

# **Oracle® Business Process Management**

Oracle BPM Process API

10g Release 3 (10.3.1)

January 2009

Copyright © 2006, 2008, 2009 Oracle. All rights reserved.

The Programs (which include both the software and documentation) contain proprietary information; they are provided under a license agreement containing restrictions on use and disclosure and are also protected by copyright, patent, and other intellectual and industrial property laws. Reverse engineering, disassembly, or decompilation of the Programs, except to the extent required to obtain interoperability with other independently created software or as specified by law, is prohibited.

The information contained in this document is subject to change without notice. If you find any problems in the documentation, please report them to us in writing. This document is not warranted to be error-free.

Except as may be expressly permitted in your license agreement for these Programs, no part of these Programs may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose.

If the Programs are delivered to the United States Government or anyone licensing or using the Programs on behalf of the United States Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the Programs, including documentation and technical data, shall be subject to the licensing restrictions set forth in the applicable Oracle license agreement, and, to the extent applicable, the additional rights set forth in FAR 52.227-19, Commercial Computer Software--Restricted Rights (June 1987). Oracle USA, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

The Programs are not intended for use in any nuclear, aviation, mass transit, medical, or other inherently dangerous applications. It shall be the licensee's responsibility to take all appropriate fail-safe, backup, redundancy and other measures to ensure the safe use of such applications if the Programs are used for such purposes, and we disclaim liability for any damages caused by such use of the Programs.

Oracle, JD Edwards, PeopleSoft, and Siebel are registered trademarks of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

The Programs may provide links to Web sites and access to content, products, and services from third parties. Oracle is not responsible for the availability of, or any content provided on, third-party Web sites. You bear all risks associated with the use of such content. If you choose to purchase any products or services from a third party, the relationship is directly between you and the third party. Oracle is not responsible for: (a) the quality of third-party products or services; or (b) fulfilling any of the terms of the agreement with the third party, including delivery of products or services and warranty obligations related to purchased products or services. Oracle is not responsible for any loss or damage of any sort that you may incur from dealing with any third party.

# Contents

<b>Oracle BPM Process API (PAPI).....</b>	<b>4</b>
Oracle BPM Process API (PAPI).....	4
PAPI Overview.....	4
Process API Usage Scenarios.....	4
Process API Architecture Overview.....	5
Structure of a Java PAPI Application.....	6
Writing Your First Java PAPI Program.....	7
Compiling a Java PAPI Program.....	11
Running a Java PAPI Program.....	12
Connecting to an Engine Running in Studio.....	13
Configuring the Log of a PAPI Client.....	13
PAPI Client Log Properties.....	14
Log Severity Levels.....	14
Running a PAPI Client in a J2EE Distributed Environment.....	15
Configuring JNDI for a Single J2EE Process Execution Engine.....	16
Configuring JNDI for Multiple J2EE Engines.....	17
<b>Oracle BPM PAPI Web Service.....</b>	<b>19</b>
PAPI Web Service Overview.....	19
What's New in PAPI Web Service 2.0?.....	19
PAPI Web Service Usage Scenarios.....	20
PAPI Web Service Architecture Overview.....	20
Enabling PAPI Web Service.....	21
Enabling PAPI Web Service in Oracle BPM Studio.....	21
Enabling PAPI Web Service in Oracle BPM Enterprise.....	21
Installing PAPI Web Service on a J2EE Application Server.....	22
PAPI Web Service Configuration.....	23
PAPI Web Service Configuration Console.....	23
Editing PAPI Web Service Configuration.....	24
Configuring PAPI Web Service Log.....	25
PAPI Web Service Security Authentication.....	26
Developing a Custom Sign On Implementation.....	27
Configuring Custom Single Sign-On Authentication.....	27
Configuring Username Token Profile.....	27
Configuring HTTP Basic Authentication.....	28
Configuring PRESET Authentication.....	29
PAPI Web Service Examples.....	29
Java JAX WS Client Example.....	29
JAX-WS Client Main -Class.....	30
Running Java JAX WS Client Example.....	35
PAPI Web Service .NET Client Example.....	35
.NET Client Main-Class.....	36

## Oracle BPM Process API (PAPI)

---

This guide is an introduction to Oracle BPM Process API (PAPI). It contains relevant information about the API architecture, an analysis of the structure of a Java application using PAPI, and instructions on how to compile and run a Java PAPI application.

### Oracle BPM Process API (PAPI)

This guide is an introduction to Oracle BPM Process API (PAPI). It contains relevant information about the API architecture, an analysis of the structure of a Java application using PAPI, and instructions on how to compile and run a Java PAPI application.

### PAPI Overview

PAPI is a Java client-server API that allows you to interact with processes deployed on an Oracle BPM Process Execution Engine.

PAPI is a Java API a Java API that can be invoked by any Java application written in Java 1.5.

PAPI provides the following operations:

- Create, send and abort process instances
- Select process instances
- Reassign process instances
- Audit an instance
- Suspend and resume process instances
- Grab and un-grab process instances
- Run interactive and global interactive activities
- Run external tasks
- Send notifications
- Get a list of process instances
- Get a list of deployed processes
- List the activities in a deployed process
- Get the latest version of a deployed process
- Manage views and presentations
- Manage attachments

Oracle BPM WorkSpace is built on PAPI. All the communication between the WorkSpace and the Process Engine is done through PAPI.

The complete reference documentation for PAPI is available at

[http://download.oracle.com/docs/cd/E13154\\_01/bpm/docs65/papi\\_javadocs/index.html](http://download.oracle.com/docs/cd/E13154_01/bpm/docs65/papi_javadocs/index.html).

### Process API Usage Scenarios

PAPI provides a way for external applications to interact with Oracle BPM.

You should use PAPI to interact with external or legacy applications. Some common usage scenarios are:

- A web application that needs to create a process instance in Oracle BPM with the information entered by the user
- An external application whose execution final result is the execution of an Oracle BPM process
- An external application that need to perform a search, or to list information about processes in Oracle BPM
- An external application that needs to trigger the execution of an activity
- Batch or automatic processing of process instances

Although you can use PAPI to replace Oracle BPM WorkSpace with a similar graphical application interface, consider customizing Oracle BPM WorkSpace to suit your needs instead. Replacing Oracle BPM WorkSpace causes you to lose-and then to have to rebuild-most of the out-of-the-box functionality WorkSpace provides including, for example, both the authentication framework and the interactive execution framework.

## Process API Architecture Overview

PAPI is a Java API that allows you to build a client to connect to the engine and perform operations on the deployed processes.

When PAPI is initialized, the connected user is authenticated against the data in the directory service. Once authenticated, the Java client using PAPI can interact with any of the engines configured in the directory service.

PAPI must connect to a Process Execution Engine only to search for or operate on process instances and deployed processes. It does not need to connect to an Engine if a request or operation requires only data stored in the Directory Service.

The following operations do not require a connection to the engine:

- List available views
- Search for a specific view
- List available presentations
- List participants in the organization

You can successfully execute these operations even when the Process Engine is down.

PAPI can connect to one or more engines at a time, provided they are configured in the same directory service. When a client makes a request, PAPI automatically routes this request to the corresponding engine.

The following diagram shows interaction between a PAPI client, the Directory Service and the Process Engine in runtime. The diagram shows a custom Java client and Oracle BPM Workspace that is also a PAPI client.

