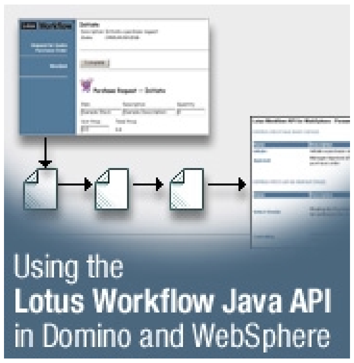


Level: Advanced
Works with: Lotus Workflow, Domino, WebSphere
Updated: 02-Dec-2002

by JoAnn Jordan
and Seol Young Park

Lotus Workflow offers more than 50 LotusScript API functions to provide developers a basic set of functionality for building customized applications. In addition, Lotus Workflow 3.0 introduced JavaScript and Java APIs to help support Web applications.

You can use the Java API to integrate Lotus Workflow beyond its "out of the box" capabilities, for example, running workflow processes on a WebSphere server.

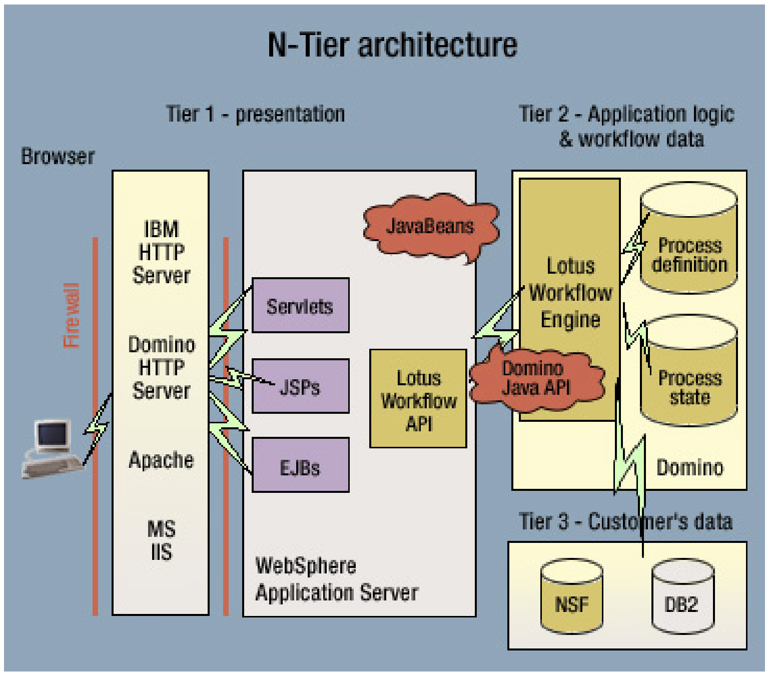
This article describes the Lotus Workflow Java API. We begin with a brief introduction to its functions, then explain how to configure Domino and WebSphere to run them. We conclude by explaining how you can use our Lotus Workflow Java API demo with the sample application that ships with Lotus Workflow to create your own applications. This demo includes a custom Purchase Order process that incorporates the Lotus Workflow Java API. This process can run in both Domino and WebSphere. (Our demo replaces the previous LWF Java API demo.) You can download our demo from the **Sandbox**.

This article assumes that you're an experienced Java programmer and are familiar with Lotus Workflow features and terminology. We also assume that you are familiar with the Domino Designer and WebSphere environments, especially WebSphere Application Server and WebSphere Studio Application Developer.

What is the Lotus Workflow Java API?

The Lotus Workflow Java API is an application programming interface you can use to build JavaServer Pages (JSPs) and servlets. This API lets you easily integrate people-centric processes into e-commerce applications. The objects in the Lotus Workflow API are almost identical to the objects found in Lotus Workflow.

The Java API supports n-tier architecture for e-business solutions. (*N-tier architecture* is a programming technique that extends the traditional client/server 2-tier architecture. For example, an application can use a 3-tier architecture in which different parts of its code run on the client, the application server, and the data server.) The following diagram shows how the Java API fits into n-tier architecture. The presentation layer (JSPs and servlets) is shown outside the firewall; within the firewall are the application logic (including the process definition) and Workflow data (residing in Domino, DB2, and other database management systems).



The Lotus Workflow Java API is implemented as a set of Java Beans designed to run on multiple platforms, including the WebSphere Application Server (WAS). It can be considered a wrapper on top of Lotus Workflow and Java beans that call Notes agents, and it can access views and fields in Notes documents. With the Lotus Workflow Java API for WebSphere, you can combine the strengths of IBM WebSphere and Lotus Domino using servlets and JSPs. You can also access Domino and Lotus Workflow services to extend workflow functionality to areas such as organization information (via the Domino Directory and Lotus Workflow Organization Directory) and automation (including mail notification).

The Lotus Workflow Java API is based on the **Object Management Group** (OMG) standards Lotus has mapped to Java interfaces. Where necessary, Lotus has extended the OMG standard to enhance functionality. Extended objects use the naming convention, *ExtendedXXXX*, for example, ExtendedDocument object. (For more information, refer to the **OMG Workflow Management Facility Specification**.)

