

Table of Contents

- Writing client-side plugins** 2
- Architecture of the client-side code*** 2
- Modifying default behavior and presentation of queries*** 3
- Modifying default behavior and presentation of forms*** 5
- Modifying default behavior and presentation of form sections*** 7
- Modifying default presentation of individual fields on forms*** 7
- Modifying default behavior of menu in visual perspectives*** 10
- Modifying default behavior and presentation of content panels in visual perspectives***
 10
- AwareApp object*** 11
- Using Javascript to integrate custom Cordova plugins for native mobile applications***
 12
- Creating a contact on the phone 13
- Send email to the selected contact 14

[Programmers Reference](#), [Client Side Plugins](#)

Writing client-side plugins

In **Aware IM** you can not only add plugins for the server (such as custom processes, channels or functions), but you can also add plugins that execute on the client within a web browser. Most of the time you would write these plugins in order to add your custom user interface functionality, or modify the default **Aware IM** user interface behaviour.

All client-side plugins must be written in Javascript and in most cases you need the knowledge of the [Kendo UI](#) Javascript library from Telerik and a popular open source Javascript library called [jQuery](#) . The description that follows assumes that the reader is familiar with Javascript, Kendo UI library and jQuery.

There are several types of the client-side plugins you can add in **Aware IM**:

1. Modify the default behaviour and presentation of forms
2. Modify the default behaviour and presentation of form sections in forms
3. Modify the default presentation of individual fields within forms
4. Modify the default presentation and behaviour of queries
5. Modify the default presentation and behaviour of content panels inside visual perspectives

We will look at each of these client-side plugins separately

Architecture of the client-side code

Before we explain how to write scripts for different components it is useful if a developer understands roughly the general architecture of the client-side code.

This is what happens behind the scenes when a screen is displayed in the browser by **Aware IM**. A screen (usually represented by visual perspectives) consists of multiple queries, forms, content panels and so on (we will call them "components").

Each component, such as a form or a query is handled by the appropriate "controller" (also called "parser"). The first thing the controller does is ask the server to provide the definition of the layout of the component. Then the controller parses XML returned by the server and prepares two Javascript objects:

1. jQuery object containing HTML of the component (form or query). This object is called "markup".
2. An array of "widget configurations". Each member of the array represents configuration for some widget of the Kendo UI library that the component includes. For example a query usually just includes a single widget that implements a query, such as the grid widget, for example. But a form may have a number of widgets - almost one per each attribute displayed by the form

Then the screen to be displayed is assembled from HTML markups of different components that the screen contains and the final HTML for the screen is built. This HTML is then given to the Kendo UI library, which creates all widgets of the screen based on the screen HTML and configurations of the

widgets prepared by the controllers. Kendo UI library modifies this HTML to add its own classes and performs other steps to ensure that its widgets are displayed correctly. Finally the resulting HTML document is given to the browser which draws it on the screen.

To summarize:

1. A screen consists of components. Each component is represented by its own controller
2. The process starts by each controller asking the server for the definition of components
3. The controller then prepares HTML markup of the component and configurations of containing widgets
4. The HTML of the screen is created from the markups of components returned by controllers
5. The HTML and widget configurations are then given to the Kendo UI library to prepare its widgets
6. Final HTML of the screen is produced and is drawn by the browser.

So where in this process do the client scripts come in? For most components there are two types of scripts – the “initialization” script and the “render” script. As a developer you can define one or both – depending on what the script needs to do. The initialization script if defined is run by **Aware IM** just before the controller of the component returns the markup of the controller and the array of widget configurations (so immediately after step 3 above). The script, therefore, has a chance to modify the markup returned by the controller or the configuration of any of the widgets of the component.

The markup can be modified using jQuery functions that manipulate HTML. The script can only modify the markup for the component, but not the entire screen, because the entire screen hasn’t been built yet.

Widget configurations represent Javascript objects with properties described by Kendo UI API Reference. For example, to see the API Reference of the Kendo UI grid widget that implements **Aware IM** queries in the standard form go to <http://docs.telerik.com/kendo-ui/api/javascript/ui/grid> and look up the Configuration section at the top. Note that you can only modify configuration of the widget (which also includes Events), but you cannot use Methods of the widget in the initialization script.

