inject($dbContext),
   fieldName:SalesPerson,
   dbSetName:SalesInfo,
   queryName:ReadSalesInfo,
   templateId:custInfoGroup.salePerAutocompleteTemplate,
   width:200px,
   height:200px
</script>
```

Also, the framework uses some rigistered services. They are registered as the functions which return the services:

```
boot.registerSvc(SERVICES.TOOLTIP_SVC, createToolTipSvc());
boot.registerSvc(SERVICES.DATEPICKER_SVC, createDatepickerSvc());
boot.registerSvc(SERVICES.UIERRORS_SVC, createUIErrorsSvc());
```

The *bind* gets an object parsing the path, just like the databinding expressing does.
It uses two parameters - the first is the path, the second (*optional*) is the source (*the same as defined in the databinding expressions*).
If the source is not provided, then the source is implicit.
If the source is provided, then it is fixed (*the rule is the same as for the source in the databinding expressions*).
For example, it is used it the TreeDemo to provide the datasource in the options.

```
<ul class="dynatree-container" data-view-
options="templateID:ItemListTemplate,syncSetDatasource:true,dataSource=bind(rootView,fbrowserVM)"></ul>
```

Note: *The bind can also be used as the converter parameter. In this case it is evaluated each time when the converter obtains the parameter.*

```
<!--using the bind function to provide the parameter for the converter-->
```

```
<span data-bind="this.value,to=Customer[ModifiedDate],converter=dateTimeConverter,param=bind(Customer.dateFormat)">
</span>
```

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

```
<!--using a file image element view for the display of a product image-->
<img style="float: left; max-height: 200px; max-width: 350px;"
                    data-bind-1="this.id,to=ProductID"
                    data-bind-2="this.fileName,to=ThumbnailPhotoFileName" alt="Product Image" src=""
                    data-view="fileImage" data-view-options="baseUri:'@Url.Action("DownloadThumbnail", "File", new
{httproute = "ApiByAction"})'" />

<!--using a month picker element view-->
 <input type="text" data-view="monthpicker"
data-view-options="tip='the selected Month and Year is converted to the date type using a converter'"
data-
bind="this.value,to=yearmonth,mode=TwoWay,source=testObject,converter=yearmonthConverter,param='MM/YYYY'"
              style="width:150px;" placeholder="MM/YYYY" />

<!--using a month autocomplete element view-->
 <input type="text" style="margin-left: 0px; width: 200px;"
            data-bind-1="this.value,to=SalesPerson,mode=TwoWay"
            data-bind-2="this.dataContext"
            data-view="autocomplete"
            data-view-options="get(custInfoGroup.salespersonAutocompleteOptions)" />
```

2.7 *Data templates*

The Data templates are pieces of the HTML markup which can be used by the UI controls for cloning their structure and displaying them on the page. A data template definition (*in the HTML markup*) must have the *id* attribute for referencing it in the options of the UI controls.
The templates are used by creating instances of the *Template* class. Internally, it retrieves the template's definition (*in the form of a string*) by the *id*, then it converts the string into the DOM. If the template uses databinding expressions then it parses and processes them.
The data template has the *dataContext* property which provides an ambient datacontext.

The framework contains several built-in controls (*element views*), which use the data templates in their code: *DataEditDialog, DataGrid, StackPanel, DynaContentElView*.

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

```
export function createTemplate(dataContext ?: any, templEvents?: ITemplateEvents): ITemplate {
   const options: ITemplateOptions = {
      dataContext: dataContext,
      templEvents: templEvents
   };
   return new Template(options);
}
```

The templates can also require some modules and css files to be preloaded dynamically before they are used. For this you can add the *data-require* attribute to the

top level html element inside the template and add a comma separated list of artifacts to load.

```
<script id="custInfoGroup.customerInfo" type="text/x-template">
      <div style="max-width: 800px;" data-bind="this.dataContext,to=currentItem"
        data-view="dataform" data-require="test1,test3,css!css/test1.css">
      <!--the other html elements here-->
   </div>
</script>
```

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

1) Defined on the page inside a script tag
2) Preloaded from the server at the start of the application.
3) Loaded on as needed basis (*registered with a loader function*)
4) Registered as a group of templates.

The first way - *defining templates on the html page*

The templates are defined in a script sections which has the id and the type attribute with the *text/x-template* value.

```
<script id="phoneTemplate" type="text/x-template">
   <span data-bind="this.value,to=Customer[Level1.Level2.Phone]"></span>
</script>
```

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

The second way - *loading all the templates from the server*

The templates are defined on the server in a text (*html*) file (*at least the url should provide a text content*). The rules for a templates definition are the same as in the first way. They are loaded by the application's *loadTemplates* method. (*See the DataGrid Demo for an example*).

```
// create and start application here
return bootstrap.startApp(() => { return new DemoApplication(options); }, (app) => {
          // an example of how to load a file with templates from the server
            app.loadTemplates(options.templates_url);
 }).then((app) => {
      // extra code here ...
 });
```

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

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

The templates can be loaded when they are needed. The application should register a loader function per template. The loader function must return a promise which is resolved (*if all is ok*) to a *html* string.

```
// create and start application here
return bootstrap.startApp(() => { return new DemoApplication(options); }, (app) => {
      // this registered function will be invoked every time when the template with that name is needed
```

```
    app.registerTemplateLoader('productEditTemplate', coreUtils.memoize(() => {
      return utils.http.getAjax(options.productEditTemplate_url); })
    );
}).then((app) => {
    // extra code here ...
});
```

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

The fourth way - *loading templates in groups*

The templates can be loaded in groups (*several templates per group*), after loading their definitions are automatically cached on the client. For each group of templates you register a unique group's name. The group name is used in the template name like *GroupName.TemplateName*. (*See the Single Page Application demo for an example*).

```
return RIAPP.bootstrap.startApp(() => {
      return new DemoApplication(options);
  }, (app) => {
      app.registerTemplateGroup('custGroup', options.spa_template1_url);

      app.registerTemplateGroup('custInfoGroup', options.spa_template2_url);

      app.registerTemplateGroup('custAdrGroup', options.spa_template3_url);

}).then((app) => {
        return app.customerVM.load();
});
```

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

Note: *These four methods of loading templates can be mixed in one application.*

2.8 *Data contents*

The *data content* is used for displaying specific content types in a predetermined way. For example, a boolean value can be displayed as a checkbox and a string value as a textbox. The numeric and the datetime values will be formatted as determined by the data content which displays them.
The values can display differently if they are in a editing or a reading state.

A *data-content* attribute can only be used inside the data forms or the data grids data cells (*otherwise it is ignored*).

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

an example of using data contents inside a data form:

