How to Implement a Server Plug-In

Felix Elliger
fel@inubit.com

Contents

1 General Information and Classification of Plug-Ins **3**
1.1 General Information 3
1.2 Classification of Plug-Ins 4
1.3 Item Offset Specification 4

2 Simple Plug-Ins **6**
2.1 Overview 6
2.2 Details 6

3 Form Plug-Ins **8**
3.1 Overview 8
3.2 Details 8

4 Dialog Plugins **11**
4.1 Overview 11
4.2 Details 11

5 Info-Dialog Plug-Ins **14**
5.1 JavaScript 14

6 Plug-In Responses **16**
6.1 OPEN - Action 16
6.2 UPDATE - Action 16
6.3 INFO - Action 17
6.4 ERROR - Action 17

7 Code Examples **18**

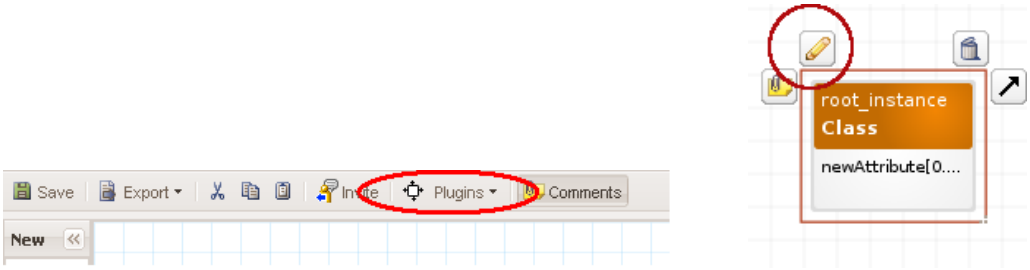
1 General Information and Classification of Plug-Ins

1.1 General Information

Server plug-ins are located within the package `com.inubit.research.server.plugins`. There are two basic alternatives how users can access a plug-in's functionality. If the plug-ins are connected to the model, they are accessible for users via the "Plugins" - Button (see Fig. 1) of the web editor's toolbar or via a toolbar icon. To connect a plug-in to certain model types implement the *ModelScope*-interface. To force a model plug-in to be rendered as a toolbar icon, override the `showInToolbar()`-method within your plug-in class.

If a plug-in is connected to specific object types, there is shown another button within the object's context menu. To connect a plug-in to certain object types implement the *ObjectScope*-interface.

Currently, there have been no tests concerning the implementation of both interfaces for one single plug-in class. From a server-side point of view this should not raise any problem.



(a) Toolbar (b) Context menu

Figure 1: Occurrences of plug-in icons

The implementer of a plug-in does not have to care about the URI that belongs to his/her plug-in. A unique URI will be assigned at server start-up, or plug-in registration. By now, there is no plug-in class that has configurable sub-URIs. This will be implemented later when such a use-case exists.

How Do I Return the Plug-In's Response? Plug-in communication is entirely based on JSON as data interchange format. Therefore, for returning the plug-in's response to the client you should always use the method `ResponseUtils.respondWithJSON`.

For more details on plug-in responses see Section 6.

1.2 Classification of Plug-Ins

Figure 2 shows the general classification of server plug-ins. All of the depicted classes are abstract and have to be sub-classed for implementing a plug-in. As stated above, the interfaces *ModelScope* and *ObjectScope* are used to connect the plug-in to specific model classes or object classes respectively.

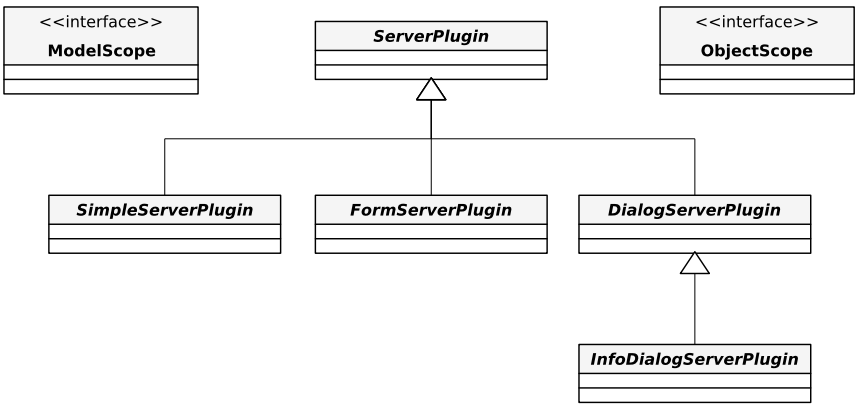


Figure 2: Classification of server plug-ins

Depending on its type the plug-in has a certain set of capabilities. These capabilities and the required implementation effort is presented in the following sections.