The render script, though, runs after everything has been drawn on the screen – i.e. after step 6 above. By this time all Kendo UI widgets will have been already created, so the script can access the widget and call its methods (see the Methods section in the Kendo UI API Reference for each widget). Configuration objects cannot be used at this stage.

The render script can also access the final browser document and manipulate it if need be using jQuery functions. The following sections describe how this can be done in more detail.

Modifying default behavior and presentation of queries

To modify the default behavior and presentation of queries you need to go to a particular query that you want to modify and click on the “Scripts” property in the list of properties of the query. You can define “initialization” script or “render” script or both (see “Architecture of the Client-side Code”)

The following objects are exposed to the initialization script:

1. “config” object - this object represents Kendo UI configuration of the main widget implementing the query (see the table below)

2. "markup" object - HTML markup prepared by the controller
3. "parser" object - the controller itself
4. "widgets" - an array of all widget configurations for the query

|Query presentation type|Kendo UI widget|Kendo UI reference|

| | | |
|--|------------------|---|
| Standard | Grid | http://docs.telerik.com/kendo-ui/api/javascript/ui/grid |
| Custom (Custom Data Template, scroll view unticked) | List View | http://docs.telerik.com/kendo-ui/api/javascript/ui/listview |
| Custom (Custom Data Template, scroll view ticked) | Scroll View | http://docs.telerik.com/kendo-ui/api/javascript/mobile/ui/scrollview |
| Custom (Mobile Data Template) | Mobile List View | http://docs.telerik.com/kendo-ui/api/javascript/mobile/ui/listview |
| Calendar/Scheduler | Scheduler | http://docs.telerik.com/kendo-ui/api/javascript/ui/scheduler |
| Chart | Chart | http://docs.telerik.com/kendo-ui/api/javascript/dataviz/ui/chart |
| Gantt | Gantt | http://docs.telerik.com/kendo-ui/api/javascript/ui/gantt |

It is usually unnecessary to modify the markup, but you are welcome to modify the configuration of the widget.

For example, the following script will change the alignment of the column that corresponds to the "Status" attribute:

```
for (var i = 0; i < config.columns.length; ++ i)
{
if (config.columns[i].field == "Status")
{
config.columns[i].attributes = { alignment: "right" }
}
}
```

The following script will make the grid "groupable", i.e allow dragging the columns to a special area in order to group the grid by this column:

```
config.groupable = true;
```

The "parser" object represents the controller and allows you to access certain properties of the system that you may need. The type of this object depends on the type of the query representation and is provided in the table below:

| Query presentation type | Aware IM Javascript object type | Source code |
|-------------------------------------|---------------------------------|--|
| Standard | AwareApp_QueryLayoutParser | AwareIM/Tomcat/webapps/AwareIM/aware_ext/parsers/queryLayoutParser.js |
| Custom (without scroll view) | AwareApp_CustomLayoutParser | AwareIM/Tomcat/webapps/AwareIM/aware_ext/parsers/customLayoutParser.js |

| | | |
|--------------------------------------|-------------------------------|--|
| Custom (with scroll view) | AwareApp_ScrollViewParser | AwareIM/Tomcat/webapps/AwareIM/aware_ext/parsers/scrollViewParser.js |
| Custom (with mobile template) | AwareApp_MobileListViewParser | AwareIM/Tomcat/webapps/AwareIM/aware_ext/parsers/customMobileParser.js |
| Chart | AwareApp_ChartParser | AwareIM/Tomcat/webapps/AwareIM/aware_ext/parsers/chartParser.js |
| Calendar/Scheduler | AwareApp_CalendarParser | AwareIM/Tomcat/webapps/AwareIM/aware_ext/parsers/calendarParser.js |
| Gantt | AwareApp_GanttParser | AwareIM/Tomcat/webapps/AwareIM/aware_ext/parsers/ganttParser.js |

The “widgets” object representing an array of all widgets that the query has can be used to modify other widgets - for example, toolbars generated to represent query operations (if these are defined for the query).

