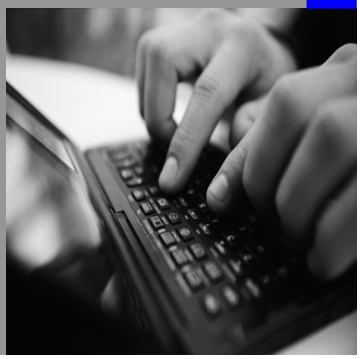


How-to Guide
SAP NetWeaver 2004s

# How To...
# Work with Java Proxies

Version 1.00 – August 2006

Applicable Releases:
SAP NetWeaver 2004s
Process Integration
Enabling Application-to-Application Processes

# Content

# 1 Scenario

This guide gives a short introduction to using Java proxies to connect applications with an XI Integration Server. The guide assumes that you are using the SAP NetWeaver Development Studio as Java IDE.

# 2 Introduction

The Java proxy runtime is designed to build Java applications that can talk directly to the Integration Server without needing any special adapters. The Java proxy runtime comes with a message processing and queuing system and provides security mechanisms.

# 3 The Step By Step Solution

We use the following standard message interfaces for the scenario:

- `FlightSeatAvailabilityQuery_Out` in namespace *http://sap.com/xi/XI/Demo/Agency2*
- `FlightSeatAvailabilityQuery_In` in namespace *http://sap.com/xi/XI/Demo/Airline*

You will find both interfaces in the Integration Repository in software component version SAP BASIS 6.40, which is part of every XI 3.0 installation.
This guide introduces all the relevant steps to create Java proxies and run a simple test scenario. You can combine this scenario with the XI 3.0 demo examples.

## 3.1 Installing the Java Proxy Runtime

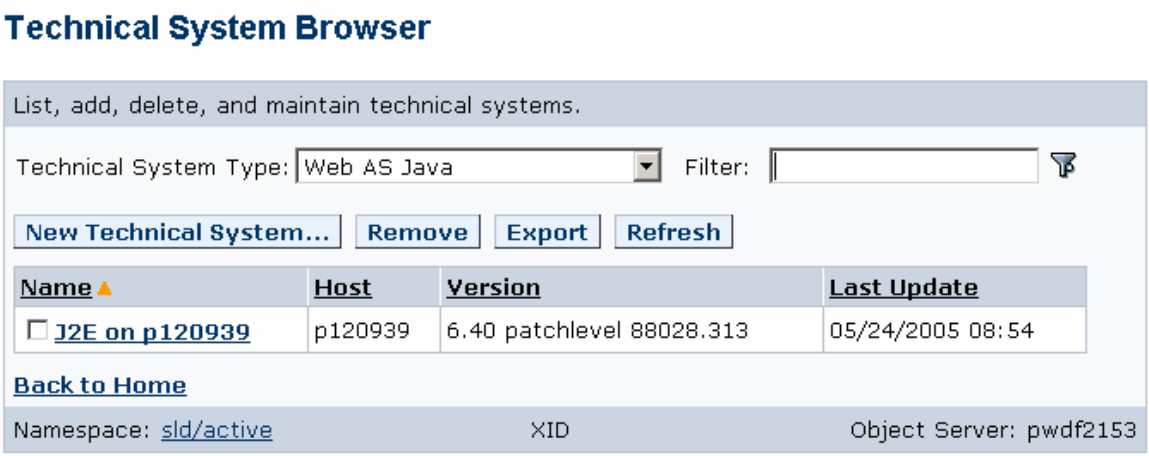
The Java proxy runtime is part of the XI 3.0 J2EE Adapter Framework. Though the Adapter Framework is part of the XI 3.0 installation, we strongly advise you not to deploy any application code in the XI system. You need to install a non-central Adapter Framework. Follow the installation guide for the J2EE Adapter Framework.


The Java proxy runtime is not designed to work with the Partner Connectivity Kit (PCK).

## 3.2 Configuring the Java Proxy Runtime

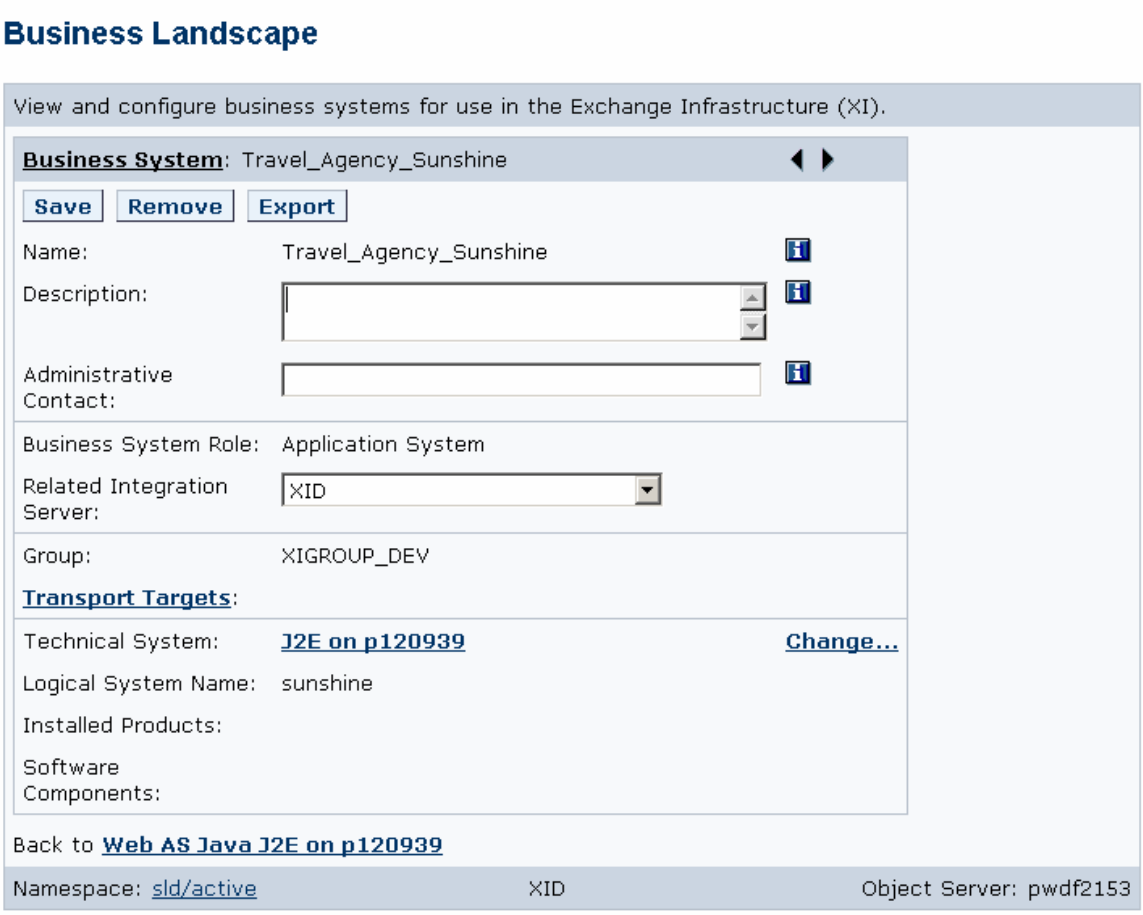
Once you have maintained the SLD access using the J2EE Visual Administrator, you will see an entry in the SLD as a Web AS Java technical system. The name of the technical system is the SID of the Adapter Framework. The default name of a non-central Adapter Framework is J2E.