To capture the current model and its selection when calling the plug-in, there exists the class *ModelInformation*. This class provides the following methods:

- `ProcessModel getProcessModel()`
  Returns the current process model the plug-in should use.
- `Set<String> getSelectedNodes()`
  Returns the IDs of nodes that are selected within the current model.
- `Set<String> getSelectedEdges()`
  Returns the IDs of edges that are selected within the current model.

As the implementer of a plug-in, you do not have to care about how this information gets to your plug-in. The *ModelInformation* object is created by the abstract classes and will be passed to the respective methods.

1.3 Item Offset Specification

The *ObjectScope*-interface provides a method `getIconOffsetInfo()` returning an instance of the accordingly named class. The information provided by an instance of this class determines the position of the plug-in's button within the object's context menu.

Therefore, it specifies a general horizontal and vertical orientation along with a horizontal and vertical pixel offset.

The orientation is specified by the enum type *Orientation*. It is recommended to use its values `LEFT, RIGHT, CENTER` for horizontal orientation and `TOP, BOTTOM, CENTER` for vertical orientation respectively.

As an example, the delete-button, that is rendered by default when a single object is selected would specify the following offset information:

- horizontal orientation = `TOP`
- horizontal offset = -11
- vertical orientation = `RIGHT`
- vertical offset = -27

These values force the button to be rendered above the top right corner of the frame surrounding the selected object. The offset values have been carefully figured out by simple testing.

2 Simple Plug-Ins

2.1 Overview

Required superclass	*SimpleServerPlugin*
Configurable Sub-URLs	*no*
Requires JavaScript	*no*
Methods to implement	*void performAction(ModelInformation mi , RequestFacade req , ResponseFacade resp , LoginableUser u)*
	String getItemText()
	(optional) *String getItemIconPath()*
Graphical representation	*a single menu item or context button*
Example(s)	*Layout*

2.2 Details

2.2.1 Basic Information

Simple plug-ins are, as their name is already telling, the simplest way of implementing a plug-in. Their graphical representation within the editor's interface is limited to a single menu item within the "Plugins"-menu, or directly within the editor's toolbar. The latter can be achieved by overriding the *showInToolbar()* method.

If the user selects the menu item, a POST-request is generated and sent to the plug-in's unique URL. The POST data consists of the current model's ID along with the IDs of selected nodes and edges. In a first step this JSON data is transformed into a *ModelInformation* object which is then passed to the `performAction` method of your simple plug-in.

2.2.2 Methods

void performAction(ModelInformation mi, RequestFacade req, ResponseFacade resp, LoginableUser u); This method is responsible for processing the incoming request and sending a resulting response to the requester. All required information has to be taken from the *ModelInformation* object.

See Section 6 for information on how responses must be structured.

String getItemText(); Return the text for the menu item. If the *showInToolbar()*-method for this plug-in returns `true` the item text will be used as tool-tip text instead.

String getItemIconPath(); Return the path to the icon of the menu item. This is especially important if the plug-in is represented as a simple button within the toolbar or as a context menu button.

3 Form Plug-Ins

3.1 Overview

Required superclass	*FormServerPlugin*
Configurable Sub-URLs	*no*
Requires JavaScript	*no*
Methods to implement	*void performFormAction(RequestFacade req , ResponseFacade resp , MultiPartObject mpo , ModelInformation mi , LoginableUser u)*
	JSONObject getFormConfig(ModelInformation mi , RequestFacade req , LoginableUser u)
	***(optional)** String getItemText()*
	***(optional)** String getItemIconPath()*
Graphical representation	*a single menu item or context button*
Example(s)	*Behavioral Interface Generation, Deploy to SolutionCenter*

3.2 Details

3.2.1 Basic Information

As we can derive from the superclass name *FormServerPlugin*, this type of plug-in displays a form. As an advantage over simple plug-ins the entered form data can be used while processing the users request. It is recommended to use this plug-in type, when the displayed form has to be generated dynamically. If the general shape of the form is indepent of a concrete model, or model state, please use *DialogPlugin* as superclass (cf. Section 4).