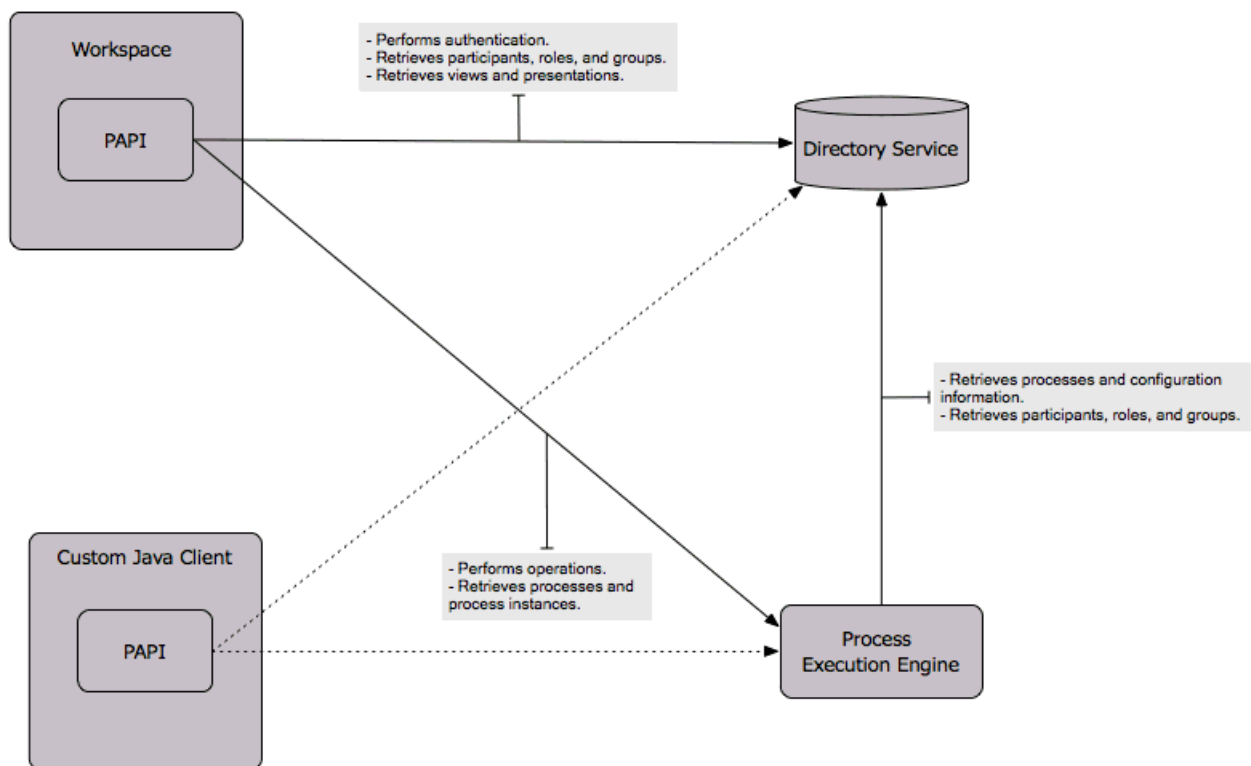


Figure 1: PAPI Components Runtime Diagram

## Structure of a Java PAPI Application

A Java PAPI application should follow a certain structure. The different methods that you need to invoke before and after performing operations with PAPI determine this structure.

A Java application that uses PAPI goes through the following stages:

1. Initialize the API.
2. Establish a session.
3. Operate with PAPI.
4. Close the session.
5. Release API resources.

### Initialize the API

The `ProcessService` class is the main entry point to the API. Before you start using PAPI, you must create a `ProcessService` object.

A `ProcessService` acts as a factory for `ProcessServiceSession` objects. To create a session, you must first create and configure a `ProcessService`.

When you create a `ProcessService` object, a connection to the Directory Service is established. This connection is used to load Oracle BPM's environment configuration information and later to authenticate the user creating the process service session.

## Establish a Session

A `ProcessServiceSession` represents the dialog between a participant and the Directory Service or one or more engines.

You need a `ProcessServiceSession` to manage and obtain information about process instances, participants, views, and presentations.

To create a `ProcessServiceSession` you need to provide the valid credentials—for example, the user identifier and password—of a participant that exists in the Directory Service.

## Operate with PAPI

Once you obtain a `ProcessServiceSession` you are ready to query for information and invoke any of the operations provided by PAPI.

## Close the Session

You need to close PAPI sessions before your application finishes so that caches are cleared and the connections to the engine are closed.

Leaving sessions open may both cause a memory leak and use network resources unnecessarily. This is because the allocated resources are never freed and the Engine continues to send information to the connected participant.

Leaving sessions open can also cause problems in updating a participant's role assignment. Because changes to role assignments are applied only after the last session has been closed, leaving a session open indefinitely means that changes to roles and permissions are never applied.

PAPI sessions do not expire by timing out. The application using PAPI is responsible for closing open sessions.

Once the session is closed it cannot be used again. Trying to invoke a method over a closed session results in an exception.

## Release API resources

It is advisable to close the `ProcessService` so that the resources it uses are released.

When a `ProcessService` is closed, the following events occur:

- All opened PAPI sessions are closed.
- Temporary files used by the API are deleted.
- The connection to the Directory Service is closed.
- Caches used by PAPI are cleared.

# Writing Your First Java PAPI Program

This section shows you how to build a Java PAPI Client that retrieves a list of process instances visible to the connected user.

## Programming a Java PAPI Client

The typical steps you have to follow when building a Java PAPI Client are:

- Import the required libraries.
- Create a process service.
- Create a process service session.
- Perform operations with PAPI.
- Close the process service.

## Import the Required Libraries

You need to import PAPI classes to be able to use them in your code. The following code imports the PAPI classes needed for this example.

```
import fuego.papi.CommunicationException;
import fuego.papi.InstanceInfo;
import fuego.papi.ProcessService;
import fuego.papi.ProcessServiceSession;
import fuego.papi.OperationException;
```

## Create a Process Service

In order to create a [ProcessService](#) you need a [java.util.Properties](#) object containing its configuration. You can create this property object and build it within your Java code, or you can load it from a properties file. This example adds the properties within the code for practical reasons.

The two mandatory properties you need to specify are the directory id and the path to the `directory.xml` file.

```
Properties configuration = new Properties();
properties.setProperty(ProcessService.DIRECTORY_ID, "default");
properties.setProperty(ProcessService.DIRECTORY_PROPERTIES_FILE, "directory.xml");
```

To create a [ProcessService](#) object you need to invoke the factory method [ProcessService.create\(\)](#) from the class `ProcessService` passing it the Property object as an argument.

If there is a problem locating the Directory Service, this method throws a [CommunicationException](#), so you need to call it inside a try-catch block.

```
try {
    ProcessService processService = ProcessService.create(configuration);
    //...
} catch (CommunicationException e) {
    System.out.println("Could not connect to Directory Service");
    e.printStackTrace();
}
```

## Create a Process Service Session

To create a [ProcessServiceSession](#) you must invoke the factory method [createSession](#) over the `ProcessService` object you've just created. This methods requires three String arguments:

- user: an existing participant in the Directory Service.
- password: the participant's password.
- host: the host from which the connection is established. This is an optional argument, it is used for monitoring purposes, so if this information is not available this argument's value can be null.

If there is a problem authenticating the specified participant, this method throws an [OperationException](#), so you need to invoke it inside a try-catch block.

```
try {
    //...
    ProcessServiceSession session = processService.createSession("user",
"password", "host");
```



```
//...
} catch (OperationException e) {
    System.out.println("Could not perform the requested operation");
    e.printStackTrace();
}
```

### Perform Operations with PAPI

The following code retrieves a list of available processes by invoking the method `processesGetIds()` over a `ProcessServiceSession` object.

It then iterates over them using those ids to obtain the process instances for each process by invoking the method `processGetInstances()` over the session object. If there is a problem performing any of the requested operations, this method throws an `OperationException`, so you need to invoke it inside a try-catch block.

Finally it iterates over those instances invoking the method `getId()` and prints its result.

```
try {
    //...
    System.out.println("Show Instances by process:");
    for (String processId : session.processesGetIds()) {
        System.out.println("\n Process: " + processId);
        for (InstanceInfo instance : session.processGetInstances(processId)) {
            System.out.println(" -> " + instance.getId());
        }
    }
} catch (OperationException e) {
    System.out.println("Could not perform the requested operation");
    e.printStackTrace();
}
```

### Close the Process Service Session

To close the session, invoke the method `close()` over the `ProcessServiceSession` object.

```
session.close();
```

Closing a session releases all the resources this session is using. After calling the method `close()`, the session can no longer be used. If you try to invoke a method on a closed session, its execution fails and a `SessionClosedException` is thrown.