The objects in the Lotus Workflow Java API are almost identical to the objects found in Lotus Workflow. All Lotus Workflow API objects adhere to the OMG specification naming conventions where possible. Wherever objects needed to be extended, they were extended and given the names *ExtendedXXX*. The following table outlines this relationship:

OMG Terminology	**Lotus Workflow Terminology**
ExtendedActivityAccessBean	LWF ActivityInstance
ExtendedProcessAccessBean	LWF Job
ExecutionObjectAccessBean	No equivalent object in LWF
ProcessManagerAccessBean	LWF Process
WorklistHandlerAccessBean	LWF WorklistHandler
BinderAccessBean	LWF Binder
ExtendedDocumentAccessBean	LWF Document
ResourceAccessBean	LWF User
AssignmentAccessBean	No object in LWF for this object

For more information, refer to the **Lotus Workflow Application Programming Interface (API) Reference**.

Configuring Domino and WebSphere for the Lotus Workflow Java API

The Lotus Workflow Java APIs take advantage of the document management capabilities of Domino, as well as the scalability of the WebSphere Application Server. To deploy the Workflow Java APIs, you must first install both Domino and WebSphere and configure Single Sign On (SSO) for both. You must also enable DIIOP in Domino. This section explains how to do this and offers links to more detailed information.

Our example has been configured and tested to work with:

- WebSphere Studio Application Developer 4.0.x
- Domino 5.0.8/5.0.9a
- Lotus Workflow v3.0a/v3.0.1

Later releases of these products should also work.

Single Sign On (SSO)

The Lotus Workflow Java APIs use an LTPA token generated by authenticating with Single Sign On (SSO). LTPA (Lightweight Third Party Authentication) is an IBM technology that uses tokens to represent users' credentials. (See the **IBM WebSphere Security Overview** for a description of LTPA.) SSO works by placing the LTPA token on the user's PC after the user successfully authenticates. Other servlets, JSPs, and Notes databases can then use this token to obtain the user's credentials. This allows a Web user to move from one application to another without being prompted repeatedly for user name and password.

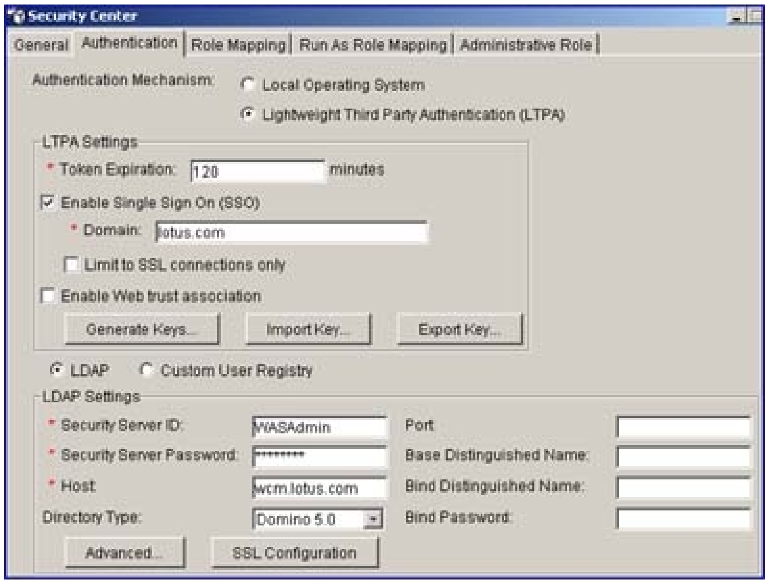
The requirements for enabling SSO are:

- The same registry (LDAP server) must be used for authentication.
- All SSO participating servers must be in the same DNS domain.
- The URLs must include the DNS domain (no IP addresses or host names).
- The browsers must be configured to accept cookies.
- Servers’ time and time zone must be correct (SSO token expiration time is absolute).
- All servers must share the LTPA keys.

In the following sections, we describe how to enable SSO in both WebSphere Advanced Server and Domino. For more information about WebSphere/Domino security and SSO, consult the IBM Redbook **Domino and WebSphere Together**.

Configuring SSO in WebSphere

SSO in the WebSphere Application Server is configured from the WAS Administrator's Console in the Security Center:



Authentication Mechanism: Local Operating System [✓ Lightweight Third Party Authentication (LTPA)]

LTPA Settings

* Token Expiration: 120 minutes
- [✓] Enable Single Sign On (SSO)
* Domain: lotus.com
- [ ] Limit to SSL connections only
- [ ] Enable Web trust association

[✓ LDAP] Custom User Registry

LDAP Settings

* Security Server ID:	WASAdmin	Port:	
* Security Server Password:	*********	Base Distinguished Name:	
* Host:	wcm.lotus.com	Bind Distinguished Name:	
Directory Type:	Domino 5.0	Bind Password:	

Before beginning the following procedure to configure SSO in WebSphere, make sure both Domino and the LDAP task are running. You also need to specify information about your Domino server and generate an LTPA key which you'll need for setting up SSO in Domino.

