Web Client Plug-in Developers Guide

Intended Audience

This document was created as a walk-through tutorial on how to create custom plug-in modules that extend the functionality of the base i2b2 Web Client.

The author(s) assume that the reader is an experienced software developer or architect who has some experience programming JavaScript and some knowledge of advanced JavaScript topics such as JSON, AJAX, and using JavaScript toolkit (APIs).

Plug-In Example #1 – Hello World

The first example plug-in is a Hello World example. This example demonstrates the steps needed to setup the directory structure, create a simple plug-in configuration file, register a plug-in with the web client framework, and have an example HTML interface loaded by the framework.

Creating a Plug-in's base directory

Your deployment of the i2b2 web client will have a directory which contains all the JavaScript code related to the i2b2 web client Framework. The default location of this directory is **/js-i2b2**.

This directory contains the Framework configuration file (**i2b2_loader.js**) and two directories (**cells** and **hive**). Within the **cells** directory are the directories for the core cells in addition to the **plug-ins** folder. Within the plug-ins folder we will create a new folder to hold our examples called "*examples*".

Within the **examples** folder create a new folder called **ExampHello**. We will be using the string "ExampHello" as the code that will be used to uniquely identify the plug-in from other plug-ins and cells. For our example we chose to use the abbreviation code "ExampHello" for our aptly-named "Hello World" example plug-in.

Within every plug-in directory there is an **assets** directory to contain all multimedia assets that are used by the plug-in. All CSS, HTML and image files are contained within this directory.

```
Create the directory "/js-i2b2/cells/plugins/examples"
Create the directory "/js-i2b2/cells/plugins/examples/ExampHello"
Create the directory "/js-i2b2/cells/plugins/examples/ExampHello
/assets"
```

Registering a Plug-in with the Web Client Framework

For the Framework to be aware of our new plug-in, we must add an entry to the *module loader configuration* file. Within the file **/js-i2b2/i2b2_loader.js** is a section containing JSON-based configuration information which has the following structure:

```
// THESE ARE ALL THE CELLS THAT ARE INSTALLED ONTO THE SERVER

i2b2.hive.tempCellsList = [
   { code: "PM",
   forceLoading: true //    <---- this must be set to true for the PM cell!
   },
   { code: "ONT" } ,
   { code: "CRC" } ,
   { code: "PLUGINMGR" ,
   forceLoading: true,
   forceConfigMsg: {
      params: []
  }
   }
];
```

This JSON structure is used to register a list of cells and plug-in modules that the framework will attempt to load if the authenticated user has been authorized to use them (via the data returned from the PM cell during successful login). The above example code listing has information for registering the following cells/plug-ins (in order):

1. Project Management Cell (forced to automatically load when the framework is loaded)
2. Ontology Cell
3. Data Repository Cell
4. Plugin Viewer Module (used to manage all non-Cell plug-in modules)

Unless a custom module is going to be running as an i2b2-compliant cell, configuration options must be included in this file using the "forcing" options below:

Configuration Option	Description
forceLoading: (Boolean)	Is the module automatically loaded during framework initialization
forceConfigMsg: (Object)	This data object is automatically populated to **i2b2.CELLCODE.cfg.config** when the module is loaded

These options can also be used to override configuration information that is being returned to the web client framework from the Project Management Cell during login authorization.
To do so we add the following to the configuration that results in the following file with our added configuration information:

> **Tip**
>
> Remember to add a comma to the end of the previous configuration in the list.

```
// THESE ARE ALL THE CELLS THAT ARE INSTALLED ONTO THE SERVER

i2b2.hive.tempCellsList = [
   { code: "PM",
   forceLoading: true //    <---- this must be set to true for the PM cell!
   },
   { code: "ONT" } ,
   { code: "CRC" } ,
   { code: "PLUGINMGR" ,
   forceLoading: true,
   forceConfigMsg: {
      params: []
   }
   },
   {code: "ExampHello" ,
   forceLoading: true, //    <---- this must be set to true for all plugins
   forceDir: "cells/plugins/examples"
      params: []
   }
];
```

> **Important**
>
> It is important to know that the name provided in the code configuration attribute will be the namespace that mustbe used by your plug-in. the code "ExampHello" resolves to an absolute namespace of thei2b2.ExampHellowithin the JavaScript VM.

Creating a Plug-in's Configuration File

For the Framework to be able to properly load our new plug-in module, we must declare which files comprise the new module. This is accomplished by creating a JSON-based configuration file within the module's root directory. For our example that would be

/js-i2b2/cells/plugins/examples/ExampHello/cell_config_data.js

The file would have the following structure:

```
// This file contains a list of all files that need to be loaded dynamically
// for this module. Every file in this list will be loaded after the module's Init()

// function is called
{
  files:[ "ExampHello.js" ],
  css:[ "ExampHello.css" ],
  config: {
    // additional configuration variables that are set by the system
    short_name: "Hello World",
    name: "Example #1 - Hello World",
    description: "This plugin cell demonstrates how to register your plugin
      with the i2b2 thin-client framework and display simple HTML.",
    category: ["celless", "plugin", "examples"],
    plugin: {
      isolateHtml: false,
      html: {
        source: 'injected_screens.html',
        mainDivId: 'ExampHello-mainDiv'
      }
    }
  }
}
```

The configuration file has three main parts to it:

1. A list of JavaScript files
2. A list of HTML CSS files
3. A configuration section.

The files and CSS configuration sections are self-explanatory. All filenames listed will be prepended the with the plug-in's base directory to generate the full file access location used by the framework in loading the files.

Important

It is important to note that all version of Microsoft Internet Explorer limit the number of dynamically loaded style sheets to a total of 31 files. The web client framework uses several style sheets and each cell or plug-in may have dynamically loaded style sheets as well.

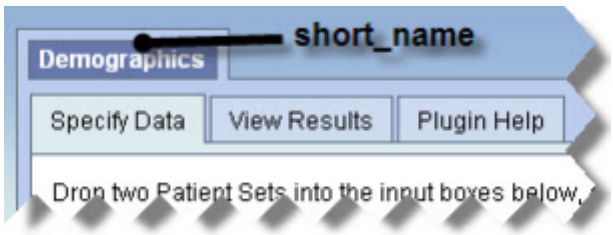
An important best practice is to only define one CSS file in your plug-in's configuration file.

To use more than one CSS file in your plug-in, create a CSS file to subsequently load your other CSS files using the @import (file.css) command.