Example of a Technical System (automatically generated):




Define a business system (type Web AS Java) related to this technical system. This business system serves as the default sender system for client proxies and should be used as the receiver system for server proxies as well.

Example of a Business System:




You must not assign more than one business system to the same technical system *Web AS Java*.

## 3.3 Setting up the Java IDE

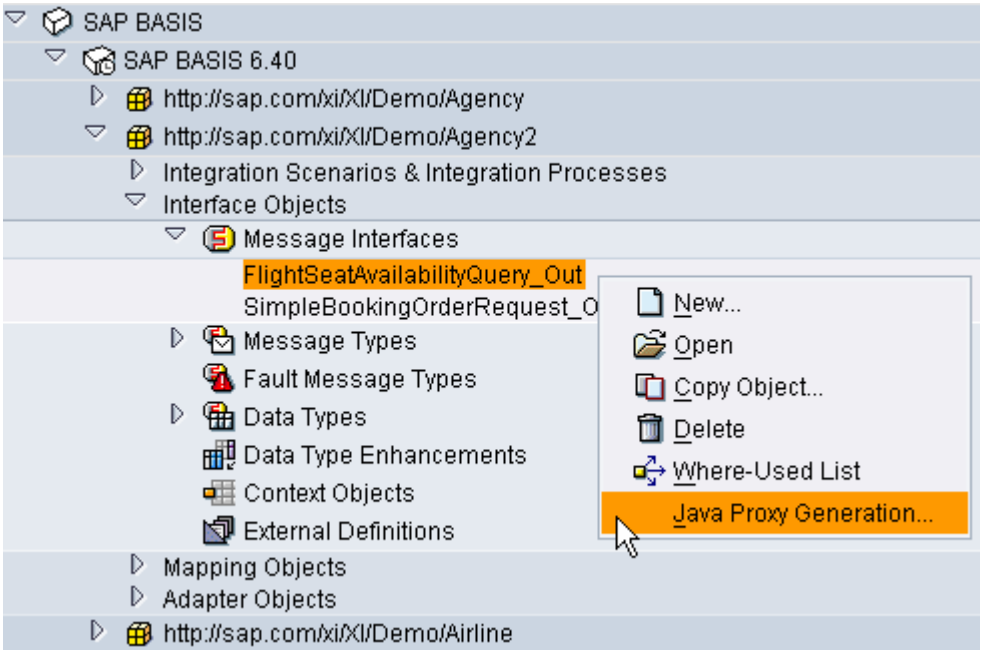
To compile the Java proxy source code you have to apply several libraries, which you will find on your non-central Adapter Framework installation:

| | |
|---|---|
| `aii_proxy_xirt.jar` | C:\usr\sap\J2E\JC00\j2ee\cluster\server0\bin\ext\com.sap.aii.proxy.xiruntime |
| `aii_msg_runtime.jar` | C:\usr\sap\J2E\JC00\j2ee\cluster\server0\bin\ext\com.sap.aii.messaging.runtime |
| `aii_utilxi_misc.jar` | C:\usr\sap\J2E\JC00\j2ee\cluster\server0\bin\ext\com.sap.xi.util.misc |
| `guidgenerator.jar` | C:\usr\sap\J2E\JC00\j2ee\cluster\server0\bin\ext\com.sap.guid |

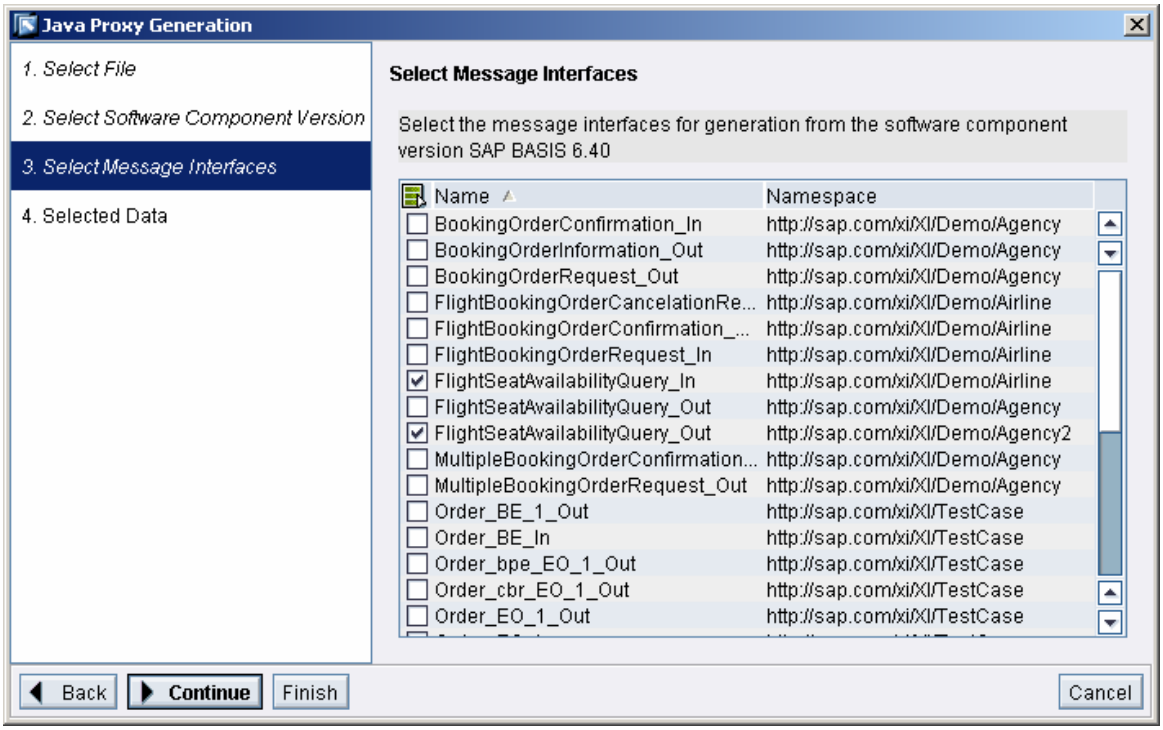
The path is an example from an installation on a Windows server and may be different in your environment.