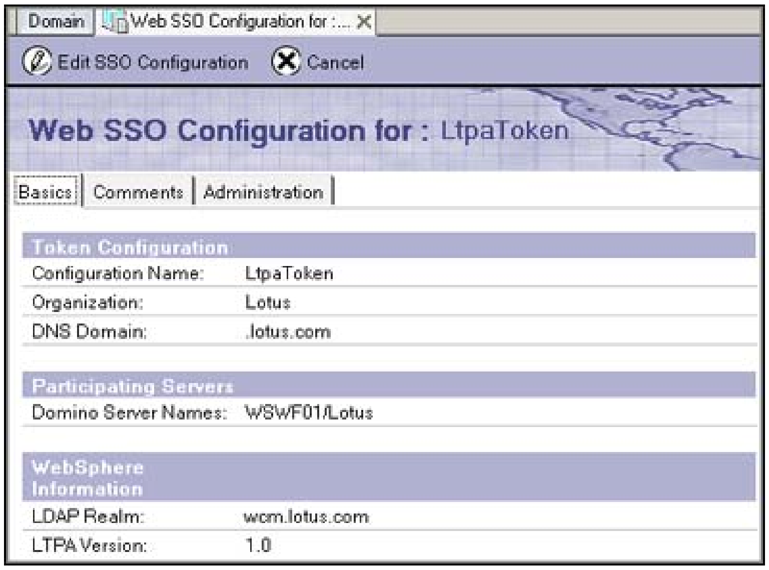
1. Start the IBM WS AdminServer service, if it isn't already running.
2. Launch the WebSphere Administrator's console.
3. Choose Console - Security Center. The Security Center appears (see preceding illustration).
4. Click the General tab, and click Enable Security.
5. Click the Authentication tab.
6. Click the Lightweight Third Party Authentication (LTPA) radio button, then click Enable Single Sign On (SSO).
7. In the Domain field, enter the DNS domain name.
8. Go to the LDAP Settings section. In Security Server ID and Security Server Password fields, enter the LDAP user name and password respectively.
9. In the Host and Directory Type fields, enter the host name and directory type of the Domino server.
10. Click the Generate Keys button. When prompted, enter a password for the key.
11. Click the Export Key button. When prompted, specify a disk or location to save the keys.
12. Exit the Security Center, and restart IBM WS AdminServer.
13. Verify that you can start the Admin console with the new user name and password.

Configuring SSO in Domino

Follow these steps to configure SSO in Domino:

1. In the Domino Directory, open the Server document for the Domino server you want to configure.
2. Select Web - Create Web SSO Configuration from the action bar. (In Domino 6, select Create Web (R5) - SSO Configuration.)
3. Enter the DNS domain in the DNS Domain field.
4. In the Participating Servers section, select the Domino servers that will participate in SSO.
5. Click Keys, then click Import WebSphere LTPA Keys.
6. Enter the location of the key file saved when the keys were exported from WebSphere (as described in the preceding section).
7. Enter the password specified when the keys were generated in WebSphere.
8. Click the Internet Protocols - Domino Web Engine tabs.
9. Set the Session authentication field to Multiple Servers (SSO).
10. Enter "LTPA Token" in the Web SSO Configuration field.
11. Save and close the Server document.
12. Restart the Domino HTTP server using the tell http restart command.
13. Verify that you see the message in the server console, Successfully loaded Web SSO Configuration.

Your SSO Configuration document should look as follows:



Web SSO Configuration for : LtpaToken

Token Configuration	
Configuration Name:	LtpaToken
Organization:	Lotus
DNS Domain:	.lotus.com
Participating Servers	
Domino Server Names:	WSWF01/Lotus
WebSphere Information	
LDAP Realm:	wcm.lotus.com
LTPA Version:	1.0

Verify that your SSO is configured correctly and working by opening a secure site in both WebSphere and Domino. You should be prompted only once for your user credentials.

Configuring the DIIOP task in Domino

The Java APIs require that the DIIOP task be running on Domino and that users initiating the workflow processes have rights to run restricted Java agents. These steps are described below:

1. At the Domino server console, load the DIIOP task by typing load diiop. (You can also start this task automatically by adding diiop to the ServerTasks variable in the server's Notes.ini file.)
2. In the Domino Directory, open the server's Server document. In the Security tab, add an asterisk (*) to the "Run restricted Java/JavaScript/COM" field. This allows all your users to run restricted Java agents.
3. Restart the Domino server for these changes to take effect.

Configuring the Lotus Workflow Java API demo

Our Lotus Workflow Java API demo is based on the Purchase Order process that comes with Lotus Workflow. We customized this process with the Lotus Workflow Java API, to allow it to run on both Domino and WebSphere. The process has all the features as the standard Purchase Order, although we have customized the UI to ensure it functions consistently on all platforms.