Each member of the “widgets” array has the following properties:

1. type (the type of the Kendo UI widget)
2. id (the unique id in the markup of the query that the widget uses)
3. config (the Kendo UI configuration of the widget)

So to modify a toolbar, for example (<http://docs.telerik.com/kendo-ui/api/javascript/ui/toolbar>) and stop it being resizable you would find the toolbar in the array and modify its “config” property like this:

```
for (var i = 0; i < widgets.length; ++ i)
{
  if (widgets[i].type == “toolbar”)
  widgets[i].config.resizable = false;
}
```

The second script (“render” script) that you can define in the Scripts dialog allows you to modify the widget representing the query after it has been drawn. Here you would use methods of the corresponding Kendo UI object, rather than configuration options. Objects available for your Javascript here are:

1. “widget” object
2. “parser” object

The “widget” object represents the widget that has been drawn.

The “parser” object is the controller object described above.

Modifying default behavior and presentation of forms

To modify the default behavior and presentation of forms you need to go to a particular object form that you want to modify and click on the Scripts property under the “Advanced” category in the list of properties of the form. You can define “initialization” script or “render” script or both (see “Architecture of the Client-side Code”).

The Javascript objects that are exposed to the initialization script for forms are the same as for queries:

1. "config" object
2. "markup" object
3. "parser" object
4. "widgets" object

Just like with queries, the parser object represents the form controller and the "markup" object represents the HTML markup of the form. However, there is a significant difference in the "config" object exposed by forms. Whereas for queries the "config" object represents some widget of the Kendo UI library, that is mainly responsible for the implementation of the query, there is no such widget for forms. A form is just an HTML code that consists of rows and columns of the Bootstrap grid system (getbootstrap.com/css/). Each row and column contains attributes of the form – depending on the type of the attribute they are either implemented as plain HTML or they may also include a configuration for a Kendo UI widget.

This HTML is also wrapped in a "panel" that includes HTML of toolbars around the form (if they are defined) and also the implementation of the default or custom panel header. Note that the HTML that includes toolbars and panel header is also included as part of the query markup.

So the "config" object for forms represents the configuration of a special Aware IM object called "panel". You can find the code of this object in `AwareIM/Tomcat/webapps/AwareIM/aware_kendo/panel.js` file. At the beginning of the file there is a description of all configuration parameters supported by this object. You can change these parameters by your script..

For example, the following script will turn off the display of the header for the form, no matter what is specified in the **Aware IM** properties of the form:

```
config.preventHeader = true;
```

However, you are unlikely to need it because most of these parameters can be customized in **Aware IM** without having to use a script.

You may, however, want to modify the markup of the form – for example, you may want to modify the generated layout or styling of some attributes. You can use it through the "markup" object (or in the "render" script after the form has been drawn). The description of the HTML markup is beyond the scope of this document. If you want to study it you can just generate a form and use the browser inspector to display the HTML of the form. You can then use your scripts to modify this markup.

You can, however, modify the widgets used by the form using the "widgets" object – you will probably modify widgets that represent attributes in their own Advanced scripts, but you can also modify other widgets used by a form, such as toolbars, for example – this has been already explained in the previous section.

The "parser" object allows you to access certain properties of the system that you may need (especially if you are modifying the markup of the form). The type of this object is `AwareApp_FormParser`. You can look up full methods of this object in the file `AwareIM/Tomcat/webapps/AwareIM/aware_kendo/parsers/formParser.js`

Some useful methods of the parser object that you can use here are:

1. `getField (attributeName, sectionName)` – get the field of the form for the specified attribute and form section
2. `getReferenceParser (refAttrName)` – get the "parser" of the reference attribute of the form,

responsible for displaying a table, calendar etc – see 8.1

The second script (“render” script) that you can define in the Scripts dialog allows you to modify the form after it has been drawn. Here you can only use jQuery methods to modify the resulting document markup. The only object available for you here is the “parser” object representing the form controller.

Modifying default behavior and presentation of form sections

Modifying the default behavior of a form section only makes sense if a form has more than one section defined. If a form has one section than you should modify the behavior of the form, not section, as described in the previous section. To modify the default behavior and presentation of a form section you need to go to a particular form section of the form and click on the “Scripts” property under the “Advanced” category in the list of properties of the form section.