### Close the Process Service

To close the `ProcessService` object, invoke the method `close()` over the `ProcessService` object.

```
processService.close();
```

This releases all the resources used by PAPI, clears the caches, deletes the temporary files, and closes the connections to the Process Engine and the Directory Service.

### Complete Code Example

The following class contains all the steps described in this section.

```

package papidoc.examples;

import fuego.papi.CommunicationException;
import fuego.papi.InstanceInfo;
import fuego.papi.ProcessService;
import fuego.papi.ProcessServiceSession;
import fuego.papi.OperationException;
import java.util.Properties;

public class PapiExample {

    public static void main(String[] args) {

        //////////////// API Initialization ////////////////
        Properties configuration = new Properties();
        configuration.setProperty(ProcessService.DIRECTORY_ID, "default");
        configuration.setProperty(ProcessService.DIRECTORY_PROPERTIES_FILE,
"directory.xml");
        configuration.setProperty(ProcessService.WORKING_FOLDER, "/tmp");

        try {
            ProcessService processService = ProcessService.create(configuration);

            //////////////// Establish a session ////////////////
            ProcessServiceSession session = processService.createSession("test",
"test", "host");

            //////////////// Operate with PAPI ////////////////
            for (String processId : session.processesGetIds()) {
                System.out.println("\n Process: " + processId);
                for (InstanceInfo instance :
session.processGetInstances(processId)) {
                    System.out.println(" -> " + instance.getId());
                }
            }

            //////////////// Close the session ////////////////
            session.close();

            //////////////// Release API Resources ////////////////
            processService.close();
        } catch (CommunicationException e) {
            System.out.println("Could not connect to Directory Service");
            e.printStackTrace();
        } catch (OperationException e) {
            System.out.println("Could not perform the requested operation");
            e.printStackTrace();
        }
    }
}

```

The following sequence diagram shows the interaction between the classes used in the example.

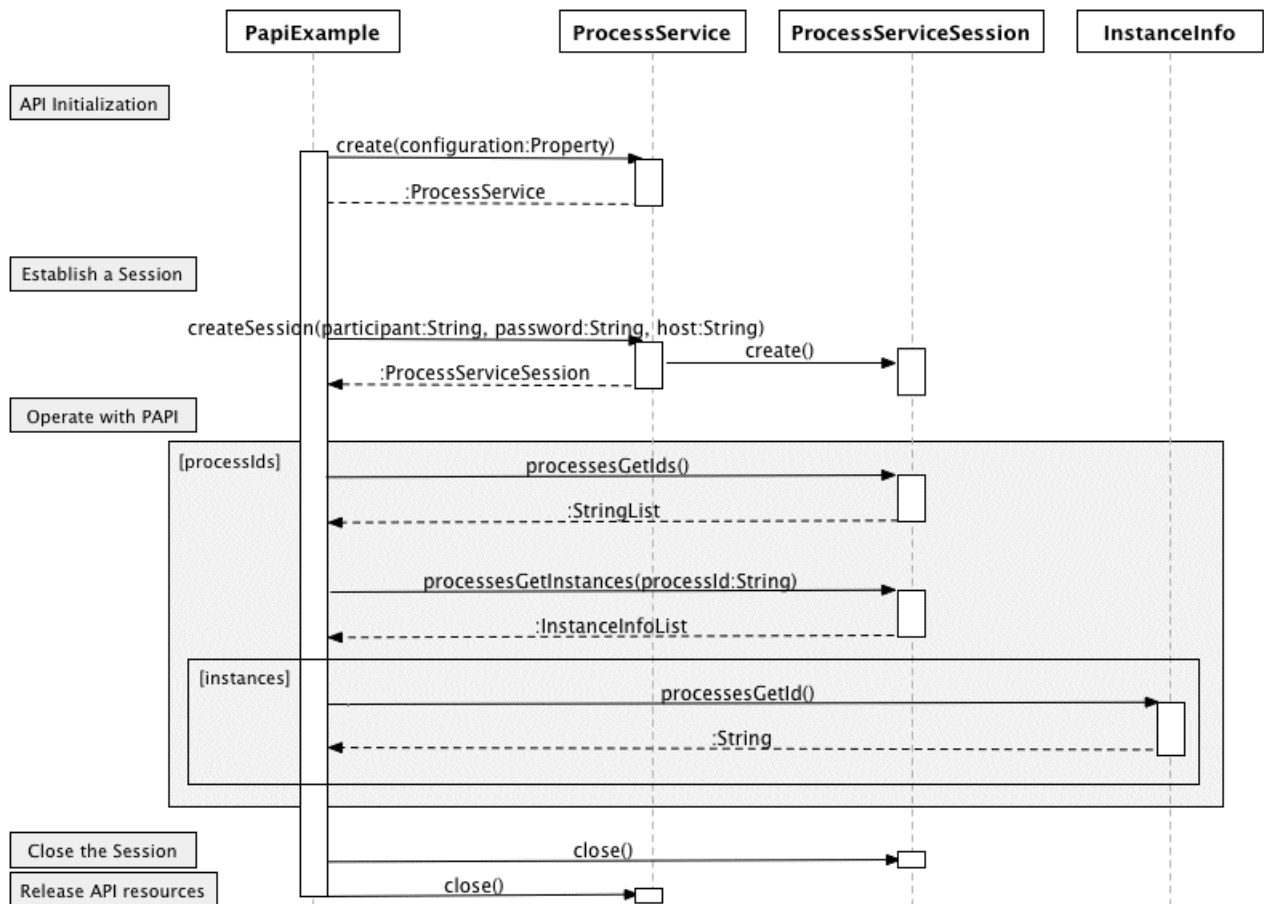


Figure 2: PAPI Client Example Sequence Diagram

## Compiling a Java PAPI Program

The following procedures show you how to compile from the command line a Java class that uses PAPI.

To compile a Java PAPI program from the command line you need to have a Java SE Development Kit 5 (JDK 5) installed. You can download the JDK from [Sun Developer Network](#). You may also use the JDK bundled with some installations of Oracle BPM Enterprise, which gets installed under `BEA_HOME/albpm6.0/enterprise/jre/`.

PAPI classes are contained in the `fuegopapi-client.jar` JAR file, which is distributed with Oracle BPM Enterprise under `BEA_HOME/albpm6.0/enterprise/client/papi/lib/fuegopapi-client.jar`.

1. Open a command-line session.
2. Add the PAPI library to the classpath by setting the environment variable `CLASSPATH`. The way of doing this depends on your operating system.

- Linux:

```
$export
CLASSPATH="/bea/albpm6.0/enterprise/client/papi/lib/fuegopapi-client.jar"
```

- Windows:

```
C:>set
CLASSPATH="C:/bea/albpm6.0/enterprise/client/papi/lib/fuegopapi-client.jar"
```

3. Compile the Java PAPI program using javac (Java Compiler) provided by the JDK:

```
javac MyFirstPapiApplication.java
```

These two steps can be merged into one by using the -classpath option when calling the java compiler:

```
javac -classpath
"C:/bea/albpm6.0/enterprise/client/papi/lib/fuegopapi-client.jar"
MyFirstPapiApplication.java
```

A file with the extension .class is generated in the directory where you compiled your program.

## Running a Java PAPI Program

The following procedures show you how to run from the command line a Java program that uses PAPI.

To run a Java PAPI program you need to have a Java SE Development Kit 5 (JDK 5) installed. You can download the JDK from [Sun Developer Network](#). You may also use the JDK bundled with some installations of Oracle BPM Enterprise, which gets installed under <ORABPM\_HOME>/jre/.

PAPI classes are contained in the fuegopapi-client.jar JAR file, which is distributed with Oracle BPM Enterprise under <ORABPM\_HOME>/client/papi/lib/fuegopapi-client.jar.

1. Open a command-line session.
2. If you are not running the program from the same command-line session where you have compiled it, you need to add the PAPI library to the Java classpath by setting the environment variable CLASSPATH. The way of doing this depends on your operating system.