The Lotus Workflow Java API demo (LWFJavaAPIDemo.zip) can be downloaded from the **Sandbox**. This demo consists of two files:

- LWFJavaAPIDemo.nsf is the Java API Demo Application database.
- LWFJavaAPIDemo.ear is the sample Enterprise Archive file. You must load this onto the WebSphere server.

Configuring the Request for Quote process (optional)

In addition to the Purchase Order process, our demo includes the forms and the process for the Request for Quote (RFQ) process from the previous Java API demo. If you want to run the RFQ process, you must copy the necessary forms from our Lotus Workflow Java API demo, and paste them into the Lotus Workflow Sample application database that comes with the product. To do this, perform the following steps. (If you only want to use our sample Purchase Order process, skip this procedure.)

1. Open the LWFJavaAPIDemo.nsf database in Domino Designer.
2. Copy and paste all the forms in the database into the Lotus Workflow Sample application database.
3. Close Domino Designer.
4. Go to the "About this database" page in LWFJavaAPIDemo.nsf, click Request for Quote and detach this process.
5. Open the Lotus Workflow Architect.
6. Choose File - Import.
7. Choose File - Active Process.
8. Open the Lotus Workflow Sample application database.
9. Choose Administrator - Cache.
10. Click the update Cache action button.
11. Close and reopen the database.

Before continuing, verify that you can start both the Purchase Order and RFQ processes in the Start New Job box.

Importing LWFJavaAPIDemo.ear into WebSphere Application Server

After you extract the Enterprise Application file LWFJavaAPIDemo.ear from LWFJavaAPIDemo.zip, you must import it into your WebSphere Application Server (WAS). To do this:

1. Start the WAS Administration console.
2. Choose Console - Wizard - Install Enterprise Application.
3. Choose Install Application (*.ear).
4. Click the Browse button to choose the LWFJavaAPIDemo.ear file.
5. Specify the name of your application.
6. Click Next through the remaining screens until you see the prompt "Selecting Virtual Hosts for Web Modules." Verify that your host name is selected.
7. In the Selecting Application Servers, verify that your application server is selected.
8. Click Next and then Finish.
9. Right-click on your node and select Regen Webserver Plugin.
10. Start your application server.
11. Start your browser.

After you install LWFJavaAPIDemo.ear on your WAS server, go to a secure database (for example, the Domino Directory database on your Domino server). This authenticates your first SSO token. Then go to http://<yourserver.xxx.yyy>/Demo/Pages/home.html to start the demo.

Using our Java API demo code

In this section, we use WebSphere Studio Application Developer (WSAD) to run and examine our demo code.

This will help you understand how our code works, so you can customize it to your own needs. Let's start by looking inside our demo. The flow is simple:

- All user requests are passed by URL (from our UI written in JSPs/HTML) to the CommandServlet.
- The CommandServlet gets the parameters of the name/value pairs and processes all requests.
- The Constants class contains all constants used by the CommandServlet to process each request.
- The CustomConstants class contains constants that need to be customized to run the demo in your environment.

We will examine the custom codes step-by-step. But first, you must import LWFJavaAPIDemo.ear into WSAD:

1. Start WSAD.
2. Choose File - Import.
3. Select the EAR file from the displayed list of file types, then click Next.
4. Browse to LWFJavaAPIDemo.ear.
5. Specify your Enterprise Application project name, and then click Finish.

The remainder of this section is presented as a series of seven main steps:

1. Modify the CustomConstants.java file.
2. Generate the LTPA token for SSO and run the application.
3. Creating a Purchase Order.
4. Define the JSPs to display.
5. View the workflow activities.
6. Open an activity.
7. Complete or claim the activities.

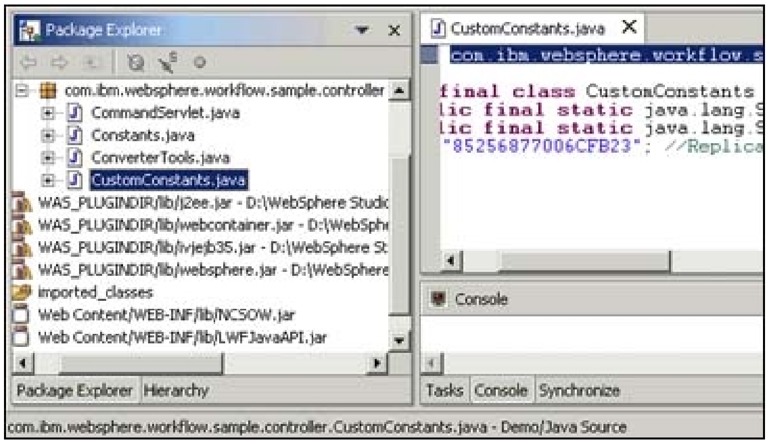
Each of these steps has its own subsection.