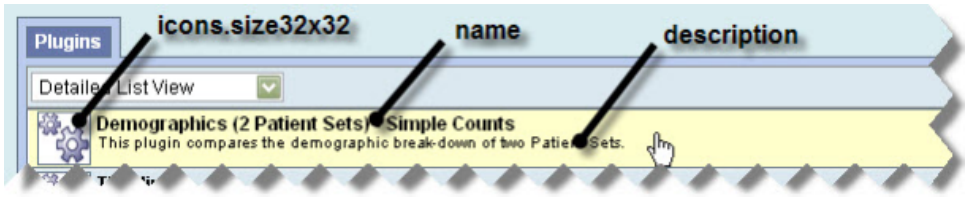
The configuration section contains various pieces of information that are used by the Framework. They are explained below:

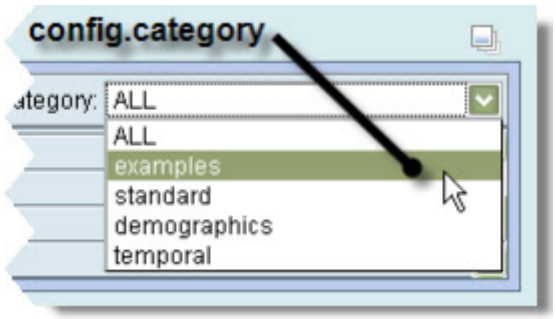
JSON Configuration Variable	Description
config.short_name	Is displayed in the title tab area of the plug-in viewer's display window.
config.name	The title string that is displayed in the plug-in viewer's listing window.
config.description	The description that is displayed in the plug-in viewer's listing window.
config.category	A list of categories that this plug-in is a member of. All plug-ins must include "plugin" value. If the plug-in does not have its own backend cell then "celless" value should also be present.
config.icons	JSON object defining one or more icon files. These files must be located in the **assets** directory of the plug-in's base directory.
config.icons.size32x32	Filename for 32x32 pixel icon used in the plug-in listing window when in detailed view mode. (The fill must be in the plug-in's **assets** directory).
config.icons.size16x16	Filename for 16x16 pixel icon used in the plug-in listing window when in summary view mode. (The fill must be in the plug-in's **assets** directory).
config.plugin	JSON object which defines and configures the module as a plug-in.
config.plugin.isolateHtml	Boolean, should the framework isolated the plug-in's HTML an IFRAME.
config.plugin.html	JSON object that contains information about the plug-in's display HTML.
config.plugin.htm.source	Filename for the plug-in's display HTML (in the local **assets** directory).

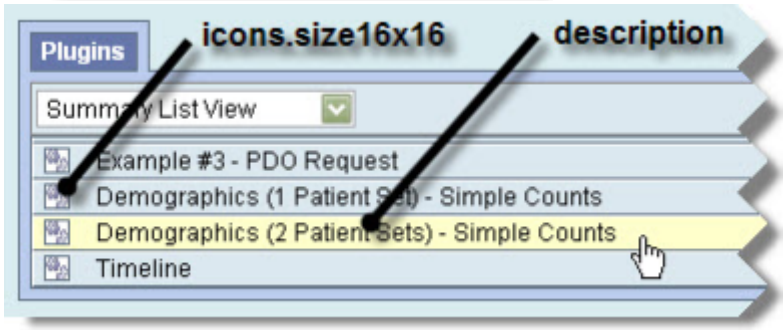
config.plugin.html. mainDivId	The unique ID of the HTML Element (in the above declared source file) whose contents will be initially displayed.



Most of the information within the configuration section is used by the plugin viewer subsystem in ways that are reflected in the user interface.







Displaying HTML

The first step in the development of any plug-in module is the creation of a user interface. The i2b2 web framework eliminates the complexity of transforming simple HTML screens into an AJAX-based application.
To start, an HTML file must be created and put into the plug-in module's asset directory (**/js-i2b2/cells/plugins/examples/ExampHello/assets**). For our "Hello World" we will create a file called **injected_screens.html** and fill it with the following HTML:

```
<html>
  <body>
  <div id="ExampHello-mainDiv">
    This "Hello World" demonstrates how to register your plugin with the i2b2 Thin-Client framework and display some HTML.

  </div>
  <div>
    This text will not display because it is outside our targeted div
  </div>
  </body>
</html>
```

The text which emphasis added is what we want to display in the plug-in viewer's display window when the plug-in is selected and loaded.
The next step is to configure the plug-in (via the **cell_config_data.js** file) with the correct HTML file and ID for the HTML Element that contains the initial UI display.

```
plugin: {
  isolateHTML: false,
  html: {
  source: 'injected_screens.html' ,
  mainDivId: 'ExampHello-mainDiv' ,

  }
}
```

Since our configuration file has defined a CSS file to load we must create that file for the plug-in module to work properly. We will fully utilize the separate CSS file later in this document but will need to create a blank file (**/js-i2b2/cells/plugins/ExampHello/assets/vwHello.css**) for now.

> **Tip**
>
> For more information on the configuration files and in particular the plug-in configuration files please see the document called Web Client Configuration Files.

After saving all files, *clearing your browser's cache* (just in case) and refreshing the screen, you should see the "Hello World – Simple HTML" plug-in in the "Plugins" list window. When the listing is clicked, it should load the plug-in and display the HTML you created above.

Creating Constructor & Destructor Functions

> **Tip**
>
> Although it is not absolutely necessary, it is good practice to create constructor and destructor functions for your plug-in.

The plug-in constructor function is called after the *Plugin Viewer* has loaded the requested plug-in's initial HTML into the main DOM tree. When the constructor is called it is passed a reference to the DOM node that contains the plug-in's HTML.
If your plug-in contains a destructor function, it is called by the Plugin Viewer before it removes the plug-in's GUI from the main DOM tree.

> **Be Careful**
>
> If your destructor function does not return **true** then the unload process is canceled.

```
i2b2.ExampHello.Init = function(loadedDiv) {
   // this function is called after the HTML is loaded into the viewer DIV

   i2b2.ExampHello.view.containerDiv = loadedDiv;
   alert("Hello World! This message is from the initialization routine.");

};
```

```
i2b2.ExampHello.Unload = function() {
   // this routine should be used to save the state of the plugin so that work can

   // resume if reloaded or to release memory currently held in plugin variables.

   return confirm("Are you sure you want to unload the Hello World plugin?");
};
```

Plug-In Example #2 – Tabs and Drag / Drop

Starting from our *Hello World plug-in* it is easy to build a second plug-in that uses standard tabs display and can accept SDX drop messages. Begin building this plug-in by copying the previous plug-in example into the directory **/js-i2b2/cells/plugins/ExampTabs** and change the filenames to reflect the naming convention of the new plug-in, resulting in the filenames:

1. ExampTabs.js
2. ExampTabs.css

The new filenames should be reflected in the cell configuration file found at:

/js-i2b2/cells/plugins/ExampTabs/cell_config_data.js

To register the new plug-in in the *module loader configuration file* add the following lines:

```
// THESE ARE ALL THE CELLS THAT ARE INSTALLED ONTO THE SERVER

i2b2.hive.tempCellsList = [
    { code: "PM",
    forceLoading: true // <------- this must be set to true for the PM cell!
    },
    { code: "ONT" } ,
    { code: "CRC" } ,
    { code: "PLUGINMGR" ,
    forceLoading: true,
    forceConfigMsg: {
        params: []
    }
   },
    {code: "ExampHello" ,
    forceLoading: true, // <------- this must be set to true for all plugins
    forceDir: "cells/plugins/examples"
        params: []
    },
    {code: "ExampTabs" ,
    forceLoading: true,
    forceDir: "cells/plugins/examples"
    forceConfigMsg: {
        DefaultTab: 3 // <------- define variable for later use
    }
   }
];
```

In this "Tabs Example" we will have the plug-in read a configuration variable called **DefaultTab** and use its value to display a specific tab when the plug-in is loaded. This is to present an example of how to use a single variable but the same principle applies if several values are needed. You will see how this value is used later in the section called *Setup Standard Display Tabs*.