The rules for defining the scripts are very similar to defining the scripts for forms, the only difference being the “parser” object. The type of this object is `AwareApp_FormSectionParser` rather than `AwareApp_FormParser` and the source code of the object is available in this file:

`AwareIM/Tomcat/webapps/AwareIM/aware_kendo/parsers/formSectionParser.js`

Modifying default presentation of individual fields on forms

To modify the default presentation of an individual field on an object form you need to go to the presentation properties of the corresponding attribute and click on the “Advanced” property. There is only one script available for you here.

As explained in the “Architecture of the client-side code” the controller of the form prepares the HTML markup of the form as well as the list of Kendo UI widgets that the form includes. Apart from other things the markup of the form contains markups of individual fields present on the form. And the collection of widgets for the form includes widgets used by individual fields (note that not all fields use widgets, some use markup only). Each type of field on the form is represented by its own **Aware IM** Javascript object (see the table below).

The form controller asks every individual field on the form to prepare its markup and the collection of widgets. Then it assembles the result into the final markup and widget collection of the form. The script for each individual field is executed just before it is given to the form controller, so that the script has a chance to modify the markup or widget configuration.

There are three objects exposed to the script:

1. “field” – this is Aware IM object representing the field (see the table below)
2. “markup” – this is the HTML markup of the field (jQuery object)
3. “config” – this is the object that represents a widget configuration of the field or null if the field does not use a widget. The object has the following properties:
 1. “type” – type of the widget
 2. “id” – the id of the element in the markup used by the widget

3. "config" - the Kendo UI configuration of the widget

You can modify any of these objects. For example, if you want to hide the field you can write the following script:

```
markup.css ("display", "none");
```

Or if you want to change which tools are available for an HTML editor field (see <http://docs.telerik.com/kendo-ui/api/javascript/ui/editor#configuration-tools>), you could write the following script:

```
config.config.tools = ["bold", "italic", "underline"]
```

There are some useful methods of the "field" object that you can use in your script (the code of all the objects representing different fields is in `AwareIM/Tomcat/webapps/AwareIM/aware_kendo/field/fields.js` file).

1. `field.getAttributeName ()` - retrieve the name of the object attribute
2. `field.getObjectname ()` - retrieve the name of the object
3. `field.getObjectId ()` - retrieve the id of the object

and so on.

Note that if you want to access the field after it has already been drawn you need to find the field on the form and so you need to modify the "render" script of the form, like so, for example:

```
var field = parser.getField ("Account", "Main");
```

```
var value = field.getValue ();
```

The following table lists all different field types and the corresponding Kendo UI widgets.

| Aware IM attribute type | Kendo UI widget | Aware IM field object |
|--|--|--------------------------------------|
| Plain Text (no choices, 1 line) | None | AwareApp_TextField |
| Plain Text (no choices, several lines) | None | AwareApp_TextAreaField |
| Plain Text, Number, Date with radio or checkbox choices | NumericText Box (http://docs.telerik.com/kendo-ui/api/javascript/ui/numerictextbox) | AwareApp_NumberField |
| Plain Text, Number, Date with text choices, choices not editable | DropDownList (http://docs.telerik.com/kendo-ui/api/javascript/ui/dropdownlist) | AwareApp_ComboField |