## 3.4 Generating the Java Proxies

Log on to the Integration Builder and choose one of the message interfaces that you want to use. Click the message interface with the secondary mouse button and choose *Java Proxy Generation* from the context menu.



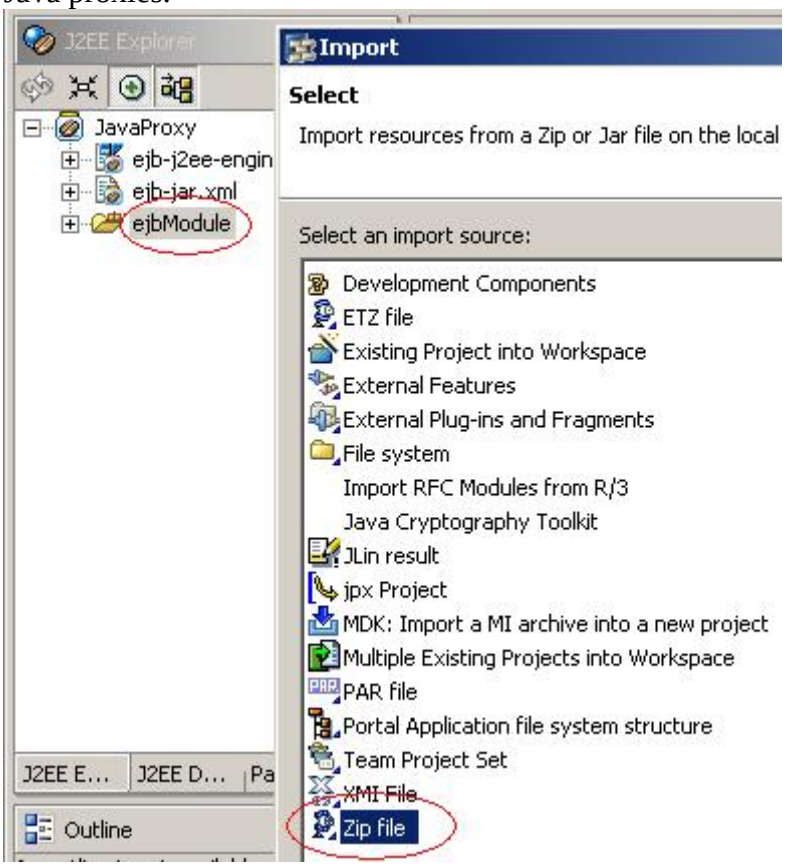

On the first screen of the wizard, choose *Create New Archive* and enter a file name with .zip extension.
On the second screen, choose software component version `SAP BASIS 6.40`.
On the next screen, select the interface



`FlightSeatAvailabilityQuery_Out` from namespace *http://sap.com/xi/XI/Demo/Agency2* and `FlightSeatAvailabilityQuery_In` from namespace *http://sap.com/xi/XI/Demo/Airline*.
On the last screen, check your work and choose *Finish*.

## 3.5 Importing the Java Proxies to the IDE

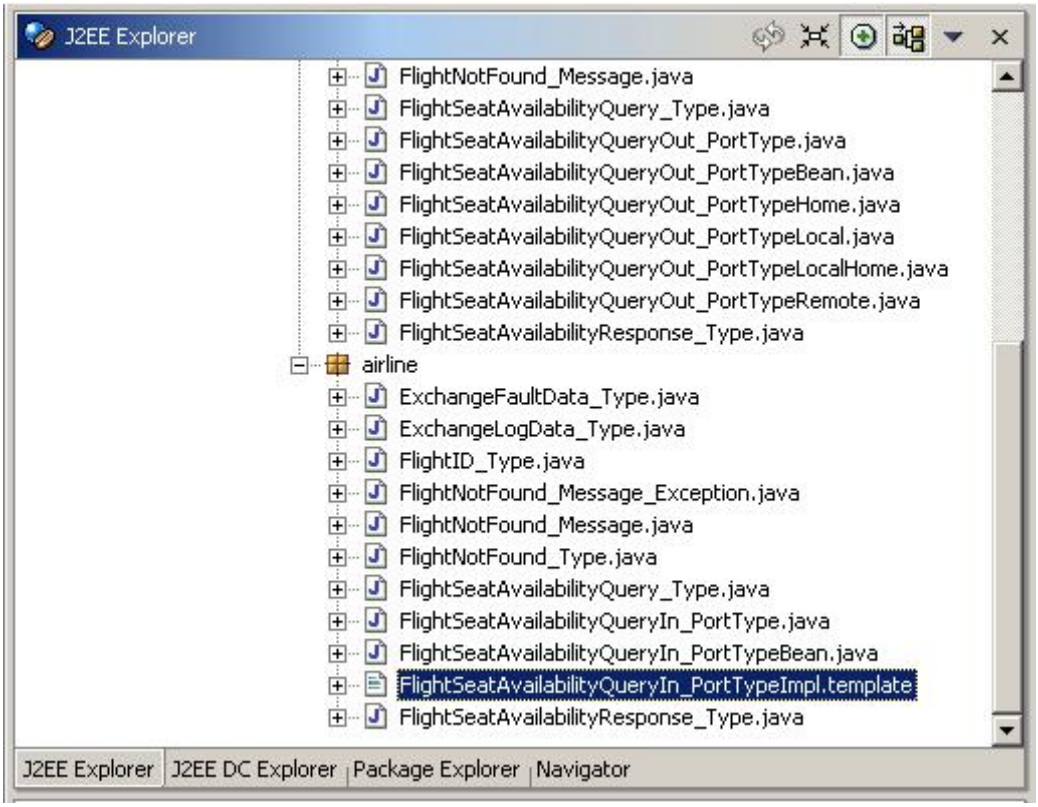
First, you create a new project of type *J2EE -> EJB Module Project.*
Open your project and select the *ejbModule* folder. This is important to make sure that the generated proxies are recognized as EJB source files.
Choose *File -> Import* in the main menu.
On the next screen, choose *Zip file*. Then select your zip or JAR file with your generated Java proxies.