- Linux:

```
$export
CLASSPATH="/OraBPMEnterpriseHome/client/papi/lib/fuegopapi-client.jar"
```

- Windows:

```
C:>set
CLASSPATH="C:/OraBPMEnterpriseHome/client/papi/lib/fuegopapi-client.jar"
```

3. Copy the file directory.xml to the location specified in your properties file or in your PAPI program. The file directory.xml resides in the directory <ORABPM\_HOME>/conf. In the analyzed example the directory.xml file was copied to the directory from where the example is run. This location is specified in the following lines of code:

```
Properties configuration = new Properties();
//...
configuration.setProperty(ProcessService.DIRECTORY_PROPERTIES_FILE,
"directory.xml");
```

4. Run your PAPI program using the `java` command provided by the JDK:

```
java MyFirstPapiApplication
```

This step and step one can be merged into one by using the `-classpath` option when calling the `java` command:

```
java -classpath "C:/OraBPMEnterpriseHome/client/papi/lib/fuegopapi-client.jar"
MyFirstPAPIApplication
```

When you run the program you see a list of the deployed processes and their instances. The following lines illustrate the generated output when executing this program connecting to an engine that has three instances sitting on the deployed process "Process":

```
Process: /Process#Default-1.0
-> /Process#Default-1.0/1/0
-> /Process#Default-1.0/3/0
-> /Process#Default-1.0/2/0
```

## Connecting to an Engine Running in Studio

You use a PAPI client with an engine running in Studio to debug and test the project you are developing.



**Note:** The PAPI client and Studio application must run in the same host.

When configuring the connection properties, replace the property

`ProcessService.DIRECTORY_PROPERTIES_FILE` for the property `ProcessService.PROJECT_PATH` and assign it the file-path to your project.

The following code shows you how to use this property:

```
Properties config = new Properties();
config.setProperty(ProcessService.DIRECTORY_ID, "default");
config.setProperty(ProcessService.PROJECT_PATH, "../BPMWorkspace/PapiTest" );
```



**Note:** When connecting to an Engine in Studio, the operations to store views and presentations are not available.

## Configuring the Log of a PAPI Client

The following procedure shows you how to configure the log of your PAPI client.

To configure the log of your PAPI client.

1. Create a properties file that contains the properties to configure your PAPI client log.

Create this file in a location that is accessible to your PAPI Client.

For a list of the properties you can include, see [PAPI Client Log Properties](#) on page 14 .

2. In PAPI client code add the necessary code to create a `Properties` object and load it using the properties file you created.

3. Modify the code in your PAPI client so that it uses the `ProcessService` constructor that receives a properties file as an argument.
4. Compile your PAPI client.

For information about how to compile a PAPI client, see [Compiling a Java PAPI Program](#) on page 11 .

The next time you run your PAPI client it logs using the properties you defined.

## PAPI Client Log Properties

The following table describes the properties you can use to configure the log of your PAPI client.

Property	Description	Default Value
fuego.papi.application	Specifies the name of the application.	papi
fuego.log.<app_name>.file	Specifies the file where to redirect the log.	By default it logs to standard error.
fuego.log.<app_name>.severities	Defines the severity level of the log message. For more information about security levels, see <a href="#">Log Severity Levels</a> on page 25.	WARNING
fuego.log.<app_name>.detailLevel	Specifies whether to log messages. If set to 0 Oracle BPM does not log any message, if set to 1 it logs all messages.	1
fuego.log.<app_name>.format	Specifies the log format for the first line of a log message.	[<{SEV}> 'dd/MM/yy HH:mm:ss'] {INDENT}{MSG}
fuego.log.<app_name>.continuationFormat	Specifies the log format for lines that follow the first line of a log message.	{INDENT}{MSG}

## Log Severity Levels

Oracle BPM allows you to define logging levels to specify the level of detail of the information stored in the Oracle BPM logs.

Log Level	Description
Fatal	Specifies a serious error that may cause the application to fail.
Severe	Specifies a serious error that may or may not cause the application to fail.
Warning	Specifies a potentially harmful situation but generally does not pose a threat to the stability of an application.
Info	<p>Specifies informational messages that highlight the progress of the application at a high level. These can include:</p> <ul style="list-style-type: none"> <li>• Changes in the engine state, including: start, stop, and restart.</li> <li>• Changes in state of engine services.</li> <li>• Changes in engine properties.</li> <li>• Changes in the state of a process deployed on the engine, including: startup, deployment, redeployment, and deprecation.</li> <li>• Actions of participants</li> </ul>

Log Level	Description
Debug	<ul style="list-style-type: none"> <li>• Work executed by the engine automatically.</li> </ul> <p>Specifies informational messages that highlight the process instances at a lower level. These can include:</p> <ul style="list-style-type: none"> <li>• Tracing a process instance, including: instance creation, changing activities, routing, and locks.</li> <li>• Changes in the state of an instance, including: running, selection, activity completion, and exceptions.</li> <li>• Actions on a process, including: executing a task, executing an activity, and executing a ToDo Item.</li> </ul>

## Running a PAPI Client in a J2EE Distributed Environment

If your PAPI Java client and the J2EE Process Engines it connects to run in different locations then you must provide the PAPI client the information to connect to the remote Process Engine. You must provide this information using standard JNDI properties.

You must configure JNDI process when the J2EE Process Execution Engine is running in an application server, and the Java PAPI client is running in:

- a different applications server
- a different cluster
- a different WebLogic domain
- a different WebSphere cell
- an external servlet container
- a standalone environment

The Java PAPI client needs information to locate a J2EE Engine that is running in a different location. You must provide the PAPI client this information using JNDI properties. The PAPI client uses these JNDI properties to create a Context and connect to the application server where the J2EE Engine is running.

PAPI requires you to specify the following properties:

- `java.naming.factory.initial`
- `java.naming.provider.url`

These are standard JNDI properties. For information about these properties, see your application server documentation. For WebLogic Server, see the *Programming WebLogic JNDI* document

If your PAPI client connects to a single Process Engine you can specify this properties using system properties, or using a file. To specify this properties using a file follow the procedures described in [Configuring JNDI for a Single J2EE Process Execution Engine](#) on page 16.

If your PAPI client needs to connect to multiple Process Engines then you have to provide it the information to connect to them using a properties file for each of the engines. To do this follow the procedures described in [Configuring JNDI for Multiple J2EE Engines](#) on page 17.

If your application server requires that you specify other properties then you can specify them in the same properties file. The PAPI client uses all the properties defined in the properties file to create a Context to connect to the application server.

## Configuring JNDI for a Single J2EE Process Execution Engine

The following procedure shows you how to configure your PAPI Java client to connect to a single J2EE Process Execution Engine running in a remote location, using JNDI.

To configure the JNDI properties you need to access a remote J2EE Process Engine from your PAPI client:

1. Create a properties file that contains the JNDI properties PAPI needs to connect to the remote J2EE Process Engine.

You must specify the following basic properties:

- java.naming.factory.initial
- java.naming.provider.url

 **Note:** If your remote server needs additional properties to connect using JNDI, specify this properties as well.

2. Copy this file to a location that is accessible to your PAPI Client.
3. Provide the PAPI client access to the file that contains the JNDI properties. You can set this property in the code of the PAPI client using system properties, or you can pass it as an argument using the `-D` option of the java command.

Property	Description
<b>fuego.j2ee.initialctx.file</b>	Specifies the name of file that contains the JNDI properties to connect to the application server where the specified J2EE Process Engine runs. You must specify the value of this property using the absolute path of the file.
<b>fuego.j2ee.initialctx.resource</b>	Specifies the name of the resource that contains the JNDI properties to connect to the application server where the specified J2EE Process Engine runs. PAPI obtains the resource using the method <code>getResource()</code> of the context <code>ClassLoader</code> . You must add this resource to the <code>CLASSPATH</code> .
<b>fuego.j2ee.initialctx.url</b>	The URL of the file that contains the jndi properties to connect to the application server where J2EE Process Engine runs.

The following example shows you how to specify the file that contains the jndi properties to connect to the server where the default J2EE Process Engine runs, using the property `fuego.j2ee.initialctx.file`.