| | | |
|--|--|--------------------------------|
| Plain Text, Number, Date with text choices, choices editable | ComboBox (http://docs.telerik.com/kendo-ui/api/javascript/ui/combobox) | AwareApp_ComboField |
| Date without choices | DatePicker (http://docs.telerik.com/kendo-ui/api/javascript/ui/datepicker) | AwareApp_DateField |
| Timestamp | DateTimePicker http://docs.telerik.com/kendo-ui/api/javascript/ui/datetimestpicker | AwareApp_DateTimeField |
| Yes/No (displayed as checkbox) | None | AwareApp_CheckboxField |
| Yes/No (displayed as a switch) | Switch http://docs.telerik.com/kendo-ui/api/javascript/mobile/ui/switch | AwareApp_SwitchField |
| Plain Text with choices represented as checkboxes | None | AwareApp_CheckboxGroupField |
| Plain Text with choices represented as radio buttons | None | AwareApp_RadioButtonGroupField |
| PlainText with multi-selector | MultiSelect http://docs.telerik.com/kendo-ui/api/javascript/ui/multiselect | AwareApp_TagField |
| PlainText with HTML format | Editor http://docs.telerik.com/kendo-ui/api/javascript/ui/editor | AwareApp_HtmlEditorField |
| Document | Upload http://docs.telerik.com/kendo-ui/api/javascript/ui/upload | AwareApp_DocumentField |
| Picture (not represented as signature) | Upload http://docs.telerik.com/kendo-ui/api/javascript/ui/upload | AwareApp_PictureField |
| Picture (represented as signature) | None | AwareApp_SignatureField |
| Shortcut | None | AwareApp_ShortcutField |
| Reference represented by a drop down | DropDownList (http://docs.telerik.com/kendo-ui/api/javascript/ui/dropdownlist) | AwareApp_SelectReferenceField |
| Reference represented by a multi-selector | MultiSelect http://docs.telerik.com/kendo-ui/api/javascript/ui/multiselect | AwareApp_TagReferenceField |
| Reference represented by a "swap select" | None | AwareApp_SwapSelectField |
| HTML cell | None | AwareApp_HtmlFiield |
| Gauge cell (linear) | LinearGauge http://docs.telerik.com/kendo-ui/api/javascript/dataviz/ui/lineargauge | AwareApp_GaugeField |
| Gauge cell (radial) | RadialGauge http://docs.telerik.com/kendo-ui/api/javascript/dataviz/ui/radialgauge | AwareApp_GaugeField |
| Google Map cell | None | AwareApp_GoogleMapField |

| | | |
|----------------------------|---|----------------------|
| Number displayed as slider | Slider http://docs.telerik.com/kendo-ui/api/javascript/ui/slider | AwareApp_SliderField |
|----------------------------|---|----------------------|

Modifying default behavior of menu in visual perspectives

The idea here is very similar. You have two scripts available - initialization and render scripts. The initialization script has a chance to modify the configuration of the menu widgets (almost all menu types except Plain List are implemented by their own Kendo UI widget (see the table below). The render script can call the methods of the widget once it has been drawn.

The following objects are exposed to the initialization script:

1. "config" - this object represents Kendo UI configuration of the menu widget
2. "parser" - the controller object (AwareApp_VPParser) - see the code in the file

AwareIM/Tomcat/webapps/AwareIM/aware_kendo/parsers/vpParser.js

For example to add some custom menu item to a toolbar menu you could write the following script:

```
config.items.push ({
type: "button",
spriteCssClass: "fa fa-edit",
text: "My Menu Item",
click: function () {
alert ("this is my menu item");
}
});
```

| Menu type | Kendo UI widget | Kendo UI reference |
|---------------|-----------------|---|
| Toolbar | ToolBar | http://docs.telerik.com/kendo-ui/api/javascript/ui/toolbar |
| Standard Menu | Menu | http://docs.telerik.com/kendo-ui/api/javascript/ui/menu |
| Panel Bar | PanelBar | http://docs.telerik.com/kendo-ui/api/javascript/ui/panelbar |
| Tree | TreeView | http://docs.telerik.com/kendo-ui/api/javascript/ui/treeview |

Modifying default behavior and presentation of content panels in visual perspectives

To modify the default behavior and presentation of content panels in visual perspectives you need to go to a particular visual perspective that you want to modify, select the content panel and then click on the "Scripts" property in the list of properties of the content panel.

The idea here is the same - you have two scripts as before. However, there are no Kendo UI widgets

to modify here – you can only modify the markup of the content panel (either during initialization or after it has been drawn).