Setup Standard Display Tabs

It is also possible for your plug-in to use standardized tabs to display of more than one page while loaded in the Plugin Viewer window. To do this, you need to add specific HTML code to your display file. The two parts of HTML represent the *tab markup* and the *sub-screens markup*.
All markup related to tabs-based functionality must exist within a DIV located at the root level of the loading DIV, in this example the following is needed:

- The DIV will need to have an *id* value of **ExampTabs-mainDiv**.
- The *tab-encapsulating DIV* must be assigned the class **yui-navset**.
- The *tab rendering routines* are expecting to find three child nodes, an unordered list assigned a class of **yui-nav** and a DIV assigned the class **yui-content**.

```
<html>
    <body>
    <div id="ExampTabs-mainDiv">
        <div id="ExampTabs-TABS" class="yui-navset">

            <!--{-}Define the tabs -->
            <ul class="yui-nav">
            <li id="ExampTabs-TAB0">
                <a href="#ExampTabs-TAB0"><em>Specify Data</em></a>
            </ul>
            <li id="ExampTabs-TAB1">
                <a href="#ExampTabs-TAB1"><em>View Results</em></a>
            </ul>
            <li id="ExampTabs-TAB2">
                <a href="#ExampTabs-TAB2"><em>Plugin Help</em></a>
            </ul>
            </ul>

            <!--{-}Define the tabs sub-screens -->
            <div class="yui-content" id="ExampTabs-CONTENT">
                <div>This "Hello World" demonstration text shows up on Tab #1</div>

                <div>This "Hello World" demonstration text shows up on Tab #2</div>

                <div>This "Hello World" demonstration text shows up on Tab #3</div>

            </div>
        </div>
    </div>
    </body>
</html>
```

A line must be added to the **plug-in configuration file** so that the tabs menu will be recognized and properly drawn by the *Plugin Viewer*.

```
plugin: {
    isolateHtml: false, // this means do not use an IFRAME
    standardTabs: true, // the plugin viewer will display a standard tabs

    html: {
    source: 'injected_screens.html',
    mainDivId: 'ExampTabs-mainDiv',
}
```

The final change required will be to put the following line into the **initialization routine** of the plug-in. This example also used the **configuration variable set** in the *module loader configuration file* as the default tab shown when the plug-in is loaded.

```
i2b2.ExampTabs.Init = function (loadedDiv) {
    // this function is called after the HTML is loaded into the viewer DIV
    i2b2.ExampTabs.view.containerDiv = loadedDiv; // save reference for later use

    var cfgObj = {activeIndex : i2b2.ExampTabs.cfg.config.DefaultTab – 1 };
    this yuiTabs = new YAHOO.widget.TabView("ExampTabs-TABS", cfgObj);

};
```

Add HTML to First Tab

In this example the first tab will be related to data entry. As such, we are going to create a DIV and register it with the SDX subsystem to allow it to recognize when data objects are dropped onto it.
In order for the DIV to be registered, it must have an **id** attribute, in this example we will call it **ExampTabs-DROPTRGT**.

```
<html>
    <body>
    <div id="ExampTabs-mainDiv">
        <div id="ExampTabs-TABS" class="yui-navset">

            <!--{-}Define the tabs –> [removed in this example]

            <!--{-}Define the tabs sub-screens –>
            <div class="yui-content" id="ExampTabs-CONTENT">
                <div>
                    <div class=" ExampTabs-MainContent">
                        <div class=" ExampTabs-MainContentPad">
                        <div>Drop an i2b2 object such as a search term or query onto the input box below, and then

                                click the "View Results" tab for information about that object.
                        </div>
                        <div class="droptrgtlbl">i2b2 Object:</div>
                        <div class="droptrgt SDX-PRS" id="ExampTabs-DROPTRGT">
                            Drop an object here
                        </div>
                    </div>
                </div>
            </div>
            <div>This "Hello World" demonstration text shows up on Tab #2</div>
            <div>This "Hello World" demonstration text shows up on Tab #3</div>
        </div>
    </div>
    </body>
</html>
```

The above HTML contains the following three DIV

1. Directions for the screen
2. A label for the drop target
3. The drop target DIV.

The next step is to create the code needed to initialize and manage the SDX drop operations.

Add SDX Registration Code

In the plug-in's initialization routine we need to register the above-defined DIV to accept SDX drag drop operations for various data types handled by the SDX subsystem.

```
i2b2.ExampTabs.Init = function (loadedDiv) {
    // this function is called after the HTML is loaded into the viewer DIV
    i2b2.ExampTabs.view.containerDiv = loadedDiv; // save reference for later use
    var cfgObj = {activeIndex : i2b2.ExampTabs.cfg.config.DefaultTab – 1 };
    this yuiTabs = new YAHOO.widget.TabView("ExampTabs-TABS", cfgObj);

    // register DIV as valid DragDrop target for Patient Record Sets (PRS) objects

    var divName = "ExampTabs-DROPTRGT";

    // register for drop events of the following datatypes
    var op_trgt = {dropTarget:true};
    i2b2.sdx.Master.AttachType(divName, 'CONCPT', op_trgt);
    i2b2.sdx.Master.AttachType(divName, 'QM', op_trgt);
    i2b2.sdx.Master.AttachType(divName, 'QI', op_trgt);
    i2b2.sdx.Master.AttachType(divName, 'PRS', op_trgt);
    i2b2.sdx.Master.AttachType(divName, 'PRC', op_trgt);
    i2b2.sdx.Master.AttachType(divName, 'PR', op_trgt);
    i2b2.sdx.Master.AttachType(divName, 'QDEF', op_trgt);
    i2b2.sdx.Master.AttachType(divName, 'QGDEF', op_trgt);
    i2b2.sdx.Master.AttachType(divName, 'XML', op_trgt);
};
```

Add Code for Drop Event Handler

A *function* needs to be added to the plug-in's main code file to accept the data from the SDX drop event. In this example we are going to be adding the function call **doDrop()** to the plug-in's root namespace **i2b2.ExampTabs**.