When the form is sent back to the server it is delivered in a special format called multipart. For details on multipart see Section 3.2.4. The implementer of the plug-in does not have to care about parsing this format since that is done before the respective methods are called.

3.2.2 Methods

void performAction(RequestFacade req, ResponseFacade resp, MultiPartObject mpo, ModelInformation mi, LoginableUser u); After the form has been submitted, this method is responsible for processing the incoming request and sending a resulting response to the requester. All available information is contained within the *ModelInformation* and the *MultiPartObject* objects. The latter contains the data the user entered into the form belonging to this plug-in.

See Section 3.2.4 for information on the multipart format.

See Section 6 for information on how responses must be structured.

JSONObject getFormConfig(ModelInformation mi, RequestFacade req, LoginableUser lu) Return a *JSONObject* that configures the required form. To facilitate this step there exist several classes within the package `com.inubit.research. server.extjs`. Detailed information on these classes is given in Section 3.2.3. Two buttons, "Submit" and "Cancel" are automatically added to the form.

String getItemText(); Return the text for the menu item. If the `showInToolbar()`-method for this plug-in returns `true` the item text will be used as tooltip text instead.

String getItemIconPath(); Return the path to the icon of the menu item. This is especially important if the plug-in is represented as a simple button within the toolbar or as a context menu button.

3.2.3 ExtJS-Form Creation

The creation of ExtJS form configurations is facilitated by the *ExtJSFormFactory* class. By calling `createEmptyForm()` you will receive an empty form to which you can add all required items. As all objects returned by the factory extend the class *JSONObject*, no further conversion is required.

Supported ExtJS form elements are (further elements may be implemented):

- Container elements (can have multiple sub-elements):
  - FieldSet
  - CheckboxGroup
- Simple elements:
  - Checkbox
  - TextField

For configuring these elements (and also the form itself) use the respective `setProperty (key, value)` method. To view all configurable attributes take a look at `http://www.extjs.com/deploy/dev/docs/`.

For accessing the entered data during request processing, you have to specify the **name**-attribute for each simple element. This name can then be used to get the corresponding value out of the created *MultiPartObject* instance.

3.2.4 Multipart Format

When the user submits the form, the server receives the data in multipart format. In a first step this format is transferred into a Java object of type *MultiPartObject*. Accessing this object will deliver the entered data to the plug-in.

The following methods are considered to be helpful, where `mpo` is an instance of class *MultiPartObject*, `mi` is a *MultiPartItem*, and `mp` is a *SimpleMultipartParser*:

- `mbo.getItems()`
  This returns all items contained in the multipart object. That means, that all none empty (or unset) form elements are returned by this call.
- `mbo.getItemByName(String name)`
  Return one specific item that is identified by its unique name. The name is taken from the **name**-attribute of the form element. The form element is only found if the name exists and the element has a none-`null` value. For checkbox elements this means, that they are only part of the submitted form if they were checked when submitting the form.
- `mbi.getContent()`
  Get the textual content of a form element. This is equal to the value the user entered into the specific field.
- `sp.parseItemContentAsByteArray(BufferedInputStream bis, String itemName)`
  Reads a specific item of the input stream as byte array. This can be used, e.g., to parse an image

4 Dialog Plugins

4.1 Overview

Required superclass	*DialogServerPlugin*
Configurable Sub-URLs	*no*
Requires JavaScript	*yes*
Methods to implement	*JSONObject getData(ModelInformation mi , RequestFacade req)*
	JSONObject saveData(JSONArray data, ModelInformation mi)
	***(optional)** String getItemText()*
	***(optional)** String getItemIconPath()*
Graphical representation	*a single menu item or context button*
Example(s)	*Domain Class Attribute Dialog*

4.2 Details

4.2.1 Basic Information

This type of plug-in is similar to form plug-ins discussed within the last section. The main difference is that the basic structure of the dialog is specified within a JavaScript file (cf. Section 4.2.3). When the plug-in is loaded into the editor the required JavaScript file(s) is(are) loaded into the client and the dialog. If the respective menu item or button gets clicked, the dialog is instantiated and presented to the user within an Ext.Window.