```
<form action="#" style="width: 100%" data-bind="this.dataContext" data-view="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>FirstName:
              </td>
              <td>
                <span data-content="fieldName:FirstName,css:{readCss:'custInfo',editCss:'custEdit'}"></span>
              </td>
            </tr>
            <tr>
              <td>SalesPerson:
              </td>
              <td>
                <span data-content="template:{readID=salespersonTemplate1,editID=salespersonTemplate2},
                        css:{readCss:'custInfo',editCss:'custEdit'}"></span>
              </td>
            </tr>
          </tbody>
        </table>
</form>
```

There are two kinds of the data content :

1) which uses the fields directly.
2) which uses the data templates.

The first option (*using a field name*):

The simplest option, but the way how it is displayed is predefined by the field's data type. For example, when the field's data type is a text, then in the reading state the data content is rendered as a text inside the tag, and when in the editing state it is rendered in the <input type="text"/> tag. All these data content types are derived from *BindingContent* type. Currently, the framework includes: *BoolContent, DateContent, DateTimeContent, NumberContent, StringContent, MultyLineContent,* and *LookupContent* types.

A type for the creation of the instance of the data content is mainly determined by the data type of the field (*Number, String, Bool, Date*) to which the data content is data bound. The decision is made by the *ContentFactory*, which is used to create instances of the data content types in the application. But this decision can be changed by explicitly specifying the name of the data content and some of the data contents may also require (*the lookup data content*) the options for their initialization.

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

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

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

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

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

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

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

an example of specifying a readOnly for the data content:

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

The second option (*using the template*):

It is more versatile than the first option because the template can have more complex display. Since, inside the template you can use multiple databinding expressions.
The only drawback here - is that it needs more efforts than the first option.
The templated data content is implemented in the framework by a *TemplateContent* class.

an example of markup for a templated data content:

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

Note: *the editID or the readID can be omitted.*

Working with the data on the client side

3.1 *Working with the simple collection's data*

The framework collection hierarchy starts from the generic *BaseCollection* class, which is the base abstract class for all specialized collections. The *CollectionItem* is the base class for all items in these collections. Each collection item implements *ICollectionItem* interface

```
export interface ICollectionItem extends IBaseObject {
   readonly _aspect: IItemAspect<ICollectionItem, any>;
   readonly _key: string;
}
```

These base classes are defined in the *jriapp_shared* bundle, which has also *BaseList*, and *BaseDictionary* implementations.
The collection is used as the data source for the controls as the *DataGrid*, the *StackPanel*, the *ListBox*.
Every collection instance has the *currentItem* property and events which notify the controls about changing the current position, adding or removing an item, and also about starting or ending of editing of the item. Only one item in the collection can be in the editing state.

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

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

```
export interface IItemAspect<TItem extends ICollectionItem, TObj extends IIndexer<any>> extends IBaseObject,
IErrorNotification, IEditable, ISubmittable {
    getFieldInfo(fieldName: string): IFieldInfo;
    getFieldNames(): string[];
    getErrorString(): string;
    deleteItem(): boolean;
    _setKey(v: string): void;
    _setIsAttached(v: boolean): void;
    raiseErrorsChanged(): void;
    readonly vals: TObj;
    readonly item: TItem;
    readonly key: string;
    readonly coll: ICollection<TItem>;
    readonly status: ITEM_STATUS;
    readonly isUpdating: boolean;
    readonly isEditing: boolean;
    readonly isCanSubmit: boolean;
    readonly isHasChanges: boolean;
    readonly isNew: boolean;
    readonly isDeleted: boolean;
    readonly isEdited: boolean;
    readonly isDetached: boolean;
}
```

The descendants of the collection type:

BaseList – is a client side implementation for the list of objects.
BaseDictionary – It extends the *BaseList* and has also the *keyName* parameter in constructor arguments. The items are indexed by the key and can be found (obtained) by their keys.

The *BaseList* and the *BaseDictionary* have a *toArray* method which returns an array of simple objects instead of the collection items.

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

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

```
this._months = new DEMODB.KeyValDictionary();
this._months.fillItems([{ key: 1, val: 'January' }, { key: 2, val: 'February' }, { key: 3, val: 'March' },
      { key: 4, val: 'April' }, { key: 5, val: 'May' }, { key: 6, val: 'June' },
      { key: 7, val: 'July' }, { key: 8, val: 'August' }, { key: 9, val: 'September' }, { key: 10, val: 'October' },
      { key: 11, val: 'November' }, { key: 12, val: 'December' }], true);
```

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

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

On the client side, the components that work with the DomainService are implemented in the *jriapp_db* bundle. The DbContext is the component which communicates with the DomainService.

The DbContext:

The instance of the DbContext is used to communicate with the data service.
The DbContext stores the data in DbSets.

The DbContext prevents repeated loading of the same entities into the DbSet. If the entity is loaded the second time it does not replace the entity in the collection but only refreshes the data in it. The DbContext checks the uniqueness on entities by comparing their primary keys. Each entity in the DbSet must have a primary key.

The DbContext is an abstract class and is not used directly. Instead, the application uses a class derived from the DbContext class. The DataService *code generation* feature can create a script with strongly typed entity classes and a strongly typed DbContext.

an example of a concrete DbContext class (generated by the service):

```
export class DbContext extends dbMOD.DbContext<DbSets, ISvcMethods, IAssocs>
{
   protected _createDbSets(): DbSets {
      return new DbSets(this);
   }
   protected _createAssociations(): dbMOD.IAssociationInfo[] {
      return [{ "name": "CustAddrToAddress", "parentDbSetName": "Address", "childDbSetName": "CustomerAddress",
"childToParentName": "Address", "parentToChildrenName": "CustomerAddresses", "onDeleteAction": 0, "fieldRels":
[{ "parentField": "AddressID", "childField": "AddressID" }] }, { "name": "CustAddrToAddress2", "parentDbSetName":
"AddressInfo", "childDbSetName": "CustomerAddress", "childToParentName": "AddressInfo", "parentToChildrenName":
"CustomerAddresses", "onDeleteAction": 0, "fieldRels": [{ "parentField": "AddressID", "childField": "AddressID" }] },
{ "name": "CustAddrToCustomer", "parentDbSetName": "Customer", "childDbSetName": "CustomerAddress",
"childToParentName": "Customer", "parentToChildrenName": "CustomerAddresses", "onDeleteAction": 0, "fieldRels":
[{ "parentField": "CustomerID", "childField": "CustomerID" }] }, { "name": "OrdDetailsToOrder", "parentDbSetName":
"SalesOrderHeader", "childDbSetName": "SalesOrderDetail", "childToParentName": "SalesOrderHeader",
"parentToChildrenName": "SalesOrderDetails", "onDeleteAction": 1, "fieldRels": [{ "parentField": "SalesOrderID",
"childField": "SalesOrderID" }] }, { "name": "OrdDetailsToProduct", "parentDbSetName": "Product", "childDbSetName":
"SalesOrderDetail", "childToParentName": "Product", "parentToChildrenName": "SalesOrderDetails", "onDeleteAction":
0, "fieldRels": [{ "parentField": "ProductID", "childField": "ProductID" }] }, { "name": "OrdersToBillAddr",
"parentDbSetName": "Address", "childDbSetName": "SalesOrderHeader", "childToParentName": "Address1",
"parentToChildrenName": null, "onDeleteAction": 0, "fieldRels": [{ "parentField": "AddressID", "childField":
"BillToAddressID" }] }, { "name": "OrdersToCustomer", "parentDbSetName": "Customer", "childDbSetName":
"SalesOrderHeader", "childToParentName": "Customer", "parentToChildrenName": null, "onDeleteAction": 0,
"fieldRels": [{ "parentField": "CustomerID", "childField": "CustomerID" }] }, { "name": "OrdersToShipAddr",
```