This function, like all SDX Drop event handlers, must accept an array containing one or more SDX Data Messages. For this example we will only be processing the first message.

```
i2b2.ExampTabs.doDrop = function (sdxData) {
    // only interested in first SDX record in the MSG
    sdxData = sdxData[0];

    // save the info to our local data model
    i2b2.ExampTabs.model.currentRec = sdxData;

    // let the user know that the drop was successful
    // by displaying the name of the object
    $("ExampTabs-PRSDROP").innerHTML = i2b2.h.Escape(sdxData.sdxInfo.sdxDisplayName);


    // our input dataset has changed and any analysis needs to be redone
    i2b2.ExampTabs.model.dirtyResultsData = true;
};
```

The above code performs the following actions:

- saves the dropped data into the data model namespace of this plug-in,
- provides visual feedback to the application user informing them that the drag & drop they performed was successful,
- set a flag within the plug-in's data model namespace to indicate that whatever calculated data is now invalidated (and thus "dirty") because the origin data has changed.

This step, along with processing being triggered on tab change, is the preferred method for managing the data input / processing / output display user interaction by plug-ins.

Register Drop Handler with the SDX Subsystem

Once a handler function is created we can register it to accept the drop events by having the SDX system call the example's local drop event handler instead of using the default handler for that SDX Data Message data type.
To do this properly, we are going to use a simple **demultiplexer function** that simply calls our real drop handler in the plug-in's root namespace. There are two reasons we do this instead of registering the real drop handler directly:

1. To remap the scope of JavaScript's this identifier so that it refers to the plug-in's base namespace
2. To save memory

When the SDX system is told to use an alternative drop event handler function, it saves an isolated clone of that function. If the same function is going to be registered to process 10 different types of SDX data then that function would be in memory 11 times (10 in the SDX system + the original copy of the function). This does not have a big impact if the cloned function is a small router function but can have a much larger impact if the drop handler is very large.

The code to map the drop event handler for this example is shown below:

```
i2b2.ExampTabs.Init = function (loadedDiv) {
    // this function is called after the HTML is loaded into the viewer DIV
    i2b2.ExampTabs.view.containerDiv = loadedDiv; // save reference for later use
    var cfgObj = {activeIndex : i2b2.ExampTabs.cfg.config.DefaultTab – 1 };
    this yuiTabs = new YAHOO.widget.TabView("ExampTabs-TABS", cfgObj);

    // register DIV as valid DragDrop target for Patient Record Sets (PRS) objects
    var divName = "ExampTabs-DROPTRGT";

    // register for drop events of the following datatypes
    var op_trgt = {dropTarget:true};
    i2b2.sdx.Master.AttachType(divName, 'CONCPT', op_trgt);
    i2b2.sdx.Master.AttachType(divName, 'QM', op_trgt);
    i2b2.sdx.Master.AttachType(divName, 'QI', op_trgt);
    i2b2.sdx.Master.AttachType(divName, 'PRS', op_trgt);
    i2b2.sdx.Master.AttachType(divName, 'PRC', op_trgt);
    i2b2.sdx.Master.AttachType(divName, 'PR', op_trgt);
    i2b2.sdx.Master.AttachType(divName, 'QDEF', op_trgt);
    i2b2.sdx.Master.AttachType(divName, 'QGDEF', op_trgt);
    i2b2.sdx.Master.AttachType(divName, 'XML', op_trgt);

    // Use an event router to demultiplex the callbacks to the single drop even handler used by
    // this plugin. This also recreates a properly scoped "this" keyword within the event handler

    var eventRouterFunct = (function (sdxData) { i2b2.ExampTabs.doDrop(sdxData);});

    // register the event handler via eventRouterFunc
    var refSDXMaster = i2b2.sdx.Master;
    refSDXMaster.setHandlerCustom(divName, 'CONCPT', 'DropHandler', eventRouterFunc);
    refSDXMaster.setHandlerCustom(divName, 'QM', 'DropHandler', eventRouterFunc);
    refSDXMaster.setHandlerCustom(divName, 'QI', 'DropHandler', eventRouterFunc);
    refSDXMaster.setHandlerCustom(divName, 'PRS', 'DropHandler', eventRouterFunc);
    refSDXMaster.setHandlerCustom(divName, 'PRC', 'DropHandler', eventRouterFunc);
    refSDXMaster.setHandlerCustom(divName, 'PR', 'DropHandler', eventRouterFunc);
    refSDXMaster.setHandlerCustom(divName, 'QDEF', 'DropHandler', eventRouterFunc);
    refSDXMaster.setHandlerCustom(divName, 'QGDEF', 'DropHandler', eventRouterFunc);
    refSDXMaster.setHandlerCustom(divName, 'XML', 'DropHandler', eventRouterFunc);
};
```

Add HTML to the Second Tab

At this point in the example's development we have three working tabs with the first tab containing text and a target DIV that accepts SDX drop events and saves the dropped information. The next step is to create an output screen to display the results of processing the dropped SDX data.

```
<html>
    <body>
    <div id="ExampTabs-mainDiv">
        <div id="ExampTabs-TABS" class="yui-navset">

            <!--{-}Define the tabs –> [removed in this example]

            <!--{-}Define the tabs sub-screens –>
            <div class="yui-content" id="ExampTabs-CONTENT">
                <div>[TAB 1's screen removed in this example]</div>
                <div>
                    <div class="ExampTabs-MainContent">
                        <div class="results-directions">
                            You must first provide an i2b2 object. Please return to the "Specify Data" tab

                            then drag and drop an i2b2 object onto the input box.
                        </div>
                        <div class="results-finished" style="display:none;">
                            <div class="results-text">
                                The following information was passed to the plugin in the i2b2 object:
                            </div>
                            <div id="ExampTabs-InfoSDX">
                                <table>
                                    <tr>
                                        <th>Display Name</th>
                                        <td class="sdxDisplayName"></td>
                                    </tr>
                                    <tr>
                                        <th>SDX Datatype Code</th>
                                        <td class="sdxType"></td>
                                    </tr>
                                    <tr>
                                        <th>Controlling Cell</th>
                                        <td class="sdxControlCell"></td>
                                    </tr>
                                    <tr>
                                        <th>Primary Key Column</th>
                                        <td class="sdxKeyName"></td>
                                    </tr>
                                    <tr>
                                        <th>Primary Key Value</th>
                                        <td class="sdxKeyValue"></td>
                                    </tr>
                                    <tr>
                                        <th>Original XML Data</th>
                                        <td class="sdxOriginalXML">
                                            <div class="sdxOriginalXML"></div>
                                        </td>
                                    </tr>
                                </table>
                            </div>
                        </div>
                    </div>
                </div>
                <div>This "Hello World" demonstration text shows up on Tab #3</div>
            </div>
        </div>
    </div>
    </body>
</html>
```

