



@task

Flash Example Explanation

- The following document discusses the layout and reasoning of the Flash API integration example.
- It includes some helpful hints and suggestions for users desiring to utilize the @task EJB API.

@TASK FLASH EXAMPLE

SUMMARY

This simple example illustrates how to access the @task API from Adobe Flash.

PROCESS

1. **Determine which APIs you need to access by referencing the @task 4.0 SDK.** Make note of the parameters the API calls require, specifically the class path if it is a complex data type. For our example we will remember the calls found in Table 1.1 below.

TABLE 1.1: API CALLS TABLE

API Call	ClassPath
getTasksFromSearch	java.lang.String, com.attask.beans.api.NameValueBean
editTask	java.lang.String, com.attask.beans.api.TaskBean

2. **Write a class that inherits from the APIAccess Class.** In this class you will write functions that request data from or submit data to the @task API. APIAccess handles the basics of flash remoting; it permits logging in and will track the sessionID for you. In our example we created the TaskAPIAccess class which contains the functions found in Table 1.2.

TABLE 1.2: FUNCTIONS OF THE TASKAPIACCESS

Function	Calls @task API
getTasksFromSearch	getTasksFromSearch
editTask	editTask
editTask_Result	*N/A
processTaskResult	*N/A

* These functions do not call the API, but are executed when a response is received from the API.

When we submit an object to the API we need to accurately mimic the properties of that object. In our example when we edit a task we simply modify the name and description properties of a task object previously returned to us from our search and send that object (which contains all the expected properties of a TaskBean) back to the API in editTask().

Take a look at our editTask function:

```
function editTask( ASObj )
{
    this.serviceObject.editTask(this.sessionID, translateToJavaObj( ASObj,
        "com.attask.beans.api.TaskBean" ) );
}
```

You will notice within the serviceObject.editTask API call translateToJavaObj() is called. This is because we need to translate the data object we are sending (a generic hashmap) to an object that is registered with the appropriate class path IN JAVA. This is why we must remember the classPath for the parameters of our API calls. Since our first parameter is a simple string (not a complex data type) we need not worry, however since our second parameter is com.attask.beans.api.TaskBean, we must first translate to that data type by calling translateToJavaObj(ASObj, "com.attask.beans.api.TaskBean").

3. Implement our class in our flash movie.

Include your custom class:

```
import mx.attask.TaskAPIAccess;
```

To create a new instance of our class, a gatewayURL is required for a connection to be established, but a sessionID is optional (but helpful to prevent unnecessary logins if we have one):

```
taskAPI = new TaskAPIAccess(gatewayURL, sessionID);
```

If we do not have a sessionID to pass in, we must log in using the following call:

```
taskAPI.login(userName, userPass);
```

At this point, if all went well, we should have established a connection with the api. Now we can make our API calls

- ### 4. Make an API call.
- In our example we perform a search for tasks that are assigned to a specific user ID. Since the API call requires a nameValueBean we need to build one within flash. We do so by creating an array and pushing objects with a name and value assignment onto it. We then pass the object to our API, and wait for a callback.

```
var searchMap = new Array();
searchMap.push({ name:"assignedToID", value: userID.toString() });
taskAPI.getTasksFromSearch(searchMap); // request tasks from the API and pass
in the search map
```

- ### 5. API Responses.
- If an API has a return type, you can be notified by listening for it. Since we have requested a list of tasks, we obviously want to be notified when our results come in. Once we receive the results, we are able to update our UI with the data.

```
taskListener.taskResult = function(evtObj:Object):Void
{
    //in this example we are only expecting a list of tasks as a response, so
we'll store them in the tasks array.
    tasks = evtObj.data;

    //loop through the tasks and add them to our list component on the stage
    for (var i = 0; i < tasks.length; i++)
    {
        taskList_mc.addItem({label:tasks[i].name, taskID:tasks[i].ID, taskIn-
dex:i});
    }

    //this listener handles change events to our list component (when a new item
is selected)
    var listListener:Object = new Object();
    listListener.change = function(evt_obj:Object)
    {
        //when a new item is selected we populate some text boxes with the appro-
pate task's information

        var taskIndex = evt_obj.target.selectedItem.taskIndex; //we track the tas-
kIndex inside the tasks array for reference
        taskDetails_mc.txt_taskName.text = tasks[taskIndex].name;
        taskDetails_mc.txt_taskDescription.text = tasks[taskIndex].description ==
undefined ? "" : tasks[taskIndex].description;
        showEditComponents( true );
    };

    taskList_mc.addEventListener("change", listListener);
};
```