Two objects are exposed for the initialization script:

1. “config” – this is the configuration of the Aware IM “panel” object, the code of which is in `AwareIM/Tomcat/webapps/AwareIM/aware_kendo/panel.js`

The markup of the panel is stored in the “bodyContent” property of the object `config.bodyContent`. This is the markup you are most likely to modify here

“parser” object – this is the controller (`AwareApp_Dashboard` in `AwareIM/Tomcat/webapps/AwareIM/aware_kendo/parsers/dashboard.js`)

AwareApp object

When writing advanced scripts as described above you can use the `AwareApp` Javascript object that contains some useful static methods. This is an example of calling one of these methods:

```
var panelId = AwareApp.getPanelId (“main”, “Accounts”, “My Accounts”);
```

The code of the object is located here: `AwareIM/Tomcat/webapps/AwareIM/aware_ext/awareApp.js`

The following methods can be used:

1. `getPanelId (frameName, tabName, contentPanelName)`

This method returns the unique id of the content panel in a visual perspective.

```
var id = AwareApp.getPanelId (‘main’, ‘Tab1’, ‘Content Panel1’);
```

Parameters:

`frameName` – name of the frame in the visual perspective that contains the panel

`tabName` – name of the tab inside the frame that contains the panel

`contentPanelName` – name of the content panel

1. `getFramePanelId (frameName)`

This method returns the unique id of the frame in a visual perspective.

```
var id = AwareApp.getFramePanelId (‘main’);
```

Parameters:

`frameName` – name of the frame in the visual perspective

1. `getTabPanelId (frameName, tabName)`

This method returns the unique id of the tab in a visual perspective.

```
var id = AwareApp.getTabPanelId ('main', 'Tab1');
```

Parameters:

frameName - name of the frame in the visual perspective that contains the panel

tabName - name of the tab inside the frame that contains the panel

1. isRTL ()

Return true if the current user uses right-to-left layout

1. getMainTabPanel ()

If a visual perspective has tabs return the tab panel holding the tabs.

1. getProcessId ()

Return the id of the currently running process or -1 if there are no processes currently running.

1. isTestingMode ()

Return true if the current user is running in the testing mode

1. startProcess, runQuery and other methods mentioned in the User Guide

Please refer to the “Links to Aware IM operations” section in the User Guide, that explains how to set up links to perform operations. All functions mentioned there can be used from your Javascript.

Using Javascript to integrate custom Cordova plugins for native mobile applications

Cordova plugins are components that provide access to some built-in features of mobile phones, (such as camera or contacts), for which there is no Javascript access. When components are integrated into the system these features become available through some special Javascript functions that the plugin makes available to the developer. Cordova plugins can only be used in native mobile applications.

Aware IM integrates some Cordova plugins out-of-the-box and provides rule actions that activate them (for example, MOBILE PUSH or MOBILE CAMERA SNAP INTO). However, there are many plugins around and it is impossible to integrate all of them into Aware IM.

Still there is a way to do this by adding some custom Javascript to your application. The following section explains how to do it.

This is the high level overview of what needs to be done:

1. Study the documentation of the Cordova plugin to fully understand Javascript methods that it exposes
2. Write the Javascript that calls the appropriate Javascript function that the plugin provides
 1. Give this function the data obtained from Aware IM if necessary. For example, read the

data from the database and provide this data to the function of the plugin. The useful Aware IM function that can be used here is `AwareApp.getObjectData()`

2. Handle the return of this function if necessary – for example write the data returned by the function to the database. Useful Aware IM functions for this are `AwareApp.createOrUpdateObject()` and `AwareApp.startProcessWithInit()`
3. Define panel operations or menu items in the mobile part of your business space version (using the Configuration Tool) that would run this Javascript. You should select an operation or menu item of the “Execute Javascript” type for this.
4. Build a native mobile application for your business space version using the “Build Native Mobile Application” command in the Configuration Tool. This will create a zip file.
5. Uzip this zip file somewhere. Find the `config.xml` file in the root of the unzipped application and open it for editing.
6. Find the section in this file that lists the plugins used by the application, for example:

```
<plugin name="cordova-plugin-camera" spec="2.0.0" />
```

Add the definition of the Cordova plugin you need to integrate – look up the documentation of the plugin for details of the plugin name and version number

1. Zip up the application again and use the PhoneGap build to create application files in the native format of the mobile phone

Let’s look at an example. We will be integrating a Cordova plugin for Contacts into the CRM mobile sample application.