Add a Processing Routine to the Plug-in

Now that we have our output defined we can create the code needed to process the previously saved dropped data into the form that will be displayed in the second tab. The processing done by this example is to simply display the information that was contained within the SDX message.

```
i2b2.ExampTabs.getResults = function () {
    // Refresh the display with info of the SDX record that was DragDropped
    if (i2b2.ExampTabs.model.dirtyResultsData) {
    var dropRecord = i2b2.ExampTabs.model.currentRec;

    // hide the directions DIV and show the output DIVs
    $$("DIV#ExampTabs-mainDiv DIV.results-directions")[0].hide();
    $$("DIV#ExampTabs-mainDiv DIV.results-finished")[0].show();

    var sdxDisplay = $$("DIV#ExampTabs-mainDiv DIV#ExampTabs-InfoSDX")[0];


    var tempRefDIV = false;
    tempRefDIV = Element.select(sdxDisplay, '.sdxDisplayName')[0];
    tempRefDIV.innerHTML = dropRecord.sdxInfo.sdxDisplayName;

    tempRefDIV = Element.select(sdxDisplay, '.sdxType')[0];
    tempRefDIV.innerHTML = dropRecord.sdxInfo.sdxType;

    tempRefDIV = Element.select(sdxDisplay, '.sdxControlCell')[0];
    tempRefDIV.innerHTML = dropRecord.sdxInfo.sdxControlCell;

    tempRefDIV = Element.select(sdxDisplay, '.sdxKeyName')[0];
    tempRefDIV.innerHTML = dropRecord.sdxInfo.sdxKeyName;

    tempRefDIV = Element.select(sdxDisplay, '.sdxKeyValue')[0];
    tempRefDIV.innerHTML = dropRecord.sdxInfo.sdxKeyValue;

    // Escape the XML text or the browser will attempt to interpret it as HTML

    var xmlDisplay = i2b2.h.Xml2String(dropRecord.origData.xmlOrig);
    xmlDisplay = '<pre>'+ i2b2.h.Escape(xmlDisplay)+'</pre>';
    Element.select(sdxDisplay, '.originalXML') [0] .innerHTML = xmlDisplay ;
    }

    // optimization – only requery when the input data is changed
    i2b2.ExampTabs.model.dirtyResultsData = false;
}
```

Cause Processing to Occur on Tab Change

At this point the example, the plug-in has a working input method, a processing routine, and an output screen. All that is left is to do is to capture an event to initiate processing and displaying of the data.

The most user-transparent event we can capture would be the user switching to the results tab. If the data is dirty when the user switches to the results tab then the input will be reprocessed. The code needed to perform in this way operates as follows:

1. Create a function and attach it to YUI's tab change event.
2. Check to see if the tab that the user has switched to is the results tab.
3. See if the plug-in has received any information from a drop event.
4. See if the saved data has been processed already (via "dirty data" flag).

```
i2b2.ExampTabs.Init = function (loadedDiv) {
    // this function is called after the HTML is loaded into the viewer DIV
    i2b2.ExampTabs.view.containerDiv = loadedDiv; // save reference for later use
    var cfgObj = {activeIndex : i2b2.ExampTabs.cfg.config.DefaultTab – 1 };
    this.yuiTabs = new YAHOO.widget.TabView("ExampTabs-TABS", cfgObj);

    // register DIV as valid DragDrop target for Patient Record Sets (PRS) objects
    var divName = "ExampTabs-DROPTRGT";

    // register for drop events of the following datatypes
    var op_trgt = {dropTarget:true};
    i2b2.sdx.Master.AttachType(divName, 'CONCPT', op_trgt);
    i2b2.sdx.Master.AttachType(divName, 'QM', op_trgt);
    i2b2.sdx.Master.AttachType(divName, 'QI', op_trgt);
    i2b2.sdx.Master.AttachType(divName, 'PRS', op_trgt);
    i2b2.sdx.Master.AttachType(divName, 'PRC', op_trgt);
    i2b2.sdx.Master.AttachType(divName, 'PR', op_trgt);
    i2b2.sdx.Master.AttachType(divName, 'QDEF', op_trgt);
    i2b2.sdx.Master.AttachType(divName, 'QGDEF', op_trgt);
    i2b2.sdx.Master.AttachType(divName, 'XML', op_trgt);

    // Use an event router to demultiplex the callbacks to the single drop event handler used by
    // this plugin. This also recreates a properly scoped "this" keyword within the event handler

    var eventRouterFunct = (function (sdxData) { i2b2.ExampTabs.doDrop(sdxData);});

    // register the event handler via eventRouterFunc
    var refSDXMaster = i2b2.sdx.Master;
    refSDXMaster.setHandlerCustom(divName, 'CONCPT', 'DropHandler', eventRouterFunc);
    refSDXMaster.setHandlerCustom(divName, 'QM', 'DropHandler', eventRouterFunc);
    refSDXMaster.setHandlerCustom(divName, 'QI', 'DropHandler', eventRouterFunc);
    refSDXMaster.setHandlerCustom(divName, 'PRS', 'DropHandler', eventRouterFunc);
    refSDXMaster.setHandlerCustom(divName, 'PRC', 'DropHandler', eventRouterFunc);
    refSDXMaster.setHandlerCustom(divName, 'PR', 'DropHandler', eventRouterFunc);
    refSDXMaster.setHandlerCustom(divName, 'QDEF', 'DropHandler', eventRouterFunc);
    refSDXMaster.setHandlerCustom(divName, 'QGDEF', 'DropHandler', eventRouterFunc);
    refSDXMaster.setHandlerCustom(divName, 'XML', 'DropHandler', eventRouterFunc);

    // Start Processing when user switches to the results tab IF the data is dirty
    this.yuiTabs.on('activeTabChange', function(ev) {
    //Tabs have changed
    if (ev.newValue.get('id')== "ExampTabs-TAB1") {
        // user switched to Results tab
        if (i2b2.ExampTabs.model.currectRec) {
        // gather statistics only if we have data
        if (i2b2.ExampTabs.model.dirtyResultsData) {
            // recalculate the results only if the input data has changed
            i2b2.ExampTabs.getResults();
        }
      }
    }
    });
};
```

Plug-in Example #3 – PDO Example

The third example plug-in adds the concept of communications with a **cell** in the form of a *PDO request*. To make this PDO Example plug-in we've taken the Tabs & Drag / Drop Example plug-in and made some changes and additions.

Setup Example

Begin building the PDO example plug-in by copying the previous plug-in example into the directory **/js-i2b2/cells/plugins/ExampPDO** and change the filenames to reflect the naming convention of the new plug-in, resulting in the following filenames:

1. ExampPDO.js
2. ExampPDO.css.

Update the plug-in configuration file with the information shown below. This file can be found at:

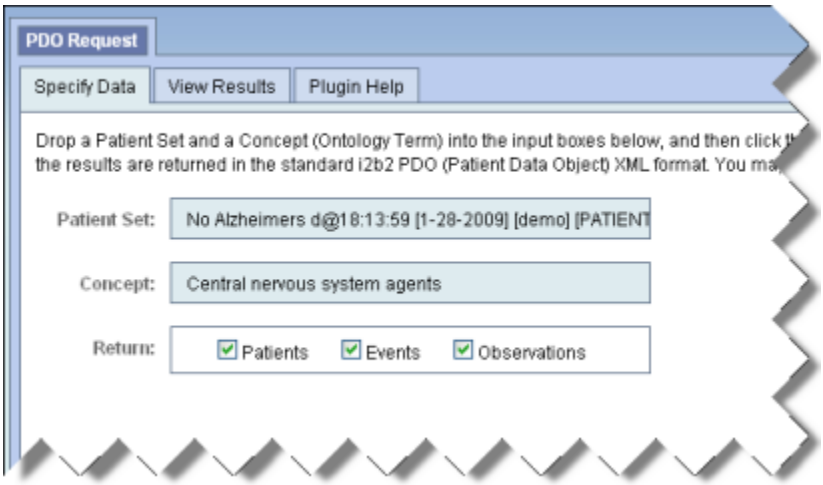
/js-i2b2/cells/plugins/ExampPDO/cell_config_data.js

```
// This file contains a list of files that need to be dynamically loaded for this
// plugin after its Init function is called
{
    files: ["ExampPDO.js" ],
    css: ["ExampPDO.css" ],
    config: {
    // additional configuration variables that are set by the system
    short_name: "PDO Request",
    name: "Example #3 - PDO Request",
    description: "This plugin demonstrates how to retrieve information using the Patient Data Object (PDO).",

    category: ["celless","plugin","examples"],
    plugin: {
        isolateHtml: false,
        standardTabs: true, // this means the plugin uses standard tabs at top
        html: {
            source: 'injected_screens.html' ,
            mainDivId: 'ExampPDO_mainDiv'
        }
    }
   }
}
```

The new plug-in was registered with the *module loader configuration* file by adding the following lines:

```
// THESE ARE ALL THE CELLS THAT ARE INSTALLED ONTO THE SERVER

i2b2.hive.tempCellsList = [
    { code: "PM",
    forceLoading: true // <------- this must be set to true for the PM cell!
    },
    { code: "ONT" } ,
    { code: "CRC" } ,
    { code: "PLUGINMGR" ,
    forceLoading: true,
    forceConfigMsg: {
    params: []
    }
    },
    {code: "ExampHello" ,
    forceLoading: true, // <------- this must be set to true for all plugins
    forceDir: "cells/plugins/examples"
    params: []
    },
    {code: "ExampTabs" ,
    forceLoading: true,
    forceDir: "cells/plugins/examples"
    },
    {code: "ExampPDO" ,
    forceLoading: true,
    forceDir: "cells/plugins/examples"
    }
];
```

HTML for Data Entry



The first data entry tab was modified to accept a *patient record set* and an *Ontology concept*. To do that we make the HTML for the first tab to be as follows:

```
<div class="ExampPDO-MainContent">
    <div class="ExampPDO-MainContentPad">
    <div>
        Drop a Patient Set and a Concept (Ontology Term) into the input boxes below, and then click the "View Results" tab to
retrieve
        information about when that concept was observed in the selected patient set. In this example, the results are returned in the
        standard i2b2 PDO (Patient Data Object) XML format. You may select whether to return information about the patients,
        events, or observations.
    </div>
    <div class="droptrgtlbl">Patient Set:</div>
    <div class="droptrgt SDX-PRS" id="ExampPDO-PRSDROP">
        Drop a Patient Set here
    </div>
    <div class="droptrgtlbl">Concept:</div>
    <div class="droptrgt SDX-CONCPT" id="ExampPDO-CONCPTDROP">
        Drop a Concept here
    </div>
    <div class="outputOptionslbl">Return:</div>
    <div class="outputOptions">
        <form>
            <span>
                <input type="checkbox" checked id="ExampPDO-OutputPatient">
                    Patients
            </span>
            <span>
                <input type="checkbox" checked id="ExampPDO-OutputEvents">
                    Events
            </span>
            <span>
                <input type="checkbox" checked id="ExampPDO-OutputObservations">
                    Observations
            </span>
        </form>
    </div>
    </div>
</div>
```

Setup Standard Tabs

This plug-in uses the standardized tabs system provided by the Plugin Viewer framework.

```
i2b2.ExampPDO.Init = function (loadedDiv) {
    // Manage YUI Tabs
    this.yuiTabs = new YAHOO.widget.TabView("ExampTabs-TABS", {activeIndex : 0});

}
```

Registering DIVs to Accept SDX Drop Events

The first and second highlighted parts in the previous section called "*HTML For Data Entry*" are the DIVs that will be used to accept SDX Drop operations. To register the first DIVs as **Drop event targets** the following code was incorporated into the plug-in's initialization routine:

```
i2b2.ExampPDO.Init = function (loadedDiv) {
    // Manage YUI Tabs
    this.yuiTabs = new YAHOO.widget.TabView("ExampPDO-TABS", {activeIndex : 0});

    // register the drop target DIVs
    var op_trgt = {dropTarge:true};
    i2b2.sdx.Master.AttachType("ExampPDO-CONCPTDROP", "CONCPT", op_trgt);

    i2b2.sdx.Master.AttachType("ExampPDO-PRSDROP", "PRS", op_trgt);
}
```

Building the Drop Event Handlers

The **SDX Drop messages** need to be processed differently depending on the associated data type requiring two separate event handlers to be created in the plug-in's main namespace:

```
i2b2.ExampPDO.prsDropped = function (sdxData) {
    sdxData = sdxData[0]; // only interested in first record

    // save the info to our local data model
    i2b2.ExampPDO.model.prsRecord = sdxData;

    // let user know the drop was successful by displaying name of the patient set
    $("ExampPDO-PRSDROP").innerHTML = i2b2.h.Escape(sdxData.sdxInfo.sdxDisplayName) ;


    // change background temporarily to give feedback of a successful drop
    $("ExampPDO-PRSDROP").style.background = "#CFB";
    setTimeout("$('ExampPDO-PRSDROP'). style.background = '#DEEBEF'", 250);

    // prevent requerying the Hive for results if the input dataset has not changed
    i2b2.ExampPDO.model.dirtyResultsData = true;
};
```

```
i2b2.ExampPDO.conceptDropped = function (sdxData) {
    sdxData = sdxData[0]; // only interested in first record

    // save the info to our local data model
    i2b2.ExampPDO.model.conceptRecord = sdxData;

    // let user know the drop was successful by displaying name of the patient set
    $("ExampPDO-CONCPTDROP").innerHTML = i2b2.h.Escape(sdxData.sdxInfo.sdxDisplayName) ;


    // change background temporarily to give feedback of a successful drop
    $("ExampPDO- CONCPTDROP").style.background = "#CFB";
    setTimeout("$('ExampPDO- CONCPTDROP). style.background = '#DEEBEF'", 250);

    // prevent requerying the Hive for results if the input dataset has not changed
    i2b2.ExampPDO.model.dirtyResultsData = true;
};
```