To ensure the described functionality, the implementer must take care of the following aspects:

- The JavaScript dialog class must conform to the requirements specified in Section 4.2.3
- The plug-in's constructor must set the field `jsFiles`, to specify the JavaScript files that are specific for this plug-in.
- The plug-in's constructor must set the field `mainClassName`, specifying the name of the main JavaScript dialog class

In contrast to form plug-ins, the dynamic aspects of dialog plug-ins are reduced to loading and saving data using the respective methods.

4.2.2 Methods

JSONObject getData(ModelInformation mi, RequestFacade req); This method is responsible for collecting the data that should be loaded into the dialog. The concrete format of the data depends on the JavaScript implementations for this plug-in (cf. Section 4.2.3). The returned JSONObject must specify a field `data` to which the data is associated.

JSONObject saveData(JSONArray data, ModelInformation mi); This method is responsible for saving the data as it is returned from the client. The concrete format and number of the array entries depends on the JavaScript implementations for this plug-in (cf. Section 4.2.3). The returned JSONObject must be a valig plug-in response.

See Section 6 for information on how responses must be structured.

String getItemText(); Return the text for the menu item. If the `showInToolbar()`-method for this plug-in returns `true` the item text will be used as tooltip text instead.

String getItemIconPath(); Return the path to the icon of the menu item. This is especially important if the plug-in is represented as a simple button within the toolbar or as a context menu button.

4.2.3 JavaScript

The main JavaScript class, i.e. the class representing the dialog, must inherit from `Ext.ux.Dialog`. This class, specifying 3 additional methods, is a sub-class of `Ext.Panel` and, thereby, also provides all standard panel methods. Listing 1 depicts the specification of `Ext.ux.Dialog`.

Please note, `setData` must be coordinated with the implementation of the server-side method `getData` since the therein specified data format must be processed by the JavaScript method. Accordingly, `getJSONData` must be coordinated with the server-side method `saveData` since the latter has to process the data created by the former.

The method `getTitle()` is a convenience function form determining the dialog's title.

In case you are not familiar with ExtJS inheritance mechanism, take a look at `http://dev.sencha.com/deploy/dev/docs/?class=Ext`. The documentation of the method `extend` presents the most important practical information.

Listing 1: Ext.ux.Dialog

```
Ext.namespace("Ext.ux");

Ext.ux.Dialog = Ext.extend( Ext.Panel, {
    constructor : function( config ) {
        Ext.ux.Dialog.superclass.constructor.call(this, config);
    },

    setData : function( data ) {
        //process the data returned by the server
    },

    getJSONData : function() {
      /*
        * collect the data that should be sent to the server in
        * JSON format
        */
    },

    getTitle : function() {
        //return the title of this dialog
    }
});
```

5 Info-Dialog Plug-Ins

This class provides a minor extension of standard dialog plug-ins. The main difference is that the dialogs have only informational purpose but cannot be used to save modified data at the server. The specified dialogs are shown as tabs below the editor (see Figure 3). Except for `saveData()`, the implementer has to implement the same methods as for dialog plug-ins (cf. Section 4).