```
"parentDbSetName": "Address", "childDbSetName": "SalesOrderHeader", "childToParentName": "Address",
"parentToChildrenName": null, "onDeleteAction": 0, "fieldRels": [{ "parentField": "AddressID", "childField":
"ShipToAddressID" }] }];
    }
    protected _createMethods(): dbMOD.IQueryInfo[] {
        return [{ "methodName": "ReadAddress", "parameters": [], "methodResult": true, "isQuery": true }, { "methodName":
"ReadAddressByIds", "parameters": [{ "name": "addressIDs", "dataType": 3, "isArray": true, "isNullable": false,
"dateConversion": 0, "ordinal": 0 }], "methodResult": true, "isQuery": true }, { "methodName":
"ReadAddressForCustomers", "parameters": [{ "name": "custIDs", "dataType": 3, "isArray": true, "isNullable": false,
"dateConversion": 0, "ordinal": 0 }], "methodResult": true, "isQuery": true }, { "methodName": "ReadAddressInfo",
"parameters": [], "methodResult": true, "isQuery": true }, { "methodName": "ReadCustomer", "parameters": [{ "name":
"includeNav", "dataType": 2, "isArray": false, "isNullable": true, "dateConversion": 0, "ordinal": 0 }], "methodResult":
true, "isQuery": true }, { "methodName": "ReadCustomerAddress", "parameters": [], "methodResult": true, "isQuery":
true }, { "methodName": "ReadCustomerJSON", "parameters": [], "methodResult": true, "isQuery": true },
{ "methodName": "ReadProduct", "parameters": [{ "name": "param1", "dataType": 3, "isArray": true, "isNullable": false,
"dateConversion": 0, "ordinal": 0 }, { "name": "param2", "dataType": 1, "isArray": false, "isNullable": false,
"dateConversion": 0, "ordinal": 1 }], "methodResult": true, "isQuery": true }, { "methodName": "ReadProductByIds",
"parameters": [{ "name": "productIDs", "dataType": 3, "isArray": true, "isNullable": false, "dateConversion": 0, "ordinal":
0 }], "methodResult": true, "isQuery": true }, { "methodName": "ReadProductCategory", "parameters": [], "methodResult":
true, "isQuery": true }, { "methodName": "ReadProductLookUp", "parameters": [], "methodResult": true, "isQuery":
true }, { "methodName": "ReadProductModel", "parameters": [], "methodResult": true, "isQuery": true },
{ "methodName": "ReadSalesInfo", "parameters": [], "methodResult": true, "isQuery": true }, { "methodName":
"ReadSalesOrderDetail", "parameters": [], "methodResult": true, "isQuery": true }, { "methodName":
"ReadSalesOrderHeader", "parameters": [], "methodResult": true, "isQuery": true }, { "methodName":
"TestComplexInvoke", "parameters": [{ "name": "info", "dataType": 0, "isArray": false, "isNullable": false,
"dateConversion": 0, "ordinal": 0 }, { "name": "keys", "dataType": 0, "isArray": true, "isNullable": false,
"dateConversion": 0, "ordinal": 1 }], "methodResult": false, "isQuery": false }, { "methodName": "TestInvoke",
"parameters": [{ "name": "param1", "dataType": 10, "isArray": false, "isNullable": false, "dateConversion": 0, "ordinal":
0 }, { "name": "param2", "dataType": 1, "isArray": false, "isNullable": false, "dateConversion": 0, "ordinal": 1 }],
"methodResult": true, "isQuery": false }];
    }
}
```

an example of creation of a strongly typed DbContext:

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

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

```
load() {
    //clear selected items
    this._clearSelection();
    //you can create several methods on the service which return the same entity type
    //but they must have different names (no overloads)
    //the query'service method can accept additional parameters which you can supply with the query
    let query = this.dbSet.createReadProductQuery({ param1: [10, 11, 12, 13, 14], param2: 'Test' });
    query.pageSize = 50;
    COMMON.addTextQuery(query, 'ProductNumber', this._filter.prodNumber);
    COMMON.addTextQuery(query, 'Name', this._filter.name);

    if (!utils.check.isNt(this._filter.childCategoryID)) {
        query.where('ProductCategoryID', RIAPP.FILTER_TYPE.Equals, [this._filter.childCategoryID]);
    }
    if (!utils.check.isNt(this._filter.modelID)) {
        query.where('ProductModelID', RIAPP.FILTER_TYPE.Equals, [this._filter.modelID]);
    }

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

```
        query.where('SellStartDate', RIAPP.FILTER_TYPE.LtEq, [this._filter.saleStart2]);
    switch (this.filter.size) {
        case 0: //EMPTY
            query.where('Size', RIAPP.FILTER_TYPE.Equals, [null]);
            break;
        case 1: //NOT EMPTY
            query.where('Size', RIAPP.FILTER_TYPE.NotEq, [null]);
            break;
        case 2: //SMALL SIZE
            query.where('Size', RIAPP.FILTER_TYPE.StartsWith, ['S']);
            break;
        case 3: //BIG SIZE
            query.where('Size', RIAPP.FILTER_TYPE.StartsWith, ['X']);
            break;
        default: //ALL
            break;
    }
    query.orderBy('Name').thenBy('SellStartDate', RIAPP.SORT_ORDER.DESC);
    return query.load();
}
```

The DbContext also allows to execute methods on the DataService annotated with the *Invoke* attribute. They are useful to execute some arbitrary method defined on the data service. The methods can accept arguments (*complex types and arrays are allowed*) and can return a result. The DataService code generation feature generates strongly typed versions of the methods, which can be easyly used from the client side code.

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

```
export interface ISvcMethods extends dbMOD.TServiceMethods {
    TestComplexInvoke: (args: {
        info: IAddressInfo2;
        keys: IKeyVal[];
    }) => RIAPP.IPromise<void>;
    TestInvoke: (args: {
        param1: number[];
        param2: string;
    }) => RIAPP.IPromise<string>;
}
```

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

```
viewModel.invokeResult = null;
const promise = viewModel.dbContext.serviceMethods.TestInvoke({ param1: [10, 11, 12, 13, 14], param2:
param.Name });
promise.then((res) => {
        viewModel.invokeResult = res;
        viewModel.showDialog();
    });
```

Caching several pages of the data on the client:

The DbContext can load the data in pages. Only one data page is displayed in the UI controls such as the DataGrid. They usually display not more than 200 rows per page. But sometimes you need to preload more rows to the client side in one query.
For this, you can set the query's *loadPageCount* property to the value more than 1. If the *loadPageCount* value is more than 1 then the query operation returns several pages of the data. The extra pages of the data are cached inside the query instance with the help of the DataCache.

If a DbSet's *pageIndex* property value is changed (*for example, when going to another data page in the data grid*), then instead of loading the data from the server it is served from the local cache. If there is no data for the page in the cache, then it is loaded from the server.

Note: *the local data caching is very useful if the query execution is slow and the navigation from one page to another takes a considerable time. In that case it is better to preload several pages of the data to the client in one query operation.*

an example of a query to return several pages at once

```
var query = this._dbSet.createReadCustomerQuery();

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

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

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

an example of loading of the related entities

```
this._orderVM.dbSet.addOnFill(function (_s, args) {
        self.loadAddressesForOrders(args.items);
    }, self.uniqueID);
```

Note: *The DbSet also has the **loaded** event which you can use to get all the data loaded from the server. The **fill** event only has the items for the current data page, although it can load at once more pages (if the loadPageCount > 1) and the extra data goes to the data cache.*

an example of using the loaded event

```
self._batchesDB.addOnLoaded((_s, args) => {
        self._batchKeys = {};
        self._batchStatus = {};
        args.vals.forEach(function (item) {
            self._batchKeys[item.BatchID] = void 0;
        });
        self.loadErrors();
        self.loadMekReplies();
        self._canSend = self._batchesDB.count > 0;
        self.startSendCmd.raiseCanExecuteChanged();
 });
```

DbSets:

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

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

```
// on the HTML page
ops.modelData = @Html.Action("ProductModelData", "RIAppDemoService");
ops.categoryData = @Html.Action("ProductCategoryData", "RIAppDemoService");