- Using system properties in the Java code of the PAPI client:

```
System.setProperty("fuego.j2ee.initialctx.file",  
"C:\\\\engine.properties");
```

- Using the option `-D` when running the PAPI client:

```
java -Dfuego.j2ee.initialctx.file=C:\\\\engine.properties
```

The following example shows you how to specify the resource that contains the jndi properties to connect to the server where the default J2EE Process Engine runs, using the property `fuego.j2ee.initialctx.resource`.

- Using system properties in the Java code of the PAPI client:

```
System.setProperty("fuego.j2ee.initialctx.resource",  
"engine.properties");
```

- Using the option `-D` when running the PAPI client:

```
java -Dfuego.j2ee.initialctx.resource=engine.properties
```



The following example shows you how to specify the URL that contains the jndi properties to connect to the server where the default J2EE Process Engine runs, using the property `fuego.j2ee.initialctx.url`.

- Using system properties in the Java code of the PAPI client:

```
System.setProperty("fuego.j2ee.initialctx.url",
    "http://server/conf/engine.properties");
```

- Using the option `-D` when running the PAPI client:

```
java
-Dfuego.j2ee.initialctx.url=http://server/conf/engine.properties
```

## Configuring JNDI for Multiple J2EE Engines

The following procedure shows you how to configure your PAPI Java client to connect to a specific J2EE Process Engine running in a remote location, using JNDI.

Follow this procedure for each of the J2EE Process Engines that you need to access from your PAPI client:

1. Create a properties file that contains the JNDI properties PAPI needs to connect to the remote J2EE Process Engine.

You must specify the following basic properties:

- `java.naming.factory.initial`
- `java.naming.provider.url`



**Note:** If your remote server needs additional properties to connect using JNDI, specify this properties as well.

2. Copy this file to a location that is accessible to your PAPI Client.
3. Provide the PAPI client access to the file that contains the JNDI properties. You can set this property in the code of the PAPI client using system properties, or you can pass it as an argument using the `-D` option of the java command.

Property	Description
<b><code>fuego.j2ee.initialctx.ENGINE_ID.file</code></b>	Specifies the name of file that contains the JNDI properties to connect to the application server where the specified J2EE Process Engine runs. You must specify the value of this property using the absolute path of the file.
<b><code>fuego.j2ee.initialctx.ENGINE_ID.resource</code></b>	Specifies the name of the resource that contains the JNDI properties to connect to the application server where the specified J2EE Process Engine runs. PAPI obtains the resource using the method <code>getResource()</code> of the context <code>ClassLoader</code> . You must add this resource to the <code>CLASSPATH</code> .
<b><code>fuego.j2ee.initialctx.ENGINE_ID.url</code></b>	The URL of the file that contains the jndi properties to connect to the application server where J2EE Process Engine runs.

The following example shows you how to specify the file that contains the jndi properties to connect to the server where the J2EE Process Engine "engine1" runs, using the property `fuego.j2ee.initialctx.ENGINE_ID.file`.

- Using system properties in the Java code of the PAPI client:

```
System.setProperty("fuego.j2ee.initialctx.engine1.file",
"C:\\engine1.properties");
```

- Using the option -D when running the PAPI client:

```
java -Dfuego.j2ee.initialctx.engine1.file=C:\\engine1.properties
```

The following example shows you how to specify the resource that contains the jndi properties to connect to the server where the J2EE Process Engine "engine1" runs, using the property `fuego.j2ee.initialctx.ENGINE_ID.resource`.

- Using system properties in the Java code of the PAPI client:

```
System.setProperty("fuego.j2ee.initialctx.engine1.resource",
"engine1.properties");
```

- Using the option -D when running the PAPI client:

```
java -Dfuego.j2ee.initialctx.engine1.resource=engine1.properties
```

The following example shows you how to specify the URL that contains the jndi properties to connect to the server where the J2EE Process Engine "engine1" runs, using the property `fuego.j2ee.initialctx.ENGINE_ID.url`.

- Using system properties in the Java code of the PAPI client:

```
System.setProperty("fuego.j2ee.initialctx.engine1.url",
"http://server/conf/engine1.properties");
```

- Using the option -D when running the PAPI client:

```
java
-Dfuego.j2ee.initialctx.engine1.url=http://server/conf/engine1.properties
```

# Oracle BPM PAPI Web Service

---

This guide contains relevant information about PAPI Web Service architecture, an analysis of the structure of PAPI Web Service clients written in different languages, and procedures that show you how to modify PAPI Web Service configuration.

## PAPI Web Service Overview

PAPI Web Service is an independent web application built on top of PAPI. This application exposes a subset of PAPI functionality using SOAP over HTTP.

Using PAPI Web Service to communicate with the Engine has the following advantages over using PAPI:

- You can use it from any programming language that supports XML and HTTP.
- It does not need any external libraries on the client side.
- The application using PAPI Web Service does not need a connection to the Directory Server. This application can run outside the domain where Oracle BPM is installed.

There are a few minor disadvantages:

- PAPI Web Service is a layer on top of PAPI. The web services client communicates with PAPI using XML. This adds a small overhead that makes PAPI Web Service slightly less performant than using PAPI directly.
- Attachments functionality is not available.
- PAPI Web Service does not handle complex types. Only methods with either primitive or catalogued XML schema type arguments and primitive return type can be invoked.

## What's New in PAPI Web Service 2.0?

This section describes the features supported by PAPI Web Service 2.0.

The following table shows the difference between PAPI Web Service 2.0 and PAPI Web Service 1.0:

PAPI Web Service 2.0	PAPI Web Service 1.0
Independent web application.	Bundled with Oracle BPM Workspace.
You can modify its default configuration.	Supports only the default configuration.
Document/literal wrapped style WSDL SOAP binding.	RPC/encoded style WSDL SOAP binding.
Supports WS Security Username Token Profile 1.1, HTTP Basic and Single Sign On (SSO) authentication mechanisms.	Does not support any standard authentication mechanisms. You need to send a session ID every time you invoke an operation.
Method arguments support catalogued schema type objects in addition to primitive types.	Method arguments only support primitive types.
The signature of the exposed methods matches the signature of their equivalent methods in PAPI.	Its semantics are completely different from PAPI's semantics.



**Note:** PAPI Web Service 1.0 is deprecated. If you still need to use it in Oracle BPM 6.0 you have to enable and start Oracle BPM Classic WorkSpace.

## PAPI Web Service Usage Scenarios

PAPI Web Service provides access to a considerable subset of PAPI operations. This section describes the scenarios where PAPI Web Service is more suitable than PAPI.

PAPI Web Service complies with Web Services standards. This allows you to take advantage of the existing common infrastructure used by other applications such as load balancers, proxies, security services and monitoring. PAPI Web Service fits perfectly into a SOA architecture.

Use PAPI Web Service to expose PAPI operations to:

- external applications written in virtually any programming language.
- applications running outside the domain where Oracle BPM resides.
- applications running behind a fire wall.

## PAPI Web Service Architecture Overview

PAPI Web Service is a web service application that exposes a considerable set of PAPI operations.

PAPI Web Service is an independent web application that runs on top of PAPI. PAPI Web Service provides a WSDL (Web Services Definition Language) descriptor that defines the operations the client can invoke and the complex types these operations may use. The client application connected to PAPI Web Service, uses SOAP (Simple Object Access Protocol) over HTTP to invoke any of the functions listed in the WSDL.

PAPI Web Service relies on PAPI to obtain the information the client requests. Then it translates this information into XML and uses SOAP to send it back to the client.

PAPI Web Service implementation is based on the following:

- JAX-WS 2.0 web service
- WS-I 1.1 compliant
- Document/literal wrapped style WSDL SOAP binding

The following diagram shows the interaction between PAPI Web Service components during runtime.