Step1: Modify the CustomConstants.java file
Before you can use our demo, you must edit its CustomConstants.java file to include the server name and the replica ID of your Lotus Workflow application database. (By default, the replica ID is the same as the Lotus Workflow Sample Application database.) To do this:

1. Go to Java perspective in WSAD.
2. Browse to <your application>/source/com.ibm.websphere.workflow.sample.controller/CustomConstants.java. (See the following screen.)
3. Open the file and add the following lines.

   ```
   public final static java.lang.String DOMINO_SERVER_NAME =<your_server_name>;
   public final static java.lang.String REPLICA_ID_APPLICATION_DATABASE
   =<Replica_ID_of_Sample_Database>;
   ```

   Where <your_server_name> is the name of the server on which your application resides, and <Replica_ID_of_Sample_Database> is the replica ID of your application database.
4. Save and close the file.



```
com.ibm.websphere.workflow.sample.controller
   CommandServlet.java
   Constants.java
   ConverterTools.java
   CustomConstants.java
WAS_PLUGINDIR/lib/j2ee.jar - D:\WebSphere Studio
WAS_PLUGINDIR/lib/webcontainer.jar - D:\WebSphe
WAS_PLUGINDIR/lib/ivjejb35.jar - D:\WebSphere St
WAS_PLUGINDIR/lib/websphere.jar - D:\WebSphere
imported_classes
Web Content/WEB-INF/lib/NCSOW.jar
Web Content/WEB-INF/lib/LWFJavaAPI.jar
```

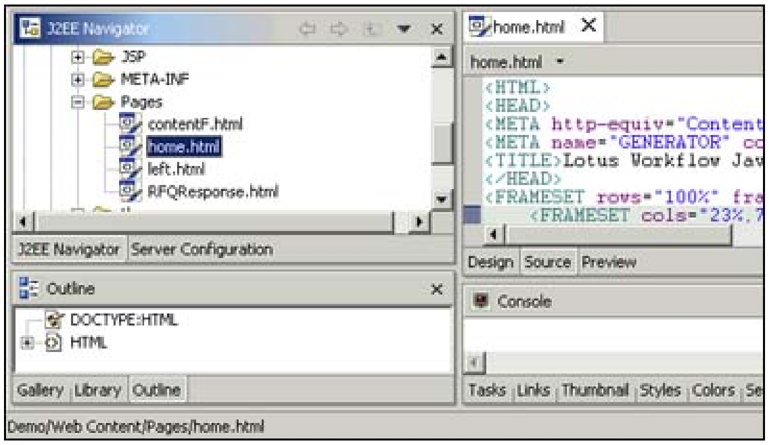
```
 com.ibm.websphere.workflow.s
final class CustomConstants
lic final static java.lang.S
lic final static java.lang.S
"85256877006CFB23"; //Replica
```

com.ibm.websphere.workflow.sample.controller.CustomConstants.java - Demo/Java Source

Step 2: Generate the LTPA token for SSO and run the application
WSAD comes equipped with a full WebSphere test environment. However, this test environment does not support

SSO and therefore, can't generate LTPA keys or tokens. To do this, you must first navigate to a secure site. This creates an LTPA token on your machine, which the Java APIs need for authentication. To do this:

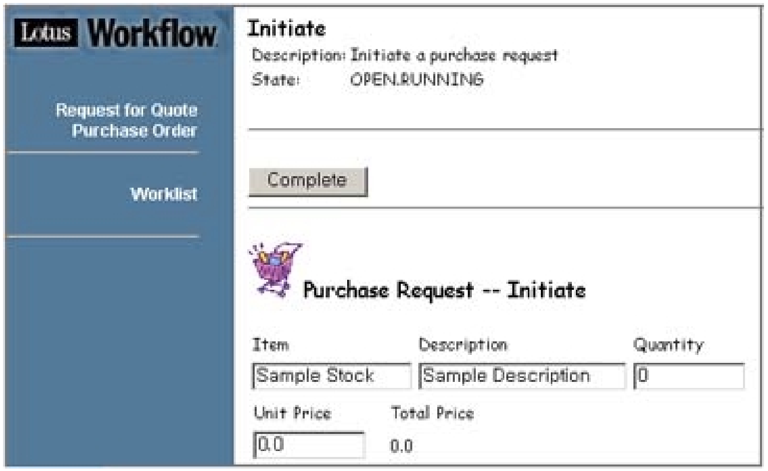
1. Go to Web perspective in WSAD.
2. Browse to <your application>/webApplication/Pages/Home.html. (See the subsequent screen.)
3. Click Run on Server to open your browser on the local host URL.
4. Browse to any secure database on your Domino server, for example names.nsf.
5. After you authenticate (which creates the LTPA token), browse to home.html in your application (for example, http://myserver.xxx.yyy:8080/Demo/Pages/home.html). Note the port number is 8080; this is the port your test sever is listening to. Your application now starts on WebSphere.



```
<HTML>
<HEAD>
<META http-equiv="Content
<META name="GENERATOR" co
<TITLE>Lotus Workflow Jav
</HEAD>
<FRAMESET rows="100%" fra
    <FRAMESET cols="23%,7
```