// on view model - setting the data
set modelData(data: {
      names: dbMOD.IFieldName[];
      rows: dbMOD.IRowData[];
   }) { this.ProductModels.fillData(data); }
set categoryData(data: {
      names: dbMOD.IFieldName[];
      rows: dbMOD.IRowData[];
      }) { this.ProductCategories.fillData(data); }
```

Calculated fields:

The DbSets could contain calculated fields (*its name is defined in the metadata*) and the strongly typed DbSet would have a strongly typed method to define the calculated field.

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

an example of a calculated field implementation:

```
self.dbContext.dbSets.FileSystemObject.defineExtraPropsField(function (item) {
        let res = <ExProps>item._aspect.getCustomVal("exprop");
        if (!res) {
            res = new ExProps(item, self.dbContext);
            item._aspect.setCustomVal("exprop", res);
            res.addOnClicked((s, a) => { self._onItemClicked(a.item); });
            res.addOnDblClicked((s, a) => { self._onItemDblClicked(a.item); });
        }

        return res;
    });
```

Entities:

The *CollectionItem* is a base class for the entity.
Basicaly, an entity is a collection item which is specific for the DbSet class. Concrete implementations of the entities have all the properties defined in the metadata for the DbSet.
The entities can also have navigation properties which are added by the associations.
You can see for example, in the Demo application- in the file RIAppDemo.BLL\Metadata\MainDemo2.xml

This is a file in which every DbSet used by the DataService is defined in xml format. It simplifies the editing of the metadata, because xml is more readable than a code. The metadata is stored on the server and a part of it is also available on the client and it is used to generate strongly typed classes.

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

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

```
<data:DbSetInfo dbSetName="Customer" isTrackChanges="True" enablePaging="True" pageSize="25"
                EntityType="{x:Type dal:Customer}">
    <data:DbSetInfo.fieldInfos>
      <data:Field fieldName="CustomerID" dataType="Integer" maxLength="4" isNullable="False"
isAutoGenerated="True"
                isReadOnly="True" isPrimaryKey="1" />
      <data:Field fieldName="NameStyle" dataType="Bool" maxLength="1" isNullable="False" />
      <data:Field fieldName="Title" dataType="String" maxLength="8" />
      <data:Field fieldName="Suffix" dataType="String" maxLength="10" />
      <data:Field fieldName="CompanyName" dataType="String" maxLength="128" />
      <data:Field fieldName="SalesPerson" dataType="String" maxLength="256" />
      <data:Field fieldName="PasswordHash" dataType="String" maxLength="128" isNullable="False"
                isAutoGenerated="True" isReadOnly="True" />
      <data:Field fieldName="PasswordSalt" dataType="String" maxLength="10" isNullable="False"
isAutoGenerated="True"
                isReadOnly="True" />
      <data:Field fieldName="rowguid" dataType="Guid" maxLength="36" isNullable="False" isAutoGenerated="True"
                isReadOnly="True" fieldType="RowTimeStamp" />
      <data:Field fieldName="ModifiedDate" dataType="DateTime" maxLength="8" isNullable="False"
                isAutoGenerated="True" isReadOnly="True" />
      <data:Field fieldName="ComplexProp" fieldType="Object">
        <data:Field.nested>
          <data:Field fieldName="FirstName" dataType="String" maxLength="50" isNullable="False" />
          <data:Field fieldName="MiddleName" dataType="String" maxLength="50" />
          <data:Field fieldName="LastName" dataType="String" maxLength="50" isNullable="False" />
          <data:Field fieldName="Name" dataType="String" fieldType="Calculated"
                  dependentOn="ComplexProp.FirstName,ComplexProp.MiddleName,ComplexProp.LastName" />
          <data:Field fieldName="ComplexProp" fieldType="Object">
            <data:Field.nested>
              <data:Field fieldName="EmailAddress" dataType="String" maxLength="50"
                      regex="^[_a-z0-9-]+(\.[_a-z0-9-]+)*@[a-z0-9-]+(\.[a-z0-9-]+)*(\.[a-z]{2,4})$" />
              <data:Field fieldName="Phone" dataType="String" maxLength="25" />
            </data:Field.nested>
          </data:Field>
        </data:Field.nested>
      </data:Field>
      <data:Field fieldName="AddressCount" dataType="Integer" fieldType="ServerCalculated" />
    </data:DbSetInfo.fieldInfos>
</data:DbSetInfo>
```

an example of an association definition (in XAML):

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

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

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

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

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

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

//  if the changes are not committed to the server they can still be rejected using the rejectChanges method

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

If you assign a new value to a field, and the entity is not in editing state, then the *beginEdit* method is called implicitly and the entity goes into the editing state.
On each call to the *beginEdit* or *endEdit* method, it causes for 'begin_edit' or 'end_edit' events to be triggered. In order to prevent triggering those events, for example, when you want to update many DbSet's items at once, you can use the DbSet's *isUpdating* property.

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

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

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

Calculated fields:

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

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

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

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

Client fields:

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

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

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

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

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

Server side calculated fields:

Server side calculated fields are used to provide a property on the entity which is calculated on the server and is used on the client for information purposes. Their changes are not submitted to the server (*even if you change them on the client their updates will be ignored*). They can be used for different purposes, when the fields values can be only obtained on the server side (*or if it easier to do*). You can even set the real database fields on the entities to a *ServerCalculated* field type. In that case the fields will function on the client side like *ClientOnly* fields. Only their values will be initially set from the server side.
They are declared on the server in the DbSet's schema.

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

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

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

Navigation fields:

An entity can also have navigation properties. They are based on foreign key relationship between the entities. The foreign key relationship in the framework are encapsulated in the association type. The relationship (*parent - child*) are defined in the 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 define (*optionally*) the names of the navigation fields, and if they were defined, then the association definition adds them to the corresponding, generated for the client side, entities. A parent entity can get (*using its navigation field*) an array of child entities and a child entity can get its parent entity (*for this they must be already loaded in the dbSets*).

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 (*the keys are usually generated on the server*), then assign the primary key values to the child entities' foreign key fields and then submit them to the server in a second batch.

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

an example of assigning parent entity to the navigation field:

```
var cust = this.currentCustomer;
var ca = this.custAdressView.addNew(); // create a new entity: CustomerAddress
ca.CustomerID = cust.CustomerID;
ca.AddressType = "Main Office"; // the default value, the user can edit it later
ca.Address = address; // assign parent entity - it can also be new and has no Primary key - it is fixed on submit
ca._aspect.endEdit();

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

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

Complex type fields:

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

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

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

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

Entity and fields validations:

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

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

an example of the custom client side data validation:

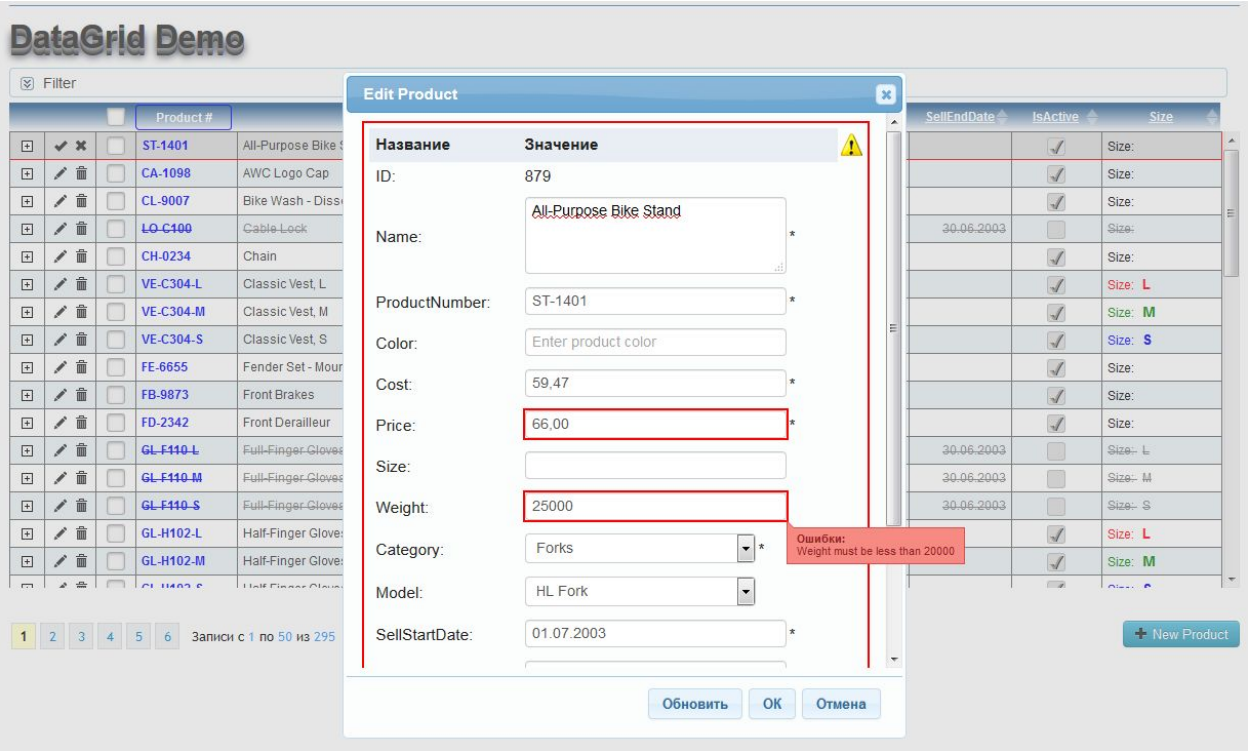
```
// example of using custom validation on client (in addition to a built-in validation)
 const validations = [{
        fieldName: <string>null, fn: (item: DEMODB.Product, errors: string[]) => {
           if (!!item.SellEndDate) { //check it must be after Start Date
              if (item.SellEndDate < item.SellStartDate) {
                 errors.push('End Date must be after Start Date');
              }
           }
        }
     },
     {
        fieldName: "Weight", fn: (item: DEMODB.Product, errors: string[]) => {
           if (item.Weight > 20000) {
              errors.push('Weight must be less than 20000');
           }
        }
     }];

     // example of using custom validation on client (in addition to a built-in validation)
     this._dbSet.addOnValidateField(function (_s, args) {
        let item = args.item;
        validations.filter((val) => {
           return args.fieldName === val.fieldName;
        }).forEach((val) => {
           val.fn(item, args.errors);
        });
     }, self.uniqueID);


     this._dbSet.addOnValidateItem(function (_s, args) {
        let item = args.item;
        validations.filter((val) => {
           return !val.fieldName;
        }).forEach((val) => {
           let errors: string[] = [];
           val.fn(item, errors);
           if (errors.length > 0) {
              args.result.push({ fieldName: null, errors: errors });
           }
        });
     }, self.uniqueID);
```

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

The databindings handle validation errors and update element views to display the errors.
If the entity has errors, the UI control (*DataGrid, DataForm*) will display error notifications (*red borders and tooltips on mouse hovering*) next to not the fields which have errors 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 the *cancelEdit* method.

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

DataView:

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

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

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

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

If you don't provide the *fn_itemsProvider* then the data is taken from the *dataSource* directly, otherwise the data is taken by executing the *fn_itemsProvider*. Then, if you provide the *fn_filter* the data is filtered with this function, and if you provide the *fn_sort* the data is sorted (*get data -> filter -> sort*).

an example of the DataView initialization:

```
// the view to filter addresses related to the current customer
  this._addressesView = new dbMOD.DataView<DEMODB.Address>(
        {
           dataSource: this._addressesDb,
           fn_sort: function (a, b) { return a.AddressID - b.AddressID; },
           fn_filter: function (item) {
              if (!self._currentCustomer)
                 return false;
              return item.CustomerAddresses.some(function (ca) {
                 return self._currentCustomer === ca.Customer;
              });
           },
           fn_itemsProvider: function (ds) {
              if (!self._currentCustomer)
                 return [];
              let 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 dbMOD.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 property is the dataSource - which is an instance of the collection which will be wrapped up with the DataView. You can omit the sorting and filtering functions if you don't need to filter or sort the data.

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

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

the Association and the ChildDataView:

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

an example of the metadata for a DbSet:

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

        <data:Metadata.Associations>
          <data:Association name="ChildToParent" parentDbSetName="FileSystemObject"
childDbSetName="FileSystemObject" childToParentName="Parent" parentToChildrenName="Children"
onDeleteAction="Cascade" >
            <data:Association.fieldRels>
              <data:FieldRel parentField="Key" childField="ParentKey"></data:FieldRel>
            </data:Association.fieldRels>
          </data:Association>
        </data:Metadata.Associations>
</data:Metadata>
```

When you define an association in the metadata, you define which field in a child entity relates to the key field in a parent entity. You can also give names for navigation properties (*parentToChildrenName and childToParentName*). These navigation properties, with the names which you have defined, will be added to the generated entity classes.

an example of getting the association and the ChildDataView instantiation:

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

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

When you want to display details for a parent entity, you should assign the ChildDataView's *parentItem* property. It triggers refreshing of the view with the new data.

After assigning the *parentItem* property the view will contain only details (*child entities*) for the parent entity.

```
this._customerVM.objEvents.onProp('currentItem', function (_s, args) {
        self._currentCustomer = self._customerVM.currentItem;
        self._custAdressView.parentItem = self._currentCustomer;
        self.objEvents.raiseProp('currentCustomer');
    }, self.uniqueID);
```

You can also use the association directly to get the parent or the child items related to the entity. For this the association has the *getChildItems(parent: IEntityItem)* and the *getParentItem(item: IEntityItem)* methods, which do this.

Working with the data on the server side

4.1 *The Data service*

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

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

```
public interface IDomainService : IDisposable
 {
    //provides the code generation implemented by providers (csharp, xaml, typescript etc.)
    string ServiceCodeGen(CodeGenArgs args);

    //information about permissions to execute service operations for the client
    Permissions ServiceGetPermissions();
    //information about service methods, DbSets and their fields information
    MetadataResult ServiceGetMetadata();

    Task<QueryResponse> ServiceGetData(QueryRequest request);
    Task<ChangeSet> ServiceApplyChangeSet(ChangeSet changeSet);
    Task<RefreshInfo> ServiceRefreshRow(RefreshInfo rowInfo);
    Task<InvokeResponse> ServiceInvokeMethod(InvokeRequest invokeInfo);
 }
```

The interface contains the methods which are invoked from the client (*using the DbContext*).
The data service is usually hosted in the ASP.NET MVC framework owing to the framework convenient design. The server part of the framework consists of 6 projects:

1) RIAPP.DataService - the main implementation of the data service. It implements the *RIAPP.DataService.BaseDomainService*.
2) RIAPP.DataService.EF - implements the *RIAPP.DataService.EF.EFDomainService<TDB>* derived from the *BaseDomainService* and can be used to work with Microsoft Entity Framework 4 domain models.
3) RIAPP.DataService.EF2 - implements the *RIAPP.DataService.EF2.EFDomainService<TDB>* derived from the *BaseDomainService* and can be used to work with Microsoft Entity Framework 6 domain models which use DbContext instead of ObjectContext.