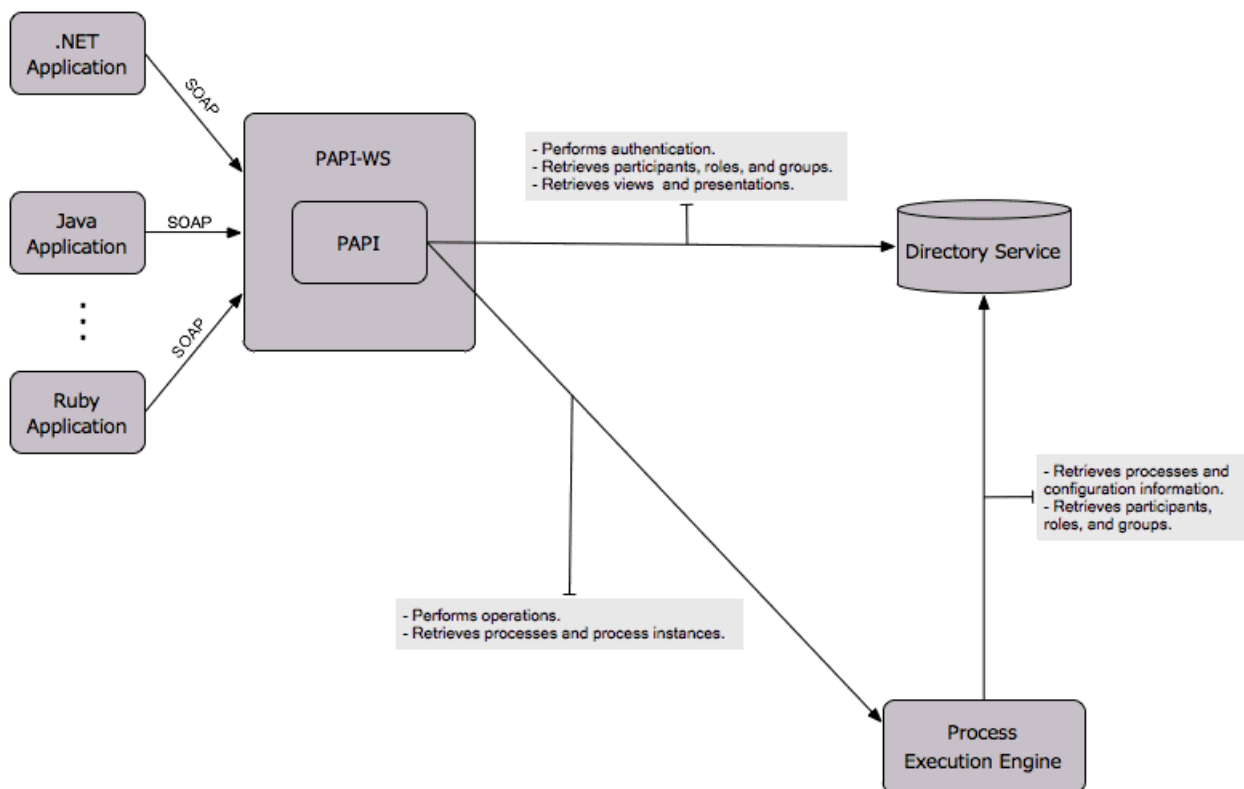


Figure 3: PAPI Web Service Runtime Architecture

## Enabling PAPI Web Service

PAPI Web Service is not enabled by default. This section shows the how to enable PAPI Web Service for each of the possible environments and configurations. These procedures depend on the type and configuration of Oracle BPM installation.

### Enabling PAPI Web Service in Oracle BPM Studio

The following procedures show you how to enable PAPI Web Service in Studio.

By default PAPI Web Service application is not enabled. To enable PAPI Web Service in Studio:

1. Right-click on the project.
2. Select **Engine Preferences**.
3. Select **Advanced**.
4. Check **Start PAPI Web Services**.

The next time you start the engine, PAPI Web Service application is started. To verify this, launch PAPI Web Service Console.

See [Launching PAPI Web Service Console in Oracle BPM Studio](#) on page 24

### Enabling PAPI Web Service in Oracle BPM Enterprise

The following procedures show you how to enable PAPI Web Service in Oracle BPM Enterprise.

To enable PAPI Web Service:

1. Launch Oracle BPM Admin Center.
2. Click **Configuration**.
3. Select **BPM Web Applications** tab.
4. Select **PAPI Web Services** check-box in the list of BPM web applications to run.

The next time you start the engine, PAPI Web Service application is started. To verify this, launch PAPI Web Service Console.

See [Launching PAPI Web Service Console in Oracle BPM Enterprise](#) on page 24.

## Installing PAPI Web Service on a J2EE Application Server

To install PAPI Web Service when the Process Execution Engine is running on a J2EE application server, you have to follow the procedures that describe how to install an Oracle BPM web application on that specific server. This section shows the procedures for WebLogic Application Server and WebSphere Application Server.

### Installing PAPI Web Service on WebLogic Server

The following procedures show you how to install PAPI Web Service on WebLogic Server

1. Build PAPI Web Service Application.

For information on how to build a web application on WebLogic Server, see steps 1 to 4 from [Build and Deploy Applications \(.ear\)](#) on page 22.

This generates two ear files, that correspond to the two supported versions of WebLogic.

2. Choose the ear file that corresponds to the version of the WebLogic Server you are using.

The following table shows the correspondence between the version of WebLogic Server and the generated ear file.

WebLogic Server Version	ear File
WebLogic Server 10	07-papiws-XAFDIDS.ear
WebLogic Server 9.2	07-papiws-wls92-XAFDIDS.ear

3. Deploy PAPI Web Service Application.

For information on how to deploy a web application on WebLogic Server, see step 5 from [Build and Deploy Applications \(.ear\)](#) on page 22.

### Build and Deploy Applications (.ear)

The Oracle BPM Process Administrator allows you to bundle the Oracle BPM applications as .ear files for installation on WebLogic.

Before creating the Oracle BPM application archives, you must have an Oracle BPM Engine for WebLogic configured.



1. Login to Oracle BPM Process Administrator. By default, it runs on `http://host:8686/webconsole`.
2. Click on **Engines** and then click on the name of your Oracle BPM Engine for WebLogic.  
You should see the configuration properties for your Engine.
3. Click on the **Basic Configuration** tab and then on **J2EE Application Server Files**.

This page allows you to re-create the .ear files of those Oracle BPM applications associated with this Engine.



**Note:** When you access this page, the Process Administrator gets the status of each of the applications by contacting Oracle BPM Deployer. You receive a warning message at the bottom of the page if there is a problem contacting Oracle BPM Deployer. If this is the case, make sure the **BPM Application**

**Deployer URL** (within the **Application Server** tab) is correct and that Oracle BPM Deployer is up and running on WebLogic.

4. Click on the "new" icon () next to each of the applications you want to install.
5. Click on the "install" icon () next to each of the applications you want to install.



**Attention:** This may take several minutes. Do not click any link on the page or click the **back** button in your browser until the page is automatically reloaded. When you click on the icon, Oracle BPM Process Administrator transfers the file over to WebLogic's Deployment Manager (by means of Oracle BPM Deployer) and then WebLogic goes through the application installation process.

### Installing PAPI Web Service on WebSphere Application Server

The following procedures show you how to install PAPI Web Service on WebSphere Application Server.

To install PAPI Web Service on WebSphere Application Server:

1. Build and deploy PAPI Web Service application.

For information on how to build and deploy Oracle BPM applications on WebSphere Application Server, see *"WAS Basic Configuration, Deploy Oracle BPM Apps in WebSphere" in Oracle BPM Configuration Guide, WebSphere Edition*.

2. Open WebSphere Console.
3. Choose **Applications ► Enterprise Applications**.
4. Click **07-papiws-FDIDS.ear** link.
5. Select **Class loading and update detection**.
6. Select **Classes loaded with application class loader first**.
7. Select **Single class loader for application**.
8. Click **OK**.

If you do not enter a value in the field labeled **Polling interval for updated files** an error message appears.

A message asking you to confirm your changes appears.

9. Click **Save**.
10. Restart the server.

The next time you start the server, PAPI Web Service application starts. To verify this, launch PAPI Web Service Console.