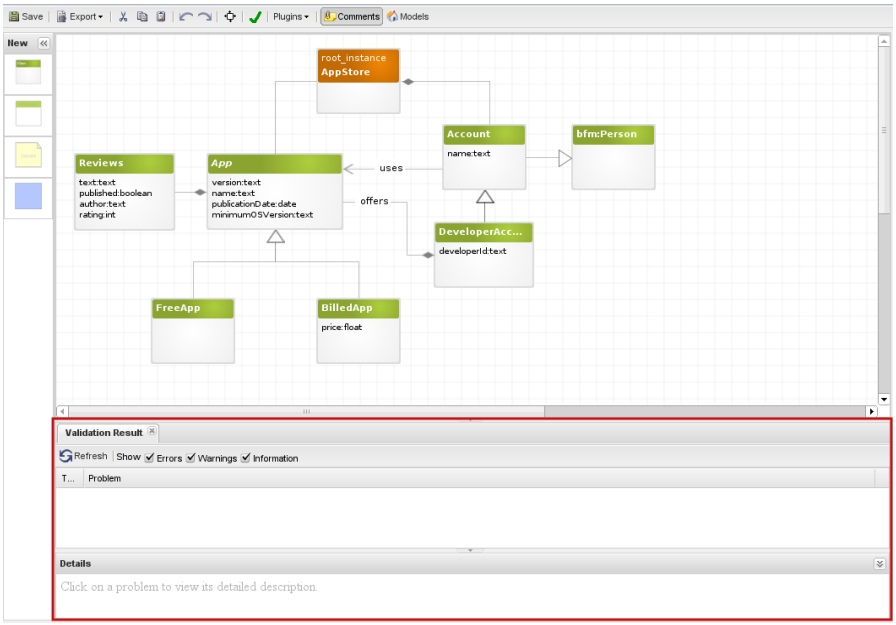


Figure 3: Display area for info-dialog plug-ins

5.1 JavaScript

In contrast to standard dialog plug-ins, the main JavaScript class has to inherit from `Ext.ux.InfoDialog`, which is itself an extension of `Ext.ux.Dialog`. Listing 2 displays the source code of `Ext.ux.InfoDialog`. As one can see, the dialog has access to the plug-in it is created for. Thereby, the dialog is able to refresh the displayed data by calling `this.plugin.refreshData()`. This refresh mechanism can be used ,e.g., to implement a refresh button.

Listing 2: Ext.ux.Dialog

```
Ext.namespace("Ext.ux");

Ext.ux.InfoDialog = Ext.extend( Ext.ux.Dialog, {
    constructor : function( config ) {
        Ext.ux.InfoDialog.superclass.constructor.call(this, config);
    },

    setPlugin : function( plugin ) {
        this.plugin = plugin;
    }
});
```

6 Plug-In Responses

The body of the HTTP response of a plug-in consists of a flag indicating the success of the operation, an action, and a data block that is required for processing the action. Responses are delivered in JSON format. The possible actions are listed in the enumeration *PluginResponseType* which is contained in `ServerPlugin.java`. The following sections describe each action in detail.

6.1 OPEN - Action

Description	Opens a new window or browser tab (depending on browser). This can usually be used to open new models.
Required data	The URI that should be opened.

Remark To add new temporary models to the model manager and return the corresponding URL, the code of Listing 3 in Section 7 might help.

6.1.1 Response Format

```
{
  success: true,
  action: 'OPEN',
  uri: '/your/valid/uri/starting/with/slash'
}
```

6.2 UPDATE - Action

Description	Update the current model. This includes changes of node and edge representations, their positions, as well as their creation, or removal. Changes of element positions are animated.
Required data	Model difference information. For details see the description in Section 6.2.1

6.2.1 Model Difference Data

Model difference data is used to communicate server side changes of a model to the editor. It states which nodes and edges have been changed, removed, or created and provides the necessary data. For details on the data format see class *ModelDifferenceTracker*.

Listing 4 shows an example of how a *ModelDifferenceTracker* can be used to collect the necessary information.

6.2.2 Response Format

```
{
  success: true,
  action: 'UPDATE',
  data: <model difference data>
}
```

6.3 INFO - Action

Description	Display an informative message to the user. The generated dialog will show an INFO icon.
Required data	The message that should be displayed.

6.3.1 Response Format

```
{
  success: true,
  action: 'INFO',
  infomsg: 'Your message to display'
}
```

6.4 ERROR - Action

Description	Display an error message to the user. The generated dialog will show an ERROR icon.
Required data	The message that should be displayed.

6.4.1 JSON Response Format

```
{
  success: false
  action: 'ERROR',
  errormsg: 'Your message to display'
}
```

7 Code Examples

Listing 3: Add a new temporary model to the model manager

```
ProcessModel pm = <your code to get new model>;

String newUri = ModelManager.getInstance().addTemporaryModel(pm);
//newUri holds the complete URI relative to the server's root-URI
```

Listing 4: Track model changes and create model difference JSON

```
RequestFacade req;
ProcessModel pm = <your code to get new model>;

ModelDifferenceTracker mdt = new ModelDifferenceTracker( pm );
pm.addListener(mdt);

//apply your custom changes to the model

pm.removeListener(mdt);
JSONObject jo = mdt.toJSON(ModelRequestHandler
                                .getAbsoluteAddressPrefix(req));
```