4) RIAPP.DataService.LinqSql - implements the *RIAPP.DataService.LinqSql.LinqForSqlDomainService<TDB>* derived from the *BaseDomainService* and can be used with Microsoft Linq For SQL domain models.
5) RIAPP.DataService.Mvc - is used to integrate the dataservice with the ASP.NET MVC. Namely it has the ASP.NET MVC controller which exposes the DataService methods.
6) RIAPP.DataService.WebApi - is used to integrate the dataservice with the ASP.NET WebApi. It defines the WebApi Controller which exposes the DataService methods.

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

the implementation of the DataServiceController:

```
[NoCache]
[SessionState(SessionStateBehavior.Disabled)]
public abstract class DataServiceController<T> : Controller
      where T : BaseDomainService
   {
      private Lazy<IDomainService> _DomainService;

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

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

      public ISerializer Serializer { get; private set; }

      [ActionName("typescript")]
      [HttpGet]
      public ActionResult GetTypeScript()
      {
         var comment = string.Format(
               "\tGenerated from: {0} on {1:yyyy-MM-dd} at {1:HH:mm}\r\n\tDon't make manual changes here, they will
be lost when this interface will be regenerated!",
               ControllerContext.HttpContext.Request.RawUrl, DateTime.Now);
         var content = DomainService.ServiceCodeGen(new CodeGenArgs("ts") { comment = comment });
         var res = new ContentResult();
         res.ContentEncoding = Encoding.UTF8;
         res.ContentType = MediaTypeNames.Text.Plain;
         res.Content = content;
         return res;
      }

      [ActionName("xaml")]
      [HttpGet]
      public ActionResult GetXAML(bool isDraft = true)
      {
         var content = DomainService.ServiceCodeGen(new CodeGenArgs("xaml") { isDraft = isDraft });
         var res = new ContentResult();
         res.ContentEncoding = Encoding.UTF8;
         res.ContentType = MediaTypeNames.Text.Plain;
         res.Content = content;
         return res;
      }
```

```
[ActionName("csharp")]
[HttpGet]
public ActionResult GetCSharp()
{
    var content = DomainService.ServiceCodeGen(new CodeGenArgs("csharp"));
    var res = new ContentResult();
    res.ContentEncoding = Encoding.UTF8;
    res.ContentType = MediaTypeNames.Text.Plain;
    res.Content = content;
    return res;
}

protected virtual IDomainService CreateDomainService()
{
    return this.CreateDomainService(null);
}

protected virtual IDomainService CreateDomainService(Action<IServiceOptions> args)
{
    Action<IServiceOptions> action = (options) =>
    {
        options.Serializer = this.Serializer;
        options.User = this.User;
        if (args != null)
            args(options);
    };
    var service = (IDomainService)Activator.CreateInstance(typeof(T), action);
    return service;
}

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

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

[ActionName("permissions")]
[HttpGet]
public ActionResult GetPermissions()
{
    var res = DomainService.ServiceGetPermissions();
    return new ChunkedResult<Permissions>(res, Serializer);
```

```
    }

    [ActionName("query")]
    [HttpPost]
    public async Task<ActionResult> PerformQuery([SericeParamsBinder] QueryRequest request)
    {
      var res = await DomainService.ServiceGetData(request).ConfigureAwait(false);
      return new ChunkedResult<QueryResponse>(res, Serializer);
    }

    [ActionName("save")]
    [HttpPost]
    public async Task<ActionResult> Save([SericeParamsBinder] ChangeSet changeSet)
    {
      var res = await DomainService.ServiceApplyChangeSet(changeSet).ConfigureAwait(false);
      return new ChunkedResult<ChangeSet>(res, Serializer);
    }

    [ActionName("refresh")]
    [HttpPost]
    public async Task<ActionResult> Refresh([SericeParamsBinder] RefreshInfo refreshInfo)
    {
      var res = await DomainService.ServiceRefreshRow(refreshInfo).ConfigureAwait(false);
      return new ChunkedResult<RefreshInfo>(res, Serializer);
    }

    [ActionName("invoke")]
    [HttpPost]
    public async Task<ActionResult> Invoke([SericeParamsBinder] InvokeRequest invokeInfo)
    {
      var res = await DomainService.ServiceInvokeMethod(invokeInfo).ConfigureAwait(false);
      return new ChunkedResult<InvokeResponse>(res, Serializer);
    }

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

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

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

For example, in the *EFDomainService* class the *ExecuteChangeSet* method saves the updates inside a transaction's scope.

```
protected override async Task ExecuteChangeSet()
{
        try
        {
          using (var transScope = new TransactionScope(TransactionScopeOption.RequiresNew,
            new TransactionOptions
            {
```

```
                IsolationLevel = IsolationLevel.ReadCommitted,
                Timeout = TimeSpan.FromMinutes(1.0)
            }, TransactionScopeAsyncFlowOption.Enabled))
        {
            await DB.SaveChangesAsync().ConfigureAwait(false);

            transScope.Complete();
        }
    }
    catch (DbEntityValidationException e)
    {
        var sb = new StringBuilder();
        foreach (var eve in e.EntityValidationErrors)
        {
            sb.AppendFormat("Entity of type \"{0}\" in state \"{1}\" has the following validation errors:",
                eve.Entry.Entity.GetType().Name, eve.Entry.State);
            sb.AppendLine();
            foreach (var ve in eve.ValidationErrors)
            {
                sb.AppendFormat("- Property: \"{0}\", Error: \"{1}\"",
                    ve.PropertyName, ve.ErrorMessage);
            }
        }
        throw new Exception(sb.ToString());
    }
    await AfterExecuteChangeSet();
}
```

The *GetMetadata* method should return the *RIAPP.DataService.Types.Metadata*. Specialized data services classes which work with Microsoft Linq for SQL and Microsoft Entity Framework (*defined in RIAPP.DataService.Linq and RIAPP.DataService.EF respectively*) override this method. They take the underlying *System.Data.Linq.DataContext* or *System.Data.Objects.ObjectContext* instance respectively and use the metadata information from it. They produce the raw metadata, which can be saved in a xml file and be edited later.

The ASP.NET MVC DataService controler has the code action

```
ActionName("code")]
[HttpGet]
public ActionResult GetCode(string lang)
{
    if (lang != null)
    {
        switch (lang.ToLowerInvariant())
        {
            case "ts":
            case "typescript":
                return GetTypeScript();
            case "xaml":
                return GetXAML();
            case "csharp":
                return GetCSharp();
            default:
                throw new Exception(string.Format("Unknown lang argument: {0}", lang));
        }
    }
    return GetTypeScript();
}
```

You can generate the raw (*draft*) XML representation of the metadata (*it is usually the first step of implementing the DataService*).

Note: *using the metadata the DataService's ServiceCodeGen method generates entities and strongly typed DbSets classes.*

For that you can navigate to the *CodeGen* url in the browser like in the example
http://YOURSERVER/RIAppDemoServiceEF/code?lang=xaml
Then you can copy and paste the XML (*save into an XML file*).

The DataService usually first implements GetMetadata, Bootstrap, and ConfigureCodeGen methods:

```
protected override Metadata GetMetadata(bool isDraft)
{
        if (isDraft)
        {
           // returns the  raw (uncorrected) programmatically generated metadata from LinqToSQL classes
           // usually at first, one gets the raw metadata, then saves it in the file and edits it
           return base.GetMetadata(true);
        }
        // returns the metadata stored in the xml file
        return Metadata.FromXML(ResourceHelper.GetResourceString("RIAppDemo.BLL.Metadata.MainDemo2.xml"));
}

 protected override void Bootstrap(ServiceConfig config)
 {
        base.Bootstrap(config);
        ValidatorConfig.RegisterValidators(config.ValidatorsContainer);
        DataManagerConfig.RegisterDataManagers(config.DataManagerContainer);
 }

 protected override void ConfigureCodeGen(CodeGenConfig config)
 {
        base.ConfigureCodeGen(config);
        config.clientTypes.AddRange(new[] { typeof(TestModel), typeof(KeyVal), typeof(StrKeyVal), typeof(RadioVal),
typeof(HistoryItem), typeof(TestEnum2) });
        //it allows getting information via GetCSharp, GetXAML, GetTypeScript
        //it should be set to false in release version
        //allow it only at development time
        config.IsCodeGenEnabled = true;
}
```

The *Bootstrap* method is used to register the validators and the data managers. The data manages implement CRUD methods for the entities.The validators implement the validation logic. You can not use them and implement all that logic in the data service, but they help to separate the logic into easily testable and reusable classes.

an example of the validator:

```
 public class CustomerValidator : BaseValidator<Customer>
 {
     public override Task<IEnumerable<ValidationErrorInfo>> ValidateModelAsync(Customer customer,
        string[] modifiedField)
     {
        var errors = new LinkedList<ValidationErrorInfo>();
        if (Array.IndexOf(modifiedField, "ComplexProp.ComplexProp.Phone") > -1 &&
           customer.ComplexProp.ComplexProp.Phone != null &&
           customer.ComplexProp.ComplexProp.Phone.StartsWith("000", StringComparison.OrdinalIgnoreCase))
           errors.AddLast(new ValidationErrorInfo
           {
              fieldName = "ComplexProp.ComplexProp.Phone",
              message = "Phone number must not start with 000!"
           });
        if (ChangeType == ChangeType.Updated && Original.ModifiedDate >= customer.ModifiedDate)
```

```
            errors.AddLast(new ValidationErrorInfo
            {
               fieldName = "ModifiedDate",
               message = "ModifiedDate must be greater than the previous ModifiedDate"
            });

         return Task.FromResult(errors.AsEnumerable());
      }
}
```

an example of the data manager:

```
public class CustomerAddressDM : AdWDataManager<CustomerAddress>
{
      [Query]
      public QueryResult<CustomerAddress> ReadCustomerAddress()
      {
         int? totalCount = null;
         var res = PerformQuery(ref totalCount);
         return new QueryResult<CustomerAddress>(res, totalCount);
      }

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

      [Authorize(Roles = new[] {ADMINS_ROLE})]
      public override void Insert(CustomerAddress customeraddress)
      {
         customeraddress.ModifiedDate = DateTime.Now;
         customeraddress.rowguid = Guid.NewGuid();
         DB.CustomerAddresses.Add(customeraddress);
      }

      [Authorize(Roles = new[] {ADMINS_ROLE})]
      public override void Update(CustomerAddress customeraddress)
      {
         var orig = GetOriginal();
         DB.CustomerAddresses.Attach(customeraddress);
         DB.Entry(customeraddress).OriginalValues.SetValues(orig);
      }

      [Authorize(Roles = new[] {ADMINS_ROLE})]
      public override void Delete(CustomerAddress customeraddress)
      {
         DB.CustomerAddresses.Attach(customeraddress);
         DB.CustomerAddresses.Remove(customeraddress);
      }
}
```

The *ConfigureCodeGen* method is used to configure for which types to generate the client side code. It is also used to register the code generation providers.

To return the DbSet data to the client the DataService must implement the query methods - they can be implemented in the data managers or directly in the data service. The Query methods return a *QueryResult* instance.

an example of the query method:

```
[Query]
public QueryResult<Address> ReadAddress()
{
        int? totalCount = null;
        var res = this.PerformQuery(DB.Addresses.AsNoTracking(), ref totalCount).AsEnumerable();
        return new QueryResult<Address>(res, totalCount);
}
```

The Query attribute is used to mark the method as the query. In case if the query methods returns an object, you can provide the name to which DbSet the query method belongs.

```
[Query(DbSetName = "ProductCategory", EntityType = typeof(ProductCategory))]
public QueryResult<object> ReadProductCategory()
{
        int? totalCount = null;
        // we return anonymous type from query instead of real entities
        // the framework does not care about the real type of the returned entities as long as they contain all the fields
        var res = this.PerformQuery(DB.ProductCategories.AsNoTracking(), ref totalCount).Select(p =>
        new
        {
           ProductCategoryID = p.ProductCategoryID,
           ParentProductCategoryID = p.ParentProductCategoryID,
           Name = p.Name,
           rowguid = p.rowguid,
           ModifiedDate = p.ModifiedDate
        });
        return new QueryResult<object>(res, totalCount);
}
```

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

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

In addition to the query methods, there are also CRUD methods (*they are optional, if you don't need to allow editing*) to perform inserts, deletes and updates on the entities. You need to add the [Insert],[Update],[Delete] attributes to mark them for the data service to know about them. You can see the demo how they are used.

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, *custom validation* methods and the *service* methods.

Refresh methods

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

```
 [Refresh]
 public Task<Product> RefreshProduct(RefreshInfo refreshInfo)
 {
   return Task.Run(() => { return DataService.GetRefreshedEntity(DB.Products,
refreshInfo); });
 }
```

Custom validation methods

They are used to validate an entity when the custom validation is needed. They can be used instead of the specialized validators.

an example of a server side custom validation method:

```
[Validate]
public IEnumerable<ValidationErrorInfo> ValidateAddress(Address address, string[] modifiedField)
{
      return Enumerable.Empty<ValidationErrorInfo>();
}
```

Note: *the modifiedField parameter allows you to validate only the modified fields.*

Service methods

They are executed from the client, to invoke the code on the server and optionally to return the result.
These methods are distinguished from the others by annotating them with an *Invoke* attribute. They can also have an *Authorize* attribute.

```
 [AllowAnonymous]
 [Invoke]
 public Task<string> TestInvoke(byte[] param1, string param2)
 {
         var ipAddressService = this.ServiceContainer.GetService<IHostAddrService>();
         string userIPaddress = ipAddressService.GetIPAddress();

         return Task.Run(() =>
         {
            var 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} User IP: {2}",
                  sb, param2, userIPaddress);
         });
 }

 [Invoke]
```

```
public void TestComplexInvoke(AddressInfo info, KeyVal[] keys)
{
        var ipAddressService = this.ServiceContainer.GetService<IHostAddrService>();
        string userIPaddress = ipAddressService.GetIPAddress();
        //p.s. do something with info and keys
}
```

The Metadata

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

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

range = "";
regex = "";
dependentOn = "";
```

The meaning of the attributes is clear from their names.
But several attributes need more explanations:

isPrimaryKey - is an integer typed attribute. So if you have two fields which are in a composite primary key, then for the first field it is set isPrimaryKey=1 and for the second isPrimaryKey=2. Each entity must have a primary key to uniquely identify the entity. Primary key fields are not editable (*readonly*). The values for the primary key can be also generated on the client side. In that case you need to allow the field assignment on the client (*for new entities only*) that is done by setting *allowClientDefault=true*.