Step 3: Creating a Purchase Order

Now that you (finally) have your demo application up and running, let's take a look at it in action. To begin, click Purchase Order. This displays the following screen:



Initiate

Description: Initiate a purchase request
State: OPEN.RUNNING

Purchase Request -- Initiate

Item	Description	Quantity
Sample Stock	Sample Description	0

Unit Price	Total Price
0.0	0.0

Internally, clicking Purchase Order starts CommandServlet, using the following line of code:

<A href="/Demo/CommandServlet?COMMAND=POCREATESTARTCLAIM&NEWPROCESSNAME=Purchase Order">Purchase Order</A>

This code passes parameters to CommandServlet, including which command to run, and the process name with which to create a job. CommandServlet calls the PO_createAndStartProcessAndClaimFirstActivity method to create a job and to return to the page Purchase_Request.jsp. You can also pass the new name for the job created via URL by modifying the <A> link tag. For example:

<A href="/Demo/CommandServlet?COMMAND=POCREATESTARTCLAIM&NEWPROCESSNAME=Purchase Order&NEWJOBNAME=xxxxxxxx">Purchase Order</A>

If you don't explicitly specify a job name, the code creates a random job name using the current date/time:

```
public void PO_createAndStartProcessAndClaimFirstActivity(HttpServletRequest request,HttpServletResponse
response)
    {
    ..................................
    Hashtable fieldsToSave = new Hashtable();
    fieldsToSave.put("Priority", "1. High");
    fieldsToSave.put("JSPOS", "Purchase_Request.jsp");
    try {
        //Create worklist handler object and pass in Domino server name and replica ID of application database
        //Constants.getWHABKey().toString returns WorklistHandlerAccessBean Key :
        <ServerName>#<ReplicaIDOfAppDB>
        worklist = new WorklistHandlerAccessBean(new
        WorklistHandlerKey(Constants.getWHABKey().toString()), request);
        .............................
        // create new process by defining the process manager name and the process name and open the
        activity in a page
        activityBean = worklist.createAndStartProcessAndClaimFirstActivity(processManagerName,
        processName, fieldsToSave);
        request.getSession(true).setAttribute("activity", activityBean);
         BinderAccessBean binder = activityBean.getBinder();
         ExtendedDocumentAccessBean document = binder.getMainDocument();
         request.getSession(true).setAttribute("document", document);
        request.getSession(true).setAttribute("activity", activityBean);
        returnURL(request, response, "JSP/Purchase_Request.jsp");
        .............................
}
```

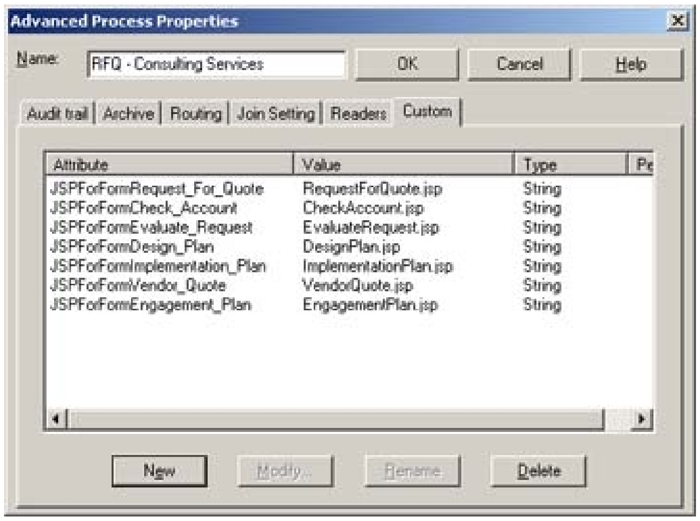
Note the fieldsToSave parameter. This parameter is a Hashtable with field names and value pairs. If it has the entry DocumentKey, its value is used as the UNID of a document in the application database, as well as the main document in the binder. The Priority argument determines the process's priority.

Step 4. Define JSPs to display
There are several ways to define which JSP to use to present the document in a page:

- You can enter the name of the JSP into the JSPOS field. To do this, either call the ExtendedDocumentAccessBean.setJSP() or create the field while creating the document as shown above.
- You can specify JSPs in predefined fields using the Lotus Workflow Architect to map between Notes forms and JSPs as Custom attributes. We use this method in our example RFQ - Consulting Services process. As you can see in the following illustration, the format of the field name is:

  ```
  JSPForForm<The_Name_Of_Your_Notes_Form>
  ```

  The values of these attributes contain the path and name of the JSP. There is a limitation, however: Field names cannot contain spaces. Use the underscore character in the form names in place of spaces.