The next step is to assign the library files to your Java project. You need these to compile the Java proxies. Choose *Project -> Properties* and then *Java Build Path -> Libraries*. Choose *Add External JARs*. You select the library files mentioned before.

## 3.6 Writing Application Code for the Server Proxies

If you have generated server Java proxies, the main file is generated as a template to prevent user code from being accidentally overwritten. Look for the files with suffix `.template` and copy them to a Java file. You can do this easily by double-clicking the file and choosing the *Save As* function.




Now you can write your application code for the server Java proxy.

### 3.7 Code Example for Server Java Proxies

The following example explains how parameters and application fault exceptions are handled.

```
package com.sap.xi.xI.demo.airline;

public class FlightSeatAvailabilityQueryIn_PortTypeImpl
             extends com.sap.aii.proxy.xiruntime.core.AbstractProxy
             implements FlightSeatAvailabilityQueryIn_PortType {

  public
       com.sap.xi.xI.demo.airline.FlightSeatAvailabilityResponse_Type
       flightSeatAvailabilityQueryIn
       (com.sap.xi.xI.demo.airline.FlightSeatAvailabilityQuery_Type
        flightSeatAvailabilityQuery)
    throws
       com.sap.xi.xI.demo.airline.FlightNotFound_Message_Exception,
       com.sap.aii.proxy.xiruntime.core.SystemFaultException,
       com.sap.aii.proxy.xiruntime.core.ApplicationFaultException{

    boolean error = false;
    String errorText = "";

    // get input parameters
    FlightID_Type flightID = flightSeatAvailabilityQuery.getFlightID();
    String airlineID = flightID.getAirlineID();
    String connectionID = flightID.getConnectionID();
    java.util.Calendar flightDate = flightID.getFlightDate();
```

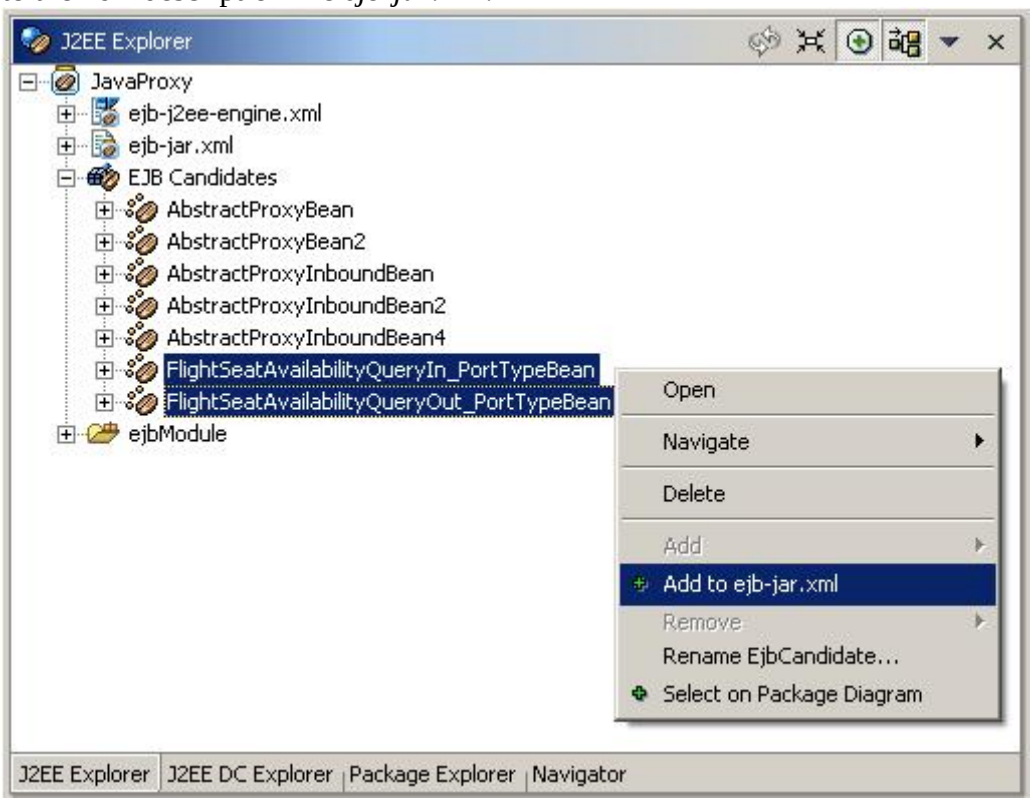
```
    // check input parameters
    if (airlineID.length()==0) {
      error = true;
      errorText = "airline ID is missing";
    }
    if (connectionID.length()==0) {
      error = true;
      errorText = "connection ID is missing";
    }

    if (error){
      com.sap.xi.xI.demo.agency.ExchangeFaultData_Type standard =
          new com.sap.xi.xI.demo.agency.ExchangeFaultData_Type();
      FlightNotFound_Type flightNotFoundType =
          new FlightNotFound_Type();
      FlightNotFound_Message flightNotFoundMessage =
          new FlightNotFound_Message();
      FlightNotFound_Message_Exception flightNotFound =
          new FlightNotFound_Message_Exception();

      // put error text to fault message
      standard.setFaultText(errorText);
      flightNotFoundType.setStandard(standard);
      flightNotFoundMessage.setFlightNotFound(flightNotFoundType);
      flightNotFound.setFlightNotFound_Message(flightNotFoundMessage);

      // throw flight not found message exception
      throw flightNotFound;

    } else {

      // provide output parameters
      // (just return any fix values)
      FlightSeatAvailabilityResponse_Type response =
          new FlightSeatAvailabilityResponse_Type();
      response.setBusinessFreeSeats(18);
      response.setBusinessMaxSeats(20);
      response.setEconomyFreeSeats(188);
      response.setEconomyMaxSeats(200);
      response.setFirstFreeSeats(12);
      response.setFirstMaxSeats(15);
      return response;
    }

  }

}
```

The code for the server proxy has to be written before deploying the Java proxy beans to the Adapter Framework. Inside the server proxies, you can use all functions of the J2EE engine, for example, call other EJBs, read or write database entries, or use the Java Messaging Service (JMS).