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

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

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

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

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

isAutoGenerated - prevents the field to accept updates from the client (*it ignores the updates for them.*).

range - the attribute is used to set accepted range of values for automatic validation. For example, range="100,5000" or for dates, range="2000-01-01,2015-01-01"
regex - the attribute is used to set regular expression for automatic validation.
For example, regex="^[_a-z0-9-]+(\.[_a-z0-9-]+)*@[a-z0-9-]+(\.[a-z0-9-]+)*(\.[a-z]{2,4})$"

fieldType - can have a value from the *FieldType* enumeration:

```
public enum FieldType
{
    None = 0,
    ClientOnly = 1,
    Calculated = 2,
    Navigation = 3,
    RowTimeStamp = 4,
    Object = 5,
    ServerCalculated = 6
}
```

Associations

The association defines foreign key references and navigation fields names

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

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

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

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

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

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

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

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

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

```
[Query]
public async Task<QueryResult<Product>> ReadProduct(int[] param1, string param2)
{
        int? totalCount = null;
        var qinf = RequestContext.CurrentQueryInfo;
        //the async delay is for the demo purposes to make it more  complex
        //await Task.Delay(5000).ConfigureAwait(false);

        //another async part
        var productsArr = await PerformQuery(ref totalCount).ToArrayAsync();
        var productIDs = productsArr.Select(p => p.ProductID).Distinct().ToArray();
        var queryResult = new QueryResult<Product>(productsArr, totalCount);

        var subResult = new SubResult
        {
           dbSetName = "SalesOrderDetail",
           Result = DB.SalesOrderDetails.AsNoTracking().Where(sod => productIDs.Contains(sod.ProductID))
        };

        // include the related SalesOrderDetails with the products in the same query result
        queryResult.subResults.Add(subResult);
        // example of returning out of band information
        queryResult.extraInfo = new {test = "ReadProduct Extra Info: " + DateTime.Now.ToString("dd.MM.yyyy
HH:mm:ss")};
        return queryResult;
}
```

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

For example, you can at first to load 20 data pages of the Customer entities, and then you can load CustomerAddress entities for the current page of the loaded customers.

You can see the example of loading related entities in this way in the ManyToMany Demo.

examples of two server side query mehods which accept parameters:

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

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

Authorization

The authorization can be applied on several levels - the data service class level, the data manager level and a method level.
To make it work you need to annotate a data service class or a method with an *Authorize* attribute. The Authorize attribute can include user roles. Without including the roles it is simply checks that the user is authenticated.
The Authorize attribute is optional, and when it is not applied then it is assumed that the access is allowed on that level.

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

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

The *OverrideAuthorizeAttribute* can be used to override the authorization on the higher level with the new rules.

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

```
public interface IAuthorizer
{
      IPrincipal principal { get; }
      Type serviceType { get; }
      void CheckUserRightsToExecute(IEnumerable<MethodInfoData> methods);
      void CheckUserRightsToExecute(MethodInfoData method);
}
```

The BaseDomainService implements the ConfigureServices method which can be overriden to register new or replace existing services, including the Authorizer.

```
 protected virtual void ConfigureServices(IServiceCollection services)
 {
        services.AddSingleton<ISerializer>(this._Serializer);
```

```
            services.AddSingleton<IPrincipal>(this._User);
            services.AddSingleton<IDomainService>(this);
            services.AddSingleton<BaseDomainService>(this);
            services.AddSingleton(this.GetType(), this);

            services.AddSingleton<IAuthorizer, AuthorizerClass>();

            services.AddSingleton<IValueConverter, ValueConverter>();

            services.AddSingleton<IDataHelper, DataHelper>();

            services.AddSingleton<IValidationHelper, ValidationHelper>();

            services.AddSingleton<IServiceOperationsHelper, ServiceOperationsHelper>();

            foreach (var descriptor in this.Config.DataManagerContainer.Descriptors)
            {
                services.AddSingleton(descriptor.ServiceType, descriptor.ImplementationType);
            }
            foreach (var descriptor in this.Config.ValidatorsContainer.Descriptors)
            {
                services.AddSingleton(descriptor.ServiceType, descriptor.ImplementationType);
            }
    }
```

Change Tracking (auditing)

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

```
protected virtual void OnTrackChange(string dbSetName, ChangeType changeType, string diffgram)
{
}
```

an example of a diffgram for the Product entity:

```
<diffgram dbset-name="Product" key-name="ProductID" key-val="712" change-type="Updated">
  <Name old="AWC Logo Cap" new="AWC Logo Cap2" />
  <ProductNumber old="CA-1098" new="CA-10988" />
  <Color old="Multi" new="Multi2" />
  <StandardCost old="6.9223" new="6.93" />
  <ProductCategoryID old="24" new="14" />
  <ProductModelID old="61" new="97" />
  <SellStartDate old="07/01/2001 00:00:00" new="07/02/2001 00:00:00" />
  <ModifiedDate old="01/31/2018 13:39:35" new="03/30/2018 15:11:29" />
</diffgram>
```

Error logging in the data service

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

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

Disposal of resources used by the data service (cleanup)

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

an example of an overridden Dispose method:

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

        base.Dispose(isDisposing);
}
```

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

The BaseDomainService has the *ServiceCodeGen* method. It invokes the *ConfigureCogeGen* method.

```
public string ServiceCodeGen(CodeGenArgs args)
{
        CodeGenConfig config = new CodeGenConfig(this);
        try
        {
            this.ConfigureCodeGen(config);
        }
        catch (Exception ex)
        {
            this._OnError(ex);
            throw;
        }

        try
        {
            if (!config.IsCodeGenEnabled)
                throw new InvalidOperationException(ErrorStrings.ERR_CODEGEN_DISABLED);
            ICodeGenProvider codeGen = config.GetCodeGen(args.lang);
            return codeGen.GetScript(args.comment, args.isDraft);
        }
        catch(Exception ex)
        {
            this._OnError(ex);
            throw;
        }
}
```

The ConfigureCodeGen is usually overriden in the derived classes. It can be used to add code generation providers. They are pluggable.

```
protected override void ConfigureCodeGen(CodeGenConfig config)
{
```

```
        base.ConfigureCodeGen(config);
        config.AddOrReplaceCodeGen("csharp", () => new CsharpProvider<TDB>(this));
}
```

The ConfigureCodeGen is also used to add client types (*the types for which the client code needs to be generated*)

```
 protected override void ConfigureCodeGen(CodeGenConfig config)
 {
        base.ConfigureCodeGen(config);
        config.clientTypes.AddRange(new[] { typeof(TestModel), typeof(KeyVal), typeof(StrKeyVal),
typeof(RadioVal), typeof(HistoryItem), typeof(TestEnum2) });
        //it allows getting information via GetCSharp, GetXAML, GetTypeScript
        //it should be set to false in release version
        //allow it only at development time
        config.IsCodeGenEnabled = true;
}
```

Some types have attributes placed on them, which affect the code generation. For example, the RadioVal has the attribute:

```
[Dictionary(KeyName = "key", DictionaryName = "RadioValDictionary")]
```

The HistoryItem is more heavily annotated:

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

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

The code generation also produces typescript *enums* from the C# *enums*.

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

The code generation workflow (from the user activity point of view)

usually starts from generation the draft xml metadata.

http://YOURSERVER/RIAppDemoServiceEF/code?lang=xaml

That's because,at the time of generating it the data service can be with no implemented the CRUD and the Invoke methods. The obtained raw xml is stored in a file and added as the text resource to the data service.
The xml file can be edited - some entities can be removed from it, some fields can be made readonly, the primary keys can be marked.

After doing this you can generate the C# methods - the CRUD methods. For this it needs the real metadata (*usually in the form of an xml file*) to be present.

http://YOURSERVER/RIAppDemoServiceEF/code?lang=charp

The generated C# code can be copied and pasted into the Data service. Then you can add security attributes on the methods, add the service (invoke) methods. You can also refactor the CRUD methods into its own data managers. As a rule this generated C# code can be used as a template, so not to write it from the scratch, but it needs some changes to it.

After implementing all the methods in the data service, you can obtain the client side code.

http://YOURSERVER/RIAppDemoServiceEF/code?lang=ts

Then you can save this generated typescript code into a module and then to start coding for the client side.