The documentation of the plugin can be found here:

<https://github.com/apache/cordova-plugin-contacts>

As we can see the plugin provides the `navigator.contacts` object that can be used to create contacts, find existing contact or pick a particular one. Let’s add the following functionality to the CRM application:

1. From the form of a customer or from a customer list create a phone contact populated with the information from the customer record in the application
2. Pick a contact from the list of phone contacts and send this contact an email that includes some information stored in the application

Creating a contact on the phone

We need to use the “create” method of the `navigator.contacts` object and provide contact data available in the customer record that we are parked on. Retrieving the data can be done using the `AwareApp.getObjectData` function. It has the following signature:

```
getObjectData: function (objectName, objectId, callBackFunction)
```

`ObjectName` and `objectId` identify the record to retrieve and `callBackFunction` specifies a function that will be called when the data has been retrieved. The function will be called with the object storing the retrieved values.

How do we get object name and id? When we define an Aware IM operation of the “Execute Script” type Aware IM automatically defines the following objects that we can use in our Javascript:

1. parser
2. context

The parser object should be already familiar and the context object stores an array of objects with `objectName` and `objectId` attributes. The record we are parked on is the first and only one in this array. So to get `objectName` we use the following `context[0].objectName`; and to get object id we use `context[0].objectId`

So the Javascript we need to write to create a contact looks like this:

```
AwareApp.getObjectData (  
  
context[0].objectName,  
  
context[0].objectId,  
  
function (objectData)  
  
{  
  
navigator.contacts.create ({  
  
"displayName": objectData["FirstName"] + " " + objectData["LastName"],  
  
"birthday": kendo.parseDate (objectData["DateOfBirth"], "dd/MM/yyyy", "en-US")  
  
});  
  
}  
  
);
```

Note that here `"displayName"` and `"birthday"` are the names of the attribute of the Contact object on the phone exposed by the plugin, whereas `"FirstName"`, `"LastName"` and `"DateOfBirth"` are the names of the attributes of the Customer object in the CRM application. Note also that all Aware IM attribute values are strings and if the plugin requires some other type (for example, date), then the strings need to be converted to the appropriate type.

The next step is to create operations of the "Execute Script" type to the form and customer list. We can add a panel operation to the "Editing Mobile" form of the Customer object and an operation with record to the "Customer - all mobile" query. We then specify the above script as a parameter of the operation.

Send email to the selected contact

We need to use the `pickContact` method of the `navigator.contacts` object to display a list of contacts, let the user pick one and then we need to start a process in the application to send an email to the email address of the contact picked by the user.

The email address returned by the plugin needs to be saved in some temporary object and then this object can be used in the process. So we will create a temporary business object (persistence type - memory) called `ContactParam` with the single `EmailAddress` attribute. We will then create a process

called `SendEmailToContact` with the `ContactParam` object as its input. The process will then use the `SEND` action to send any email to this email address (the email can use tag expressions to retrieve some information from the system – for example, from `SystemSettings` or from the logged in user).

To start a process we will use the `AwareApp.startProcessWithInit` function. It has the following signature:

```
startProcessWithInit: function (procName, renderOption, objName, initValues, context)
```

Here `procName` is the name of the process to start, `renderOption` is where to display the results of the process (we can use `null`), `objName` is the name of the parameter object, `initValues` is the object storing values of the parameter object and `context` contains additional parameter objects (`null` in our case)

So our Javascript can look like this:

```
navigator.contacts.pickContact (function (contact)
```

```
{
```

```
var email = contact.emals[0].value;
```

```
AwareApp.startProcessWithInit (
```

```
“SendEmailToContact”,
```

```
null,
```

```
“ContactParam”,
```

```
{ “EmailAddress” : email }
```

```
},
```

```
function (error { console.log (error); }
```

```
);
```

Then we just need to add a command of the “Execute Script” type to the mobile menu of the application.

From:

<http://www.awareim.com/dokuwiki/> - **Documentation**

Permanent link:

<http://www.awareim.com/dokuwiki/docs/3500/0800?rev=1680669488>

Last update: **2023/04/05 04:38**