## 3.8 Creating the EJBs from the Java Proxies

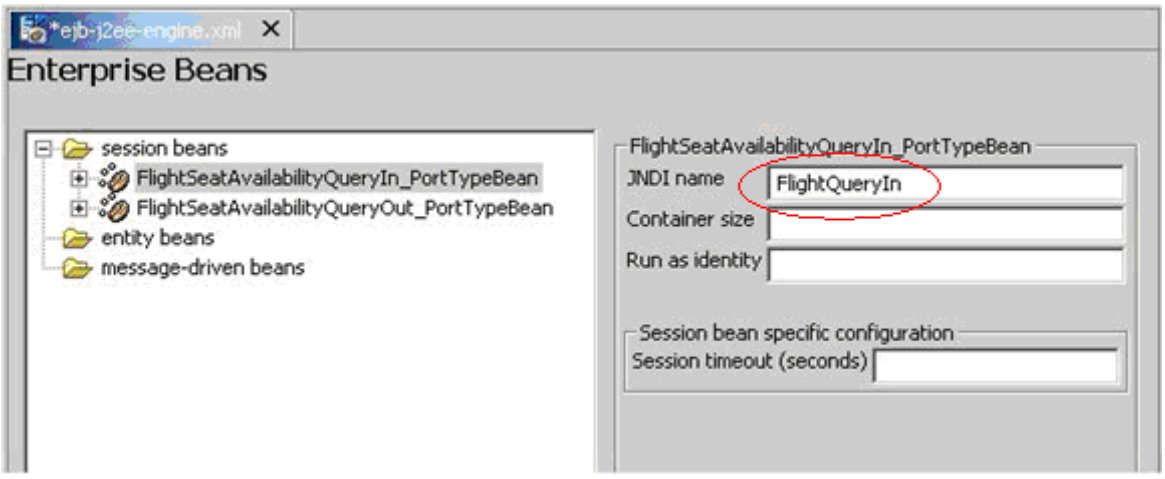
The developer studio automatically recognizes (with help of the @ejb tags in the comment lines) which generated files follow the EJB convention and provides them under

the node EJB Candidates. Select them and click the secondary mouse button to add them to the EJB description file *ejb-jar.xml*.




If you are working with server proxies, you receive the error message: *Bean problem: No interface classes found*. This is because no Java source code is available for the EJB interface classes (the interface classes are provided in the library `aii_proxy_xirt.jar`). Go to the *Package Explorer* view in the Development Studio. Select your project, choose *Close Project* in the context menu, and then *Open Project*. Then go back to the *J2EE Explorer* view.

You should assign a JNDI name in the `ejb-j2ee-engine.xml`. You need this name to call the Java Proxies. Here is an example:

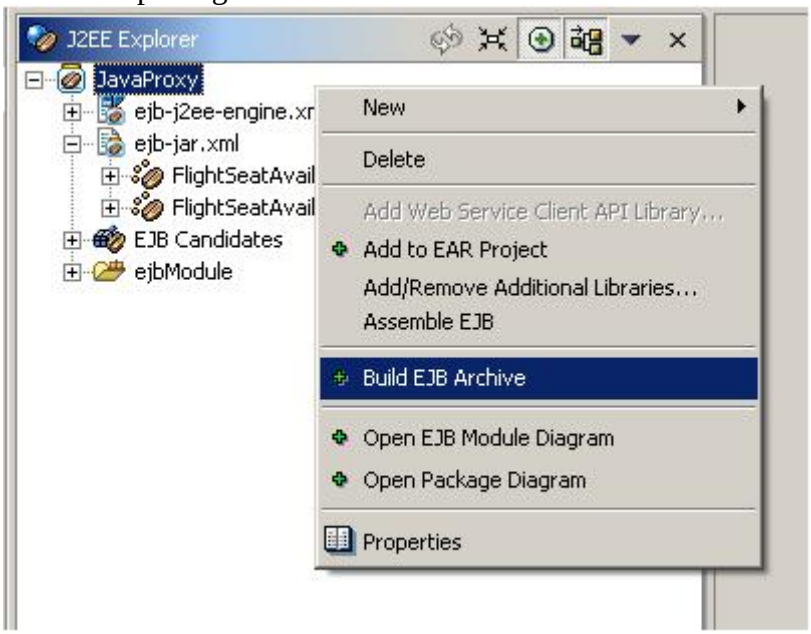


If you do not assign a JNDI name explicitly, the system will automatically take **`sap.com/<EAR Project Name>/<Bean Name>`** as the JNDI name.

If you want to access the payload or use other services of the Java proxy runtime after sending the message, you have to use *stateful* session beans. Edit the source of the ejb-jar.xml and change the *session type* to *Stateful*.

## 3.9 Creating a Java Archive

Select the package and choose “Build EJB Archive” in the context menu:




## 3.10 Creating an Enterprise Application Archive (EAR)

To deploy your Java proxies to the J2EE Adapter Framework, you have to create an EAR file.

Create a new project as *J2EE -> Enterprise Application Project*. Select your EJB project(s) as reference.

You need to assign library references to the project. Double-click *application-j2ee-engine.xml*; add a new *library* reference with reference type *weak* and provider name *sap.com* for the following libraries:

```
com.sap.aii.proxy.xiruntime
com.sap.aii.messaging.runtime
com.sap.xi.util.misc
com.sap.guid
```

Instead of assigning the references one by one, you can replace the source with the following code:

```
<application-j2ee-engine>
       <reference reference-type="weak">
              <reference-target
```

```
                        provider-name="sap.com"
                        target-type="library">com.sap.aii.proxy.xiruntime
                </reference-target>
        </reference>
        <reference reference-type="weak">
                <reference-target
                        provider-name="sap.com"
                        target-type="library">com.sap.aii.messaging.runtime
                </reference-target>
        </reference>
        <reference reference-type="weak">
                <reference-target
                        provider-name="sap.com"
                        target-type="library">com.sap.xi.util.misc
                </reference-target>
        </reference>
        <reference reference-type="weak">
                <reference-target
                        provider-name="sap.com"
                        target-type="library">com.sap.guid
                </reference-target>
        </reference>
        <provider-name>sap.com</provider-name>
        <fail-over-enable
                mode="disable"/>
</application-j2ee-engine>
```