## PAPI Web Service Configuration

You can configure PAPI Web Service by modifying a set of properties either by using the provided user interface or by editing the file where these properties are stored. This section shows you how to modify PAPI Web Service configuration in Oracle BPM Studio and Enterprise.

### PAPI Web Service Configuration Console

PAPI Web Service provides a console where you can view its configuration properties and other useful information such as the endpoint and the WSDL URLs.

PAPI Web Service console is available in Oracle BPM Enterprise and Studio. The console enables you to edit the PAPI Web Service configuration.

The information displayed in the PAPI Web Service console comprises the following:

- **Style:** shows the format that the WSDL defines for the SOAP messages sent between the web service and the client. PAPI Web Service uses document/literal wrapped format. You cannot change this style.
- **SSO:** shows if Single Sign On authentication is enabled.
- **WS-Security Username Token Profile Authentication:** shows if Username Token Profile authentication is enabled.
- **HTTP Basic Authentication:** shows if HTTP Basic authentication is enabled.
- **PRESET Authentication:** shows if PRESET authentication is enabled. This type of authentication is valid only for Oracle BPM Enterprise. In Oracle BPM Studio the value of this property is always false, and it cannot be changed.
- **Endpoint:** shows the URL of PAPI Web Service endpoint.
- **WSDL:** shows the URL where PAPI Web Service WSDL is published. Most web services stacks include a tool to automatically generate stubs based on a WSDL. You need to provide this tool with the WSDL URL displayed here.

### Launching PAPI Web Service Console in Oracle BPM Studio

The following procedure shows you how to launch PAPI Web Service console in Oracle BPM Studio.

To launch PAPI Web Service console:

1. Enable PAPI Web Service in an already existing project.  
See [Enabling PAPI Web Service in Oracle BPM Studio](#) on page 21.
2. Start the Process Engine.
3. Choose **Run ► Launch PAPI Web Services**.

The default browser opens showing Oracle BPM Web Service console.

### Launching PAPI Web Service Console in Oracle BPM Enterprise

The following procedure shows you how to launch PAPI Web Service console in Oracle BPM Enterprise.

1. Start Oracle BPM Admin Center.
2. Enable PAPI Web Service.  
See [Enabling PAPI Web Service in Oracle BPM Enterprise](#) on page 21
3. Click **Start BPM Web Applications**.
4. Click **Launch PAPI Web Services Console**.

The default browser opens showing Oracle BPM Web Service console.

## Editing PAPI Web Service Configuration

This section shows you how to change PAPI Web Service configuration in Studio and in Enterprise.

The way of editing PAPI Web Service configuration varies between both types of installation.

In an Enterprise installation PAPI Web Service's configuration is stored in the `papiws.properties` file, located under `<ORABPM_HOME>/webapps/papiws/WEB-INF`. This file contains additional advanced properties that you can use to tune PAPI Web Service performance. Each property has a comment that describes their function.

### Editing PAPI Web Service Configuration in Studio

The following procedure shows you how to edit PAPI Web Service configuration in Studio.

To edit PAPI Web Service configuration:

1. Launch PAPI Web Service console  
See [Launching PAPI Web Service Console in Oracle BPM Studio](#) on page 24.



2. Click **Change configuration**.  
The displayed properties become editable and a **Save changes** button appears next to the **Change configuration** button.
3. Modify the values of the properties you need to change.
4. Click **Save changes**.  
A message informing changes were successfully applied appears.
5. Restart the engine to apply changes.

Launch PAPI Web Service console to verify your changes were applied.

### Editing PAPI Web Service Configuration in Oracle BPM Enterprise

The following procedures show you how to edit PAPI Web Service configuration in an Oracle BPM Enterprise.

To edit PAPI Web Service configuration:

1. Start Oracle BPM Admin Center.
2. Modify the values of the properties you need to change.
3. Click **OK**.
4. Click **Start BPM Web Applications** to apply the changes.

Click **Launch PAPI Web Services Console** to verify your changes were applied.

## Configuring PAPI Web Service Log

PAPI Web Service keeps a log of the performed operations that can be used for troubleshooting. The following procedure shows you how to configure the directory where log files are stored, and the severity levels to which you can filter the logged messages.

To enable the log for PAPI Web Service application:

1. Start Oracle BPM Admin Center.
2. Click **Configuration**.
3. Select **PAPI Web Services** tab.
4. Enter the complete path of the directory where you want to save PAPI Web Service logs in the **Log Folder** field, or click **Browse...** and select the directory.
5. Select a severity level from the **Log Message Severity Level** drop-down list.

The available severity levels are:

- Debug
- Info
- Warning
- Severe
- Fatal

For more information about severity levels, see [Log Severity Levels](#) on page 25.

The next time you start PAPI Web Service the changes made to the log configuration are applied.

### Log Severity Levels

Oracle BPM allows you to define logging levels to specify the level of detail of the information stored in the Oracle BPM logs.

Log Level	Description
Fatal	Specifies a serious error that may cause the application to fail.

Log Level	Description
Severe	Specifies a serious error that may or may not cause the application to fail.
Warning	Specifies a potentially harmful situation but generally does not pose a threat to the stability of an application.
Info	<p>Specifies informational messages that highlight the progress of the application at a high level. These can include:</p> <ul style="list-style-type: none"> <li>• Changes in the engine state, including: start, stop, and restart.</li> <li>• Changes in state of engine services.</li> <li>• Changes in engine properties.</li> <li>• Changes in the state of a process deployed on the engine, including: startup, deployment, redeployment, and deprecation.</li> <li>• Actions of participants</li> <li>• Work executed by the engine automatically.</li> </ul>
Debug	<p>Specifies informational messages that highlight the process instances at a lower level. These can include:</p> <ul style="list-style-type: none"> <li>• Tracing a process instance, including: instance creation, changing activities, routing, and locks.</li> <li>• Changes in the state of an instance, including: running, selection, activity completion, and exceptions.</li> <li>• Actions on a process, including: executing a task, executing an activity, and executing a ToDo Item.</li> </ul>

## PAPI Web Service Security Authentication

This section describes the different types of authentication mechanisms that PAPI Web Service supports.

PAPI Web Service supports the following types of authentication:

- Custom Single Sign On (SSO) authentication
- UsernameToken Profile 1.1 (Plain-text)
- HTTP Basic authentication
- PRESET authentication

You can independently enable or disable any of these authentication mechanisms.



**Note:** By default, Username Token Profile authentication is selected. You must select at least one authentication method to provide PAPI Web Service the necessary information to authenticate against the engine.

When PAPI Web Service starts running, it activates the authentication providers that correspond to the enabled authentication mechanisms.

Every time a client makes a request to PAPI Web Service, this request goes through an authentication phase before reaching the service endpoint. During this phase the activated authentication providers will be called in the order they appear in the preceding list. When one of these providers successfully authenticates the request, the application grants access to the web service. If all the activated providers reject access, the request is rejected.

## Developing a Custom Sign On Implementation

Papi Web Service can use a custom Single Sign On (SSO) implementation to authenticate the client. The following procedures show you how to develop a custom SSO implementation for PAPI Web Service.

To compile the class containing your custom SSO implementation you need to have a Java SE Development Kit 5 (JDK 5) installed. You can download the JDK from [Sun Developer Network](#).

To configure SSO Authentication:

Implement the interface `fuego.sso.SSOUserLoginInterface`.

a) Add the file `fuego.core.jar` to the CLASSPATH.

This file resides in `<ORABPM_HOME>/lib`.

b) Create a Java class that implements the interface `fuego.sso.SSOUserLoginInterface`.

This class should contain your custom SSO implementation.

c) Compile the class created in the previous step.

## Configuring Custom Single Sign-On Authentication

The following procedure shows you how to configure your web application to use a Custom Single Sign-On implementation.

The following procedure assumes that you have developed and compiled a class that implements the corresponding SSO interface.

To configure your web application to use your custom Single Sign-On implementation:

1. Copy the compiled class that contains your SSO implementation to the `WEB-INF/lib` directory of the web application.

This directory is located under `<ORABPM_HOME>/webapps/<Web_Application_Name>`.