Attribute	Value	Type
JSPForFormRequest_For_Quote	RequestForQuote.jsp	String
JSPForFormCheck_Account	CheckAccount.jsp	String
JSPForFormEvaluate_Request	EvaluateRequest.jsp	String
JSPForFormDesign_Plan	DesignPlan.jsp	String
JSPForFormImplementation_Plan	ImplementationPlan.jsp	String
JSPForFormVendor_Quote	VendorQuote.jsp	String
JSPForFormEngagement_Plan	EngagementPlan.jsp	String

- If neither method is used, the form name is used as the JSP name with an extension. For example, if the form name is Sample Form | DWFSampleForm, the JSP name would be DWFSampleForm.jsp.

With all of these methods, the order of the definitions is important. The two methods which return the JSP name, ExtendedDocumentAccessBean.getJSP(), and BinderAccessBean.getDocuments("SELECT JSP", -1), first check the JSPOS field, then the mapping, and finally the form name.

Step 5: Viewing the workflow activities
ActivityList.jsp in our demo demonstrates how to display jobs in a page. First, we get the WorklistHandlerAccessBean with the WorklistHandlerKey, which has the format <ServerName>#<ReplicaIDOfAppDB>. Then we retrieve information from the WorklistHandlerAccessBean using an SQL-like query language. The following example shows how to get either activities in progress or new activities.

```
    Syntax : SELECT Attribute1,Attribute2,... [FROM ViewName] [WHERE <@Formula>]

<%
    ..............
    WorklistHandlerAccessBean worklist = new WorklistHandlerAccessBean(new
    WorklistHandlerKey(Constants.getWHABKey().toString()),request);
    ownedActivities = worklist.getActivities("SELECT
    ACTIVITY,NAME,DESCRIPTION,STATE,OWNER,DUETIME FROM ACTIVITYOWNER", -1);
    canClaimActivities = worklist.getActivities("SELECT
    ACTIVITY,NAME,DESCRIPTION,STATE,OWNER,DUETIME FROM CANCLAIM", -1);
 %>
```

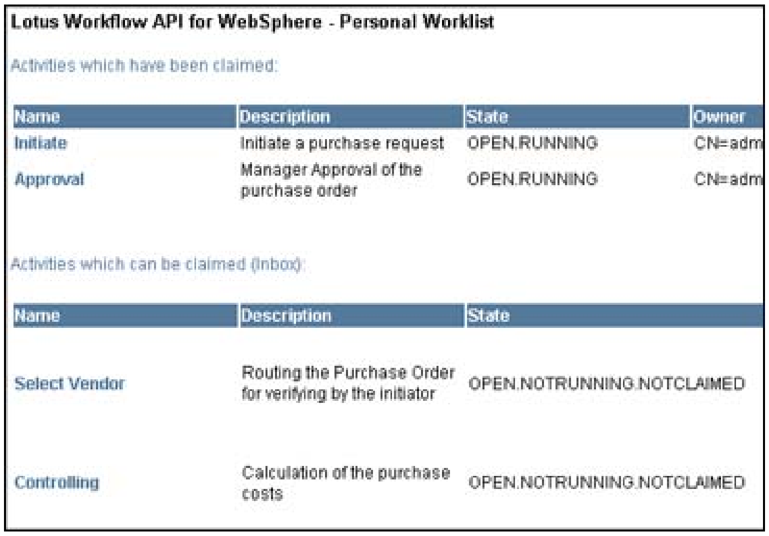
Result sets are returned as a non-null java.util.Vector, and its elements are instances of java.lang.String. If an error occurs (for example, the wrong syntax in the query statement), null is returned.

The strings have the following characteristics:
- Each string is a concatenation of attribute values of the selected attributes separated by the | character. The SELECT clause, which determines which attributes are returned, is determined in the given queryStatement.
- If an attribute has multiple values (for example, OWNER for ExtendedProcesses), the values are separated by commas.
- Date attributes (STARTTIME, DUETIME, etc.) are converted to String via java.util.Date.toString().

After we get the results, we now parse the string returned by the query shown above in the format Activity Key | Name | Description | Activity status | Activity Owner(s) | Due date. We then save this to the array Token to display. The first token (Activity Key) is used as a link to open the actual activity.

```
for (Enumeration e = ownedActivities.elements(); e.hasMoreElements();) {
        ........................
        String label = e.nextElement().toString();
        int j =label.indexOf("|");   // First token
        int k = 0;
        while( j >= 0) {
                Tokens[k] = label.substring(i,j);
                i = j + 1;
                j = label.indexOf("|", i);   // Rest of tokens
                k = k+1;
        }
        Tokens[k] = label.substring(i); // Last token
}
```

After the code runs, the following screen appears:



Lotus Workflow API for WebSphere - Personal Worklist

Activities which have been claimed:

Name	Description	State	Owner
Initiate	Initiate a purchase request	OPEN.RUNNING	CN=adm
Approval	Manager Approval of the purchase order	OPEN.RUNNING	CN=adm

Activities which can be claimed (Inbox):