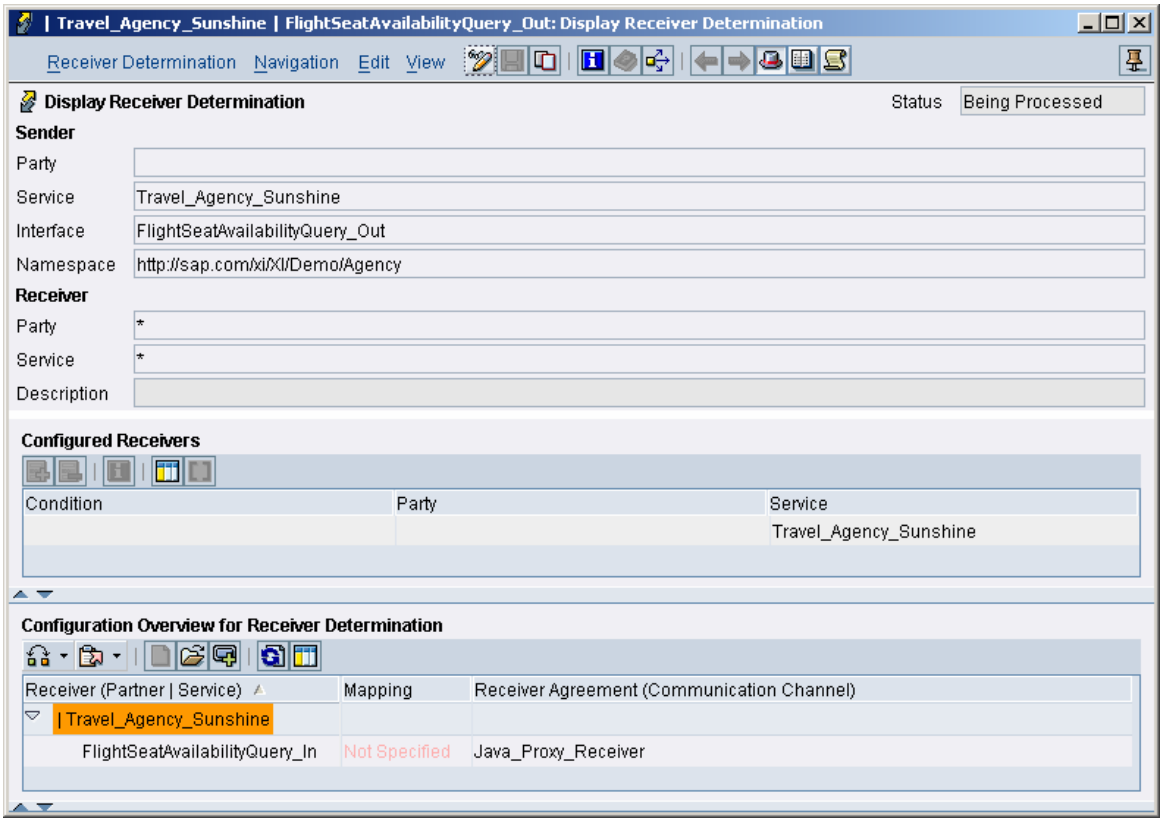
If you want to use additional libraries for your Java Proxy project, you might have to apply them here as well.

If you want to use libraries that you have deployed with help of the file aii_af_jmsproviderlib.sda, you must add the reference target type *com.sap.aii.af.jmsproviderlib*

Select the project and choose *Build Application Archive* in the context menu. Make sure that the libraries of the Java Proxy runtime are not in the generated EAR file, or delete them with WinZip. You can now deploy the EAR file from the Deploy Service in the Visual Administrator tool or with the Software Deployment Manager (SDM).

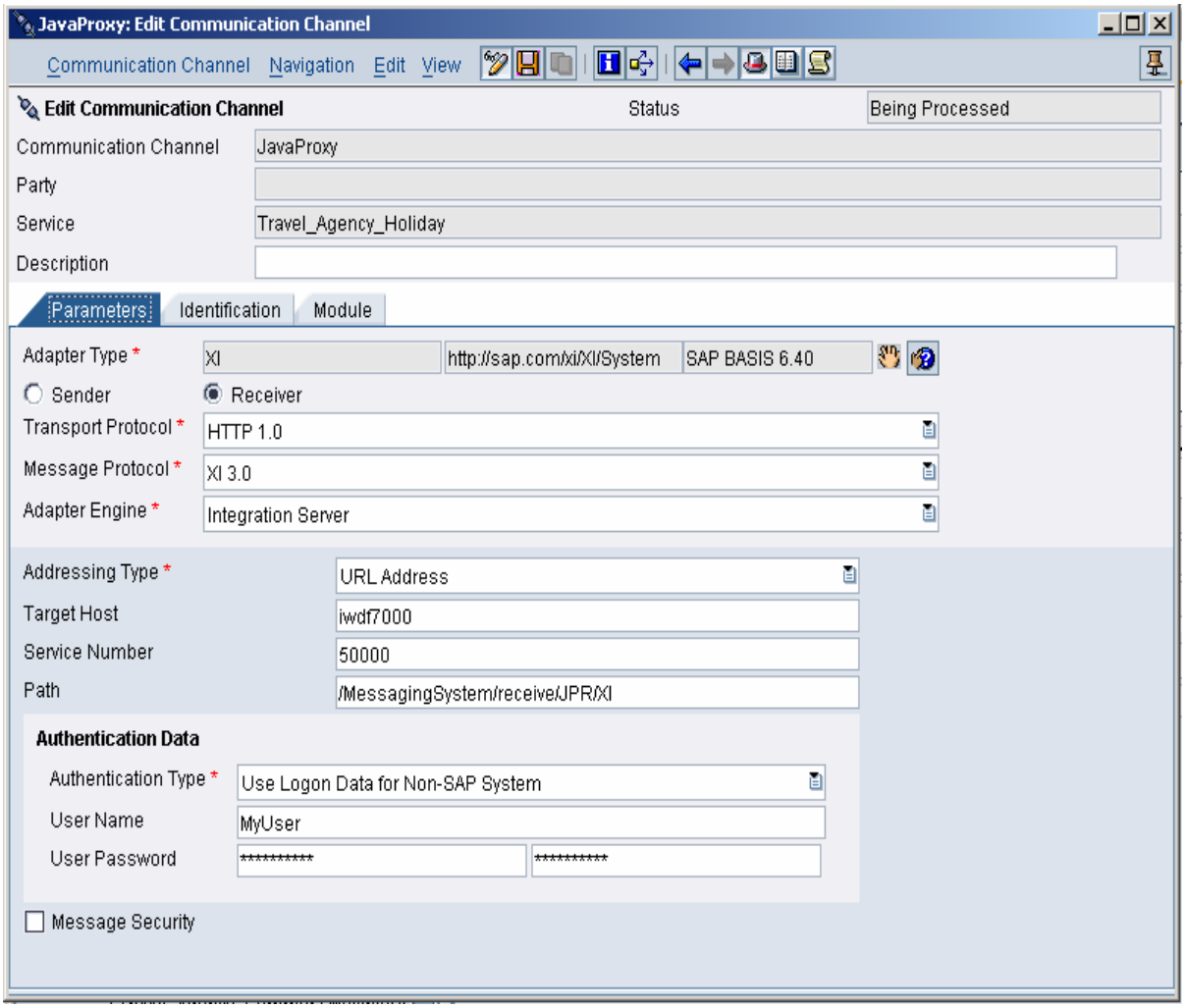
## 3.11 Configure your Client Java Proxies