Connecting the Drop Event Handlers

Once the drop event handlers were created they were registered to the proper **Drop Target DIVs** inside the plug-in's initialization code:

```
i2b2.ExampPDO.Init = function (loadedDiv) {
    // Manage YUI Tabs
    this.yuiTabs = new YAHOO.widget.TabView("ExampPDO-TABS", {activeIndex : 0});

    // register the drop target DIVs
    var op_trgt = {dropTarge:true};
    i2b2.sdx.Master.AttachType("ExampPDO-CONCPTDROP", "CONCPT", op_trgt);
    i2b2.sdx.Master.AttachType("ExampPDO-PRSDROP", "PRS", op_trgt);

    // drop event handlers used by this plugin
    i2b2.sdx.Master.setHandlerCustom("ExampPDO-CONCPTDROP", "CONCPT", "DropHandler", i2b2.ExampPDO.conceptDropped);

    i2b2.sdx.Master.setHandlerCustom("ExampPDO-PRSDROP", "PRS", "DropHandler", i2b2.ExampPDO.prsDropped);
}
```

Implementing the Output Options Functionality

To manage the user selecting one or more different types of output an **outputOptions** configuration namespace was created within the plug-in's data model namespace (**i2b2.ExampPDO.model**). This set of namespace extensions need to be setup before any other code attempts to access them and as such will be placed at the beginning of the plug-in's initialization code like so:

```
i2b2.ExampPDO.Init = function (loadedDiv) {
    // Manage YUI Tabs
    this.yuiTabs = new YAHOO.widget.TabView("ExampPDO-TABS", {activeIndex : 0});

    // set default output options
    i2b2.ExampPDO.model.outputOptions = {};
    i2b2.ExampPDO.model.outputOptions.patients = true;
    i2b2.ExampPDO.model.outputOptions.events = true;
    i2b2.ExampPDO.model.outputOptions.observations = true;

    // register the drop target DIVs
    var op_trgt = {dropTarge:true};
    i2b2.sdx.Master.AttachType("ExampPDO-CONCPTDROP", "CONCPT", op_trgt);
    i2b2.sdx.Master.AttachType("ExampPDO-PRSDROP", "PRS", op_trgt);

    // drop event handlers used by this plugin
    i2b2.sdx.Master.setHandlerCustom("ExampPDO-CONCPTDROP", "CONCPT", "DropHandler", i2b2.ExampPDO.conceptDropped);

    i2b2.sdx.Master.setHandlerCustom("ExampPDO-PRSDROP", "PRS", "DropHandler", i2b2.ExampPDO.prsDropped);
}
```

To update the information in the **outputOptions** data model namespace we needed to connect the user's interaction with the output checkbox group with a function that performs the changes to the output options. First, a function was created and added to the plug-in's base namespace:

```
i2b2.ExampPDO.chgOutputOption = function (ckBox,option) {
    // This function updates the data model object that controls the output format

    // requested during the PDO call
    i2b2.ExampPDO.model.outputOptions[option] = ckBox.checked;
    i2b2.ExampPDO.model.dirtyResultsData = true;
};
```

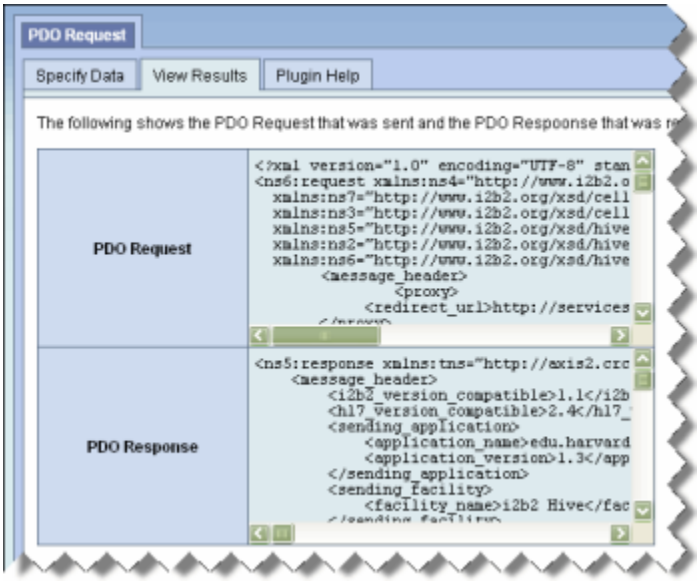
The HTML checkbox elements are now updated and connected to the created function via an **onChange** event handler. The HTML updates are as so:

```
<div class="ExampPDO-MainContent">
    <div class="ExampPDO-MainContentPad">
    -
    -
    -
    <div class="outputOptionslbl">Return:</div>
    <div class="outputOptions">
        <form>
            <span>
                <input type="checkbox" checked id="ExampPDO-OutputPatient"
                    onChange="i2b2.ExampPDO.chgOutputOption(this,'patients');" >
                    Patients
            </span>
            <span>
                <input type="checkbox" checked id="ExampPDO-OutputEvents"
                    onChange="i2b2.ExampPDO.chgOutputOption(this,'events');" >
                    Events
            </span>
            <span>
                <input type="checkbox" checked id="ExampPDO-OutputObservations"
                    onChange="i2b2.ExampPDO.chgOutputOption(this,'observations');" >

                    Observations
            </span>
        </form>
    </div>
    </div>
</div>
```

HTML for Processing Output

The output screen for this example plug-in consists of two boxes that display the following:

1. The XML request **sent to** the PDO function
2. The XML **returned from** the PDO request.



```
<div id="ExampTabs-TABS" class="yui-navset">
<ul class="yui-nav"> . . . </ul>
<div class="yui-content" id="ExampPDO-CONTENT">
<div> [Tab #1's HTML form . . .] </div>

<div>
<div class="ExampPDO-MainContent">
<div class="results-directions">
Please return to the "Specify Data" tab and select both a Patient Set and a Concept.
</div>
<div class="results-working" style="display:none;">
The PDO request is processing...
</div>
<div class="results-finished" style="display:none;">
<div class="results-text">
The following shows the PDO Request that was sent and the PDO Response that was
received:
</div>
<div id="ExampTabs-InfoPDO">
<table>
<tr>
<th>PDO Request</th>
<td class="InfoPDO-Request">
<div class="originalXML"></div>
</td>
</tr>
<tr>
<th>PDO Response</th>
<td class="InfoPDO-Response">
<div class="originalXML"></div>
</td>
</tr>
</table>
</div>
</div>
</div>
</div>
</div>
</div>
```

Processing via PDO Request

The plug-in's processing routine builds a PDO request message and then sends it. Once the AJAX call returns with the response the routine then displays the message request and message response text in the output tab.
The function **getResults()** was added to the plug-in's base namespace which first checks to see if the dirty data flag is set before extracting information from the Ontology concept that was previously saved.

```
i2b2.ExampPDO.getResults = function () {
if (i2b2.ExampPDO.model.dirtyResultsData) {
// translate the concept XML for injection as PDO item XML
var t = i2b2.ExampPDO.model.conceptRecord.origData.xmlOrig;

var cdata = {};
cdata.level = i2b2.h.getXNodeVal(t, "level");
cdata.key = i2b2.h.getXNodeVal(t, "key");
cdata.tablename = i2b2.h.getXNodeVal(t, "tablename");
cdata.dimcode = i2b2.h.getXNodeVal(t, "dimcode");
cdata.synonym = i2b2.h.getXNodeVal(t, "synonym_cd");
```