Name	Description	State
Select Vendor	Routing the Purchase Order for verifying by the initiator	OPEN.NOTRUNNING.NOTCLAIMED
Controlling	Calculation of the purchase costs	OPEN.NOTRUNNING.NOTCLAIMED

Step 6: Opening an activity
The ActivityList.jsp generates a link to open each activity. For instance:

```
<A href="/Demo/CommandServlet?COMMAND=OPENACTIVITY&ACTIVITY=ExtendedActivity%23tjfdud%........."
target="contentF">Check Account</A>.
```

When you click an activity link, the openActivity method in the CommandServlet executes and passes the activity and document objects to the session. This information is used in the DisplayActivity.jsp.

```
public void openActivity(HttpServletRequest request, HttpServletResponse response) {
        try {
                activity =          new ExtendedActivityAccessBean(new ExecutionObjectKey(activityKeyString),request);
                BinderAccessBean binder = activity.getBinder();
                ExtendedDocumentAccessBean document = binder.getMainDocument();

                // set activity and document objects in HttpSession
                request.getSession(true).setAttribute("document", document);
                request.getSession(true).setAttribute("activity", activity);

                // Opens the activity with the JSP specified
                returnURL(request, response, "JSP/DisplayActivity.jsp");
                ......
        } //openActivity method
```

In the DisplayActivity.jsp, we use the useBean tag to access the activity and the name of the document passed and use this information in the JSP codes:

```
<jsp:useBean id="document" type="com.ibm.websphere.workflow.ExtendedDocumentAccessBean"
scope="session"></jsp:useBean>
<jsp:useBean id="activity" type="com.ibm.websphere.workflow.ExtendedActivityAccessBean"
scope="session"></jsp:useBean>
```

Using the <jsp:include page="<%= document.getJSP() %>" ...> line, we display the rest of the page with the JSP returned.

Step 7: Completing or claiming activities
The following code provides context-sensitive links or buttons to complete or claim an activity. It checks the status of the activity first and generates the button to complete it on-the-fly. When a user clicks the Complete button to submit a document, CommandServlet executes the PO_SaveAndComplete method:

```
<% if (activityState.equalsIgnoreCase("OPEN.RUNNING")) {  %>
<INPUT type="submit" name="Complete" value="Complete">
<INPUT type="hidden" name="COMMAND" value="POSAVECOMPLETE">
<% }  %>

public void PO_saveAndComplete(HttpServletRequest request,HttpServletResponse response) {
            .....................................................
            // get document and activity from HttpSession
            document =(com.ibm.websphere.workflow.ExtendedDocumentAccessBean)
            request.getSession(true).getAttribute("document");
            activity =(com.ibm.websphere.workflow.ExtendedActivityAccessBean)
            request.getSession(true).getAttribute("activity");

            // Get parameters from the HttpRequest objects and manipulate data if necessary.
            String stock = request.getParameter("Stock");
            String description = request.getParameter("Description");
            tmpField = request.getParameter("Quantity");
            if (!tmpField.equals(""))
                  iQuantity = Integer.valueOf(tmpField).intValue();
            .............................................
            //Then save the data back to document.
            document.updateField("Stock", stock);
            document.updateField("Description", description);
            document.updateField("Quantity", new Integer(iQuantity));
            .............................................
            activity.complete();
            .............................................
      }
```

You also can design decision and/or routing options before completing or claiming. To do this, get the ExtendedDocumentAccessBean object and setDecision method before calling activity.claim() or activity.complete():

```
activity.setDecision("Decision",request.getParameter("Decision"));
```

To enable/disable decision/routing options, add the following code to your JSP:

```
<% if (activityName.equalsIgnoreCase("Options")&& activity.getState().equalsIgnoreCase("OPEN.RUNNING"))
{%>
<INPUT type="radio" name="Decision" value="Multiple Source">Multiple Source   active<BR>
<INPUT type="radio" name="Decision" value="Single Source">Single Source<BR>
<% }  else if(activityName.equalsIgnoreCase("Options")) {%>
<INPUT type="radio" name="Decision" value="Multiple Source" disabled>Multiple Source disable<BR>
<INPUT type="radio" name="Decision" value="Single Source" disabled>Single Source<BR>
<%}%>
```

Building Web applications with the power of Lotus Workflow

We've shown how you can use Lotus Workflow Java API in WebSphere and Domino. Our goal is to give you an indication of how easily you can integrate the power of Lotus Workflow into your Web-based e-business solutions. Give it a try!

ABOUT JOANN JORDAN
JoAnn Jordan joined Lotus/IBM in 1996 and has been a member of the Lotus Workflow quality assurance team since 2000. Previously, JoAnn worked on products such as Lotus Smart Suite and Freelance Graphics.

ABOUT SEOL YOUNG PARK
Seol Young Park joined Lotus/IBM in 1994. She's been a member of the Lotus Workflow team since 1999, serving as a developer for the Lotus Workflow Engine. Previously, Seol Young worked on multiple projects for the International Product Development team.