When you send data with the client Java proxies, the sender service is automatically set to the business system that you have maintained in the SLD. The interface name and namespace are derived from the message interfaces from which the Java proxies are generated. For configuration in the Integration Directory, it is sufficient to create a receiver determination, an interface determination, and a receiver agreement for the receiver channel to which the message is to be sent. A sender agreement and a sender channel are only necessary for Java proxies if you have special requirements for message security.




## 3.12 Configure Your Server Java Proxies

To receive messages with a server Java proxy, you have to create a receiver channel as follows:




You have to enter the Adapter Engine where your Java proxy beans are deployed as the *Target Host*. The *Service Number* is the HTTP port of the Adapter Engine. You can use XIAPPLUSER or a user with similar roles as the user.

Make sure that you set *Authentication Type* to **`Use Logon Data for Non-SAP System.`**

The mapping between the interface name and the class name of the server Java proxy is done by the proxy server. You have to register the server Java proxies to the proxy server. You do this by typing a string directly in the address bar of your Internet browser as follows:

```
http://<Host>:<Port>/ProxyServer/register?ns=<Namespace>&interface=<MessageInterface>&bean=<JNDI_Name>&method=<MethodName>
```

If you want the system to make use of local EJBs, you have to prefix the JNDI name with **`localejbs/`**.

In our example we would have:

```
http://iwdf7000:50000/ProxyServer/register?ns=http://sap.com/xi/XI/Demo/Airline&interface=FlightSeatAvailabilityQuery_In&bean=FlightQueryIn&method=flightSeatAvailabilityQueryIn
```

## 3.13 Calling the Client Java Proxies from a J2SE Application

If your application is not running on a J2EE server, you can call the client Java proxies by using the JNDI service of the J2EE engine. You have to provide the URL of the p4 service of the J2EE server of the Adapter Framework, user, and password.

The following example explains how parameters and application fault exceptions are handled. Adjust the URL, user, and password parameters according to your environment.

```
package demo;

import com.sap.xi.xI.demo.agency2.*;
import java.text.*;
import java.util.*;
import javax.naming.*;
import javax.rmi.*;

public class JavaProxyCall {

  public static void main(String[] args) {

    FlightSeatAvailabilityQueryOut_PortTypeHome   queryOutHome;
    FlightSeatAvailabilityQueryOut_PortTypeRemote queryOutRemote;

    String airlineId = "LH";
    String connectionId = "0400";
    String flightDate = "2004-12-01";

    // check for EJB class on server

    try {
      // Get naming context
      Properties p = new Properties();
      p.put(Context.INITIAL_CONTEXT_FACTORY,
            "com.sap.engine.services.jndi.InitialContextFactoryImpl");
      p.put(Context.PROVIDER_URL, "YourServer:50004");
      p.put(Context.SECURITY_PRINCIPAL, "YourUser");
      p.put(Context.SECURITY_CREDENTIALS, "YourPassword");
      Context ctx = new InitialContext(p);
      // Look up JNDI name of proxy bean
      Object ref = ctx.lookup("FlightQueryOut");
      // Cast to Home interface
      queryOutHome = (FlightSeatAvailabilityQueryOut_PortTypeHome)
                      PortableRemoteObject.narrow(ref,

FlightSeatAvailabilityQueryOut_PortTypeHome.class);
      // Get Remote interface
      queryOutRemote = queryOutHome.create();
    } catch (Exception e) {
      System.out.println("RemoteException occurred: "+e.getMessage());
      e.printStackTrace();
      return;
    }

    // Set parameters for the proxy call
    FlightSeatAvailabilityQuery_Type flightQuery =
        new FlightSeatAvailabilityQuery_Type();
    FlightID_Type flightId = new FlightID_Type();
    flightId.setAirlineID(airlineId);
    flightId.setConnectionID(connectionId);
```

```
    flightId.setFlightDate(getCalendarFlight(flightDate));
    flightQuery.setFlightID(flightId);

    // Call the Java proxy and get the return parameters
    try {
      // initialise the Message Specifier
      queryOutRemote.$messageSpecifier();
      FlightSeatAvailabilityResponse_Type response =
          queryOutRemote.flightSeatAvailabilityQueryOut(flightQuery);

      // write result to screen:
      System.out.println("First Class:");
      System.out.println("max:  "+response.getFirstMaxSeats());
      System.out.println("free: "+response.getFirstFreeSeats());
      System.out.println("Business Class:");
      System.out.println("max:  "+response.getBusinessMaxSeats());
      System.out.println("free: "+response.getBusinessFreeSeats());
      System.out.println("Economy Class:");
      System.out.println("max:  "+response.getEconomyMaxSeats());
      System.out.println("free: "+response.getEconomyFreeSeats());
    // Handle application fault exception
    } catch (FlightNotFound_Message_Exception e) {
      ExchangeFaultData_Type et
   = e.getFlightNotFound_Message().getFlightNotFound().getStandard();
      System.out.println("FlightNotFound_Message_Exception occurred: "
                         + et.getFaultText());
    // Handle message details
       if (et.isSetFaultDetail()){
         ExchangeLogData_Type[] faultDetail = et.getFaultDetail();
         for (int i=0;i<faultDetail.length;i++){
           System.out.println(faultDetail[i].getText());
         }
       }
    // Handle other exceptions
    } catch (com.sap.aii.proxy.xiruntime.core.FaultException e) {
      System.out.println("FaultException occurred: "+e.getMessage());
      e.printStackTrace();
    } catch (java.rmi.RemoteException e) {
      System.out.println("RemoteException occurred: "+e.getMessage());
      e.printStackTrace();
    } catch (Exception e) {
      System.out.println("Undefined Exception occurred: "
                         +e.getMessage());
      e.printStackTrace();
    }
}

// method for preparing date parameters
  private static Calendar getCalendarFlight(String date) {
    long millis = 0L;
    SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
    try {
      millis = dateFormat.parse(date).getTime();
    } catch (ParseException parseexception) {
    }
    Calendar calendarFlight = Calendar.getInstance();
    calendarFlight.setTimeInMillis(millis);
    return calendarFlight;
  }
}
```