The routine then checks the **outputOptions** configuration namespace and generates the output section of the request message.

```
var output_options = ' ';
if (i2b2.ExampPDO.model.outputOptions.patients) {
output_options += '<patient_set select="using_input_list" onlykeys="false"/>\n>';

}
if (i2b2.ExampPDO.model.outputOptions.events) {
output_options += '<event_set select="using_input_list" onlykeys="false"/>\n>';
}
if (i2b2.ExampPDO.model.outputOptions.observations) {
output_options += '<observation_set blob="false" onlykeys="false"/>\n>';
}
```

Finally, the query's main filter message is built using data from the saved *Patient Record Set* and the values extracted from the saved Ontology concept.

```
var msg_filter = <input_list>
<patient_list max="6" min="1">\n ' // <--- only the first 5 records

<patient_set_coll_id>
i2b2.ExampPDO.model.prsRecord.sdxInfo.sdxKeyValue
</patient_set_coll_id>
</patient_list>
</input_list>
<filter_list>
<panel_name="cdata.key">
<panel_number>0</panel_number>
<panel_accuracty_scale>0</panel_accuracy_scale>
<invert>0</invert>
<item>
<hlevel>cdata.level</hlevel>
<item.key>cdata.key</item.key>
<dim_tablename>cdata.tablename</dim_tablename>
<dim_dimcode>cdata.dimcode</dim_dimcode>
<item_is_synonym>cdata.synonym</item_is_synonym>
</item>
</panel>
</filter_list>
<output_option>
output_options
</output_option>
```

Once the XML of the filter message has been created a "**Scoped-callback**" **object** is created.

- ***All interactions with the Core Cell Communicators utilize the scoped-callback object.***

The scoped callback object consists of two parts:

1. A callback function that contains code that will be executed when the AJAX call returns with the result from the cell server

1. A JavaScript scope identifier which sets the context that the callback function is executed in (what the value of *this* keyword will be set to.

```
// callback processor
var scopedCallback = new i2b2_scopedCallback();
scopedCallback.scope = this;
scopedCallback.callback = function(results) {
// THIS function is used to process the AJAX results of the Cell communicator call
// results data object contains the following attributes:
// refXML: xmlDomObject <--- for data processing
// msgRequest: xml (string)
// msgResponse: xml (string)
// error: boolean
// errorStatus: string [only with error=true]
// errorMsg: string [only with error=true]

// check for errors
if (results.error) {
alert('The results from the server could not be understood. Press F12 for more information. ');
console.error("Bad Results from Cell Communicator: ",results);
return false;
}

// Remove the waiting status line and show results from AJAX call to Cell
$$("DIV#ExampPDO-mainDiv DIV#ExampPDO-TABS DIV.results-working")[0].hide();
$$("DIV#ExampPDO-mainDiv DIV#ExampPDO-TABS DIV.results-finished")[0].show();
var divResults = $$("DIV#ExampPDO-mainDiv DIV#ExampPDO-InfoPDO")[0];
Element.select(divResults, '.InfoPDO-Request .originalXML')[0].innerHTML = '<pre>'+i2b2.h.Escape(results.msgResponse)+ '</pre>';


// optimization – only requery when the input data is changed ("Dirty Data")
i2b2.ExampPDO.model.dirtyResultsData = false;
}
```

Once a scoped callback object is ready, the GUI is updated to provide visual feedback to the user that something is happening. Finally, the request is sent to the CRC using the core CRC Communicator object (namespace **i2b2**.**CRC**.**ajax**).

```
// Remove the waiting status line and show results from AJAX call to Cell
$$("DIV#ExampPDO-mainDiv DIV#ExampPDO-TABS DIV.results-directions")[0].hide();
$$("DIV#ExampPDO-mainDiv DIV#ExampPDO-TABS DIV.results-finished")[0].hide ();
$$("DIV#ExampPDO-mainDiv DIV#ExampPDO-TABS DIV.results-working")[0]. show();

// AJAX CALL USING THE EXISTING CRC CELL COMMUNICATOR
var msg_vals = {patient_limit: 5, PDO_Request: msg_filter};
i2b2.CRC.ajax.getPDO_fromInputList("Plugin:ExampPDO", msg_vals, scopedCallback);

}
}
```

Connecting Processing to Tab Change

The event which starts the processing of input data is the user switching to the "*Results*" *tab* after entering or changing the input data. The code is set during the plug-in's initialization routine and is simply capturing the **tabChange** event in the YUI tabs object.

```
i2b2.ExampPDO.Init = function (loadedDiv) {
// Manage YUI Tabs
this.yuiTabs = new YAHOO.widget.TabView("ExampPDO-TABS", {activeIndex : 0});

// set default output options
i2b2.ExampPDO.model.outputOptions = {};
i2b2.ExampPDO.model.outputOptions.patients = true;
i2b2.ExampPDO.model.outputOptions.events = true;
i2b2.ExampPDO.model.outputOptions.observations = true;

// register the drop target DIVs
var op_trgt = {dropTarge:true};
i2b2.sdx.Master.AttachType("ExampPDO-CONCPTDROP", "CONCPT", op_trgt);
i2b2.sdx.Master.AttachType("ExampPDO-PRSDROP", "PRS", op_trgt);

// drop event handlers used by this plugin
i2b2.sdx.Master.setHandlerCustom("ExampPDO-CONCPTDROP", "CONCPT", "DropHandler", i2b2.ExampPDO.conceptDropped);

i2b2.sdx.Master.setHandlerCustom("ExampPDO-PRSDROP", "PRS", "DropHandler", i2b2.ExampPDO.prsDropped);

// execute PDO call when YUI tabs changing
this.yuiTabs.on('activeTabChange', function(ev) {
// Tabs have changed
if (ev.newValue.get('id')=="ExampPDO-TAB1") {
// user switched to Results tab
if (i2b2.ExampPDO.model.conceptRecord && i2b2.ExampPDO.model.prsRecord) {

// contact PDO only if we have data
if (i2b2.ExampPDO.model.dirtyResultsData) {
// recalculate the results only if the input data has changed
i2b2.ExampPDO.getResults();
}
}
}
});
};
```

Clear Plug-in Data on Unload

It is good practice to clear the data from the plug-in once it has been unloaded by the *Plugin Viewer* framework by adding the following:

```
i2b2.ExampPDO.Unload = function () {
// purge old data
i2b2.ExampPDO.model.prsRecord = false;
i2b2.ExampPDO.model.conceptRecord = false;
i2b2.ExampPDO.model.dirtyResultsData = true;
i2b2.ExampPDO.model.outputOptions.patients = true;
i2b2.ExampPDO.model.outputOptions.events = true;
i2b2.ExampPDO.model.outputOptions.observations = true;

return true;
};
```