To make this example run, you have to provide the Java archive created in step 3.9 as the library for the proxy call and additionally the following libraries:

From XI:
**aii_adapter_xi_svc.jar, aii_af_cci.jar, aii_af_cpa.jar, aii_af_mp.jar, aii_af_ms_api.jar, aii_af_ms_spi.jar, aii_af_service_message_security.jar**

From the J2EE client JARs: (*/usr/sap/J2E/JC00/j2ee/j2eeclient/signed*):
**ejb20.jar, exception.jar, guidgenerator.jar, jARM.jar, jperflib.jar, jta.jar, log_api.jar, logging.jar, sapj2eeclient.jar, sapni.jar, sapxmltoolkit.jar**

You also need the library with the generated proxies.

## 3.14 Calling the Client Java Proxies from a J2EE Application

If you call the Java proxies from another EJB or from a servlet, the call is different. You need not provide user and password, as the program is already running on the J2EE server.
Here is the relevant part of the above code adjusted for a call within the J2EE server:

```
try {
  // Get naming context
  Context ctx = new InitialContext();
  // Look up the EJB name in the environment
  Object ref = ctx.lookup("java:comp/env/ejb/FlightQueryOut");
  // Cast to Home interface
  queryOutHome = (FlightSeatAvailabilityQueryOut_PortTypeHome)
                 PortableRemoteObject.narrow(ref,
                 FlightSeatAvailabilityQueryOut_PortTypeHome.class);
  // Get Remote interface
  queryOutRemote = queryOutHome.create();
} catch (Exception e) {
  System.out.println("RemoteException occurred: "+e.getMessage());
  e.printStackTrace();
  return;
}
```

The link between the calling servlet and the called Java proxy EJB is made in the `web.xml`. The link from the calling EJB to the Java proxy EJB is made in the `ejb-jar.xml`.
In both cases you have to add the following entry for remote calls:

```
<ejb-ref>
<ejb-ref-name>ejb/FlightQueryOut</ejb-ref-name>
<ejb-ref-type>Session</ejb-ref-type>
<home>com.sap.xi.xI.demo.agency.FlightSeatAvailabilityQueryOut_PortTypeHome
</home>
<remote>com.sap.xi.xI.demo.agency.FlightSeatAvailabilityQueryOut_PortTypeRemote
</remote>
<ejb-link>JavaProxy.jar#FlightSeatAvailabilityQueryOut_PortTypeBean</ejb-link>
</ejb-ref>
```

You have to add the following entry for local calls:

```
<ejb-local-ref>
<ejb-ref-name>ejb/FlightQueryOut</ejb-ref-name>
<ejb-ref-type>Session</ejb-ref-type>
<local-
home>com.sap.xi.xI.demo.agency.FlightSeatAvailabilityQueryOut_PortTypeLocalHome
</local-home>
<local>com.sap.xi.xI.demo.agency.FlightSeatAvailabilityQueryOut_PortTypeLocal
</local>
<ejb-link>JavaProxy.jar#FlightSeatAvailabilityQueryOut_PortTypeBean</ejb-link>
</ejb-local-ref>
```

You can use the Developer Studio to generate these entries automatically.

### 3.15 Testing the Java Proxies

Once your configuration is complete and you start your J2SE Java program, you should receive the following output:

```
First Class:
max:  15
free: 12
Business Class:
max:  20
free: 18
Economy Class:
max:  200
free: 188
```

# 4 Monitoring the Status of Java Proxy Runtime

If you have troubles with your Java Proxies, you should first check the status of the Java Proxy Runtime (JPR) inside the adapter monitor.
Start the *Runtime Workbench* from the *SAP Exchange Infrastructure* start page, choose Component Monitoring and select the Adapter Engine (where the Java Proxies are deployed) as the component.
On the Status tab page you choose *Adapter Monitoring* to call the adapter monitor.
In the adapter Monitor expand the namespace *http://sap.com/xi/XI/System* and choose *JPR* to open the Monitor for the Java Proxy Runtime.


| Status | Name | Text |
|---|---|---|
| | Proxy Server | java:comp/env/JPR4TS<br>com.sap.aii.proxy.xiruntime.sbeans.JPRBean |
| | SLD access | SLD host:port = PWDF2153.wdf.sap.corp:54000<br>JPR configuration:<br>subsystem = com.sap.aii.utilxi.sld.JavaProxyRuntime@1e14d25<br>applicationId = prt<br>technicalHost = com.sap.aii.utilxi.sld.J2EESystem@8fe015<br>host = PWDF2153<br>httpPort = 54000<br>httpsPort = 54001<br>contextRoot = prt<br>user = XIAFUSER<br>password = ******<br>locale = en<br>SAPsystem = XID<br>dbHost = PWDF2153<br>businessSystem = Travel_Agency_Holiday |
| | Messaging System | Connection name = JPR<br>Time out = 5000000 msec<br>IS URL = http://pwdf2153.wdf.sap.corp:50040/sap/xi/engine?type=entry<br>IS client = 100 |
| | Logical locking | Namespace = JPR<br>Owner = 20060420133147441000000PWDF2153.....405037550 |
| | Properties | D:\usr\sap\XID\DVEBMGS40\j2ee\cluster\server0\jpf.properties with 0 entries |
| | Registry | D:\usr\sap\XID\SYS\global\xi\jpf.registry with 5 entries<br>Click here to list the contents |
| | Cache | D:\usr\sap\XID\SYS\global\xi\jpf.cache with 2 entries<br>com.sap.aii.connect.landscape.name = PWDF2153.wdf.sap.corp:54000<br>ServiceName = Travel_Agency_Holiday |
| | JARM | off |
| | Version | 3.0.12 from $DateTime: 2005/05/17 17:24:52 $<br>$Id: //tc/aii/30_REL/src/_proxy/java/com/sap/aii/proxy/xiruntime/core/XmlSystemAccess.java#7 $ |

Page 1 / 1