2. Edit the web application configuration and select the SSO option.

3. Enter the fully qualified name of the class containing the SSO implementation.

The next time you start the web application, SSO authentication is activated.

## Configuring Username Token Profile

PAPI Web Service can use Web Services Security Username Token Profile to authenticate the client. The following procedures show you how to configure Username Token Profile for PAPI Web Service.

To configure Username Token Profile authentication:

- Edit PAPI Web Service configuration and select the Username Token Profile authentication option.

See [Editing PAPI Web Service Configuration](#) on page 24.

The next time you start PAPI Web Service application, Username Token Profile authentication is activated.

- Configure your web services client to send the Username Token SOAP header when it authenticates against PAPI Web Service.

The way of doing this depends on the programming language and the stack used to code your client.

For example, for a client using Java JAX-WS stack you need to add the following method, and invoke it before executing any operation.

```
import javax.xml.soap.SOAPElement;
import javax.xml.soap.SOAPException;
import javax.xml.soap.SOAPFactory;
import com.sun.xml.ws.api.message.Header;
```

```

import com.sun.xml.ws.api.message.Headers;
import com.sun.xml.ws.developer.WSBindingProvider;

//...

    private static final String SECURITY_NAMESPACE =
"http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd";

//...

private static void addUserNameTokenProfile(PapiWebService papiWebServicePort)

    throws SOAPException {
    SOAPFactory soapFactory = SOAPFactory.newInstance();
    QName securityQName = new QName(SECURITY_NAMESPACE, "Security");
    SOAPElement security = soapFactory.createElement(securityQName);
    QName tokenQName = new QName(SECURITY_NAMESPACE, "UsernameToken");
    SOAPElement token = soapFactory.createElement(tokenQName);
    QName userQName = new QName(SECURITY_NAMESPACE, "Username");
    SOAPElement username = soapFactory.createElement(userQName);
    username.addTextNode("test");
    QName passwordQName = new QName(SECURITY_NAMESPACE, "Password");
    SOAPElement password = soapFactory.createElement(passwordQName);
    password.addTextNode("test");
    token.addChildElement(username);
    token.addChildElement(password);
    security.addChildElement(token);
    Header header = Headers.create(security);
    ((WSBindingProvider) papiWebServicePort).setOutboundHeaders(header);
}

//...

```

## Configuring HTTP Basic Authentication

PAPI Web Service can use HTTP Basic authentication to authenticate the client. The following procedures show you how to configure HTTP Basic authentication for PAPI Web Service.

To configure HTTP Basic authentication:

1. Edit PAPI Web Service configuration and select the HTTP Basic authentication option.

See [Editing PAPI Web Service Configuration](#) on page 24

The next time you start PAPI Web Service application, HTTP Basic authentication is activated.

2. Configure your web services client to use HTTP Basic authentication when it authenticates against PAPI Web Service.

The way of doing this depends on the programming language used to code your client.

For example, for a client using Java JAX-WS stack you need to add the following method, and invoke it before executing any operation.

```

import java.util.Map;
import javax.xml.ws.BindingProvider;

//...

    private static void addHttpBasicAuthentication(PapiWebService
papiWebServicePort) {

```

```

        Map<String, Object> requestContext = ((BindingProvider)
papiWebServicePort).getRequestContext();
        requestContext.put(BindingProvider.USERNAME_PROPERTY, "test");
        requestContext.put(BindingProvider.PASSWORD_PROPERTY, "test");
    }

    //...

```

## Configuring PRESET Authentication

A PRESET is a set of properties that you can define in a `directory.xml` file for different purposes. This mechanism of authentication is only available for Enterprise installations. The following procedures shows you how to configure PRESET authentication for PAPI Web Service.

To configure PRESET authentication:

1. Use the ant task `managedirectory`, to add a PRESET with a valid user and password to the `directory.xml` file that corresponds to the PAPI Web Service web application.

The `directory.xml` file for PAPI Web Service web application is located under `<ORABPM_HOME>/webapps/papiws/WEB-INF`. This file is a copy of the XML file named after the Directory Configuration name located under `<ORABPM_HOME>/webapps/conf`.

See [managedirectory ant task](#).

2. Edit PAPI Web Service configuration and enter the PRESET name in the field labeled "Set PRESET ID for PRESET authentication".

See [Editing PAPI Web Service Configuration](#) on page 24

The next time you start the PAPI Web Service application, PRESET authentication is activated.

## PAPI Web Service Examples

You can develop a PAPI Web Service client in different programming languages. Some languages may even provide more than one stack to develop a web service client. This section shows examples of PAPI Web Service clients developed in different languages and stacks.

### Java JAX WS Client Example

This section shows you how to develop a Java client using the JAX WS stack. It uses the PAPI Web Service to retrieve a list of process instances visible to the connected user.

This example contains an analysis of the code of a PAPI Web Service client developed using JAX WS. The source code includes:

- A set of ant scripts to generate the stubs from the WSDL, compile the code and run it
- A lib directory containing the external libraries needed to code and compile the JAX-WS client

You can use this project as a basis to develop more complex examples. To do this replace the class `PapiWsJaxWsExample` by the classes you develop, and change the target run in the ant script so that it executes the new class.

A Java JAX-WS client contains two different type of classes:

- JAX-WS portable artifacts
- Client Java classes

## JAX-WS portable artifacts

The web service client code uses these artifacts to operate with PAPI Web Service. JAX-WS provides a tool called Wsimport to generate these classes based on the WSDL PAPI Web Service. When you run Wsimport using PAPI Web Service WSDL as an input argument it generates the following classes:

- Service Endpoint Interface
- Service
- Exception classes
- Java classes mapped from the schema types referenced in the WSDL

This example uses Wsimport ant task to generate this artifacts. For information about Wsimport, see <https://jax-ws.dev.java.net/nonav/2.1.2/docs/wsimport.html>.

## Client Java Classes

A PAPI Web Service client includes one or more Java classes that contain the code to invoke PAPI Web Service and operate with it. You have to code these classes yourself. The code in these classes uses JAX-WS portable artifacts to access the web service and to operate with it.

The client shown in this example contains only one class because it is a simple example. The code in this class performs the following actions:

- Invokes the web service
- Authenticates using Username Token Profile and HTTP authentication
- Uses JAX-WS portable artifacts to obtain the list of process instances

For a detailed analyses of this class, see [JAX WS Client Main Class](#).

## Download

You can download the set of java classes of this example from [http://edocs.bea.com/albsi/docs60/resources/papi\\_ws/ALBPM-PapiWs-JaxWs-Example.zip](http://edocs.bea.com/albsi/docs60/resources/papi_ws/ALBPM-PapiWs-JaxWs-Example.zip). For information on how to run this example see [Running Java JAX WS Client Example](#) on page 35.

## JAX-WS Client Main -Class

This section analyzes the main-class of the JAX-WS client example step by step.

### Programming a JAX-WS Client

The typical steps you have to follow to use PAPI Web Service with JAX-WS stack are:

- Import the required libraries
- Initialize the web service client
- Configure authentication
- Operate with PAPI Web Service

### Import the Required Libraries

This example uses classes from `java.net`, `javax.xml` and `com.sun.xml.ws` packages. You have to import these classes to be able to use them in your code. The following code imports the classes from these packages that are used in this example.

```
import java.net.MalformedURLException;
import java.net.URL;
import java.util.Map;
import javax.xml.namespace.QName;
import javax.xml.soap.SOAPElement;
import javax.xml.soap.SOAPException;
```

```
import javax.xml.soap.SOAPFactory;
import javax.xml.ws.BindingProvider;
import javax.xml.ws.Service;
import com.sun.xml.ws.api.message.Header;
import com.sun.xml.ws.api.message.Headers;
import com.sun.xml.ws.developer.WSBindingProvider;
```

You also have to import the automatically generated JAX-WS portable artifacts. It is common to call these classes stubs.

```
import stubs.InstanceInfoBean;
import stubs.InstanceInfoBeanList;
import stubs.OperationException_Exception;
import stubs.PapiWebService;
import stubs.PapiWebService_Service;
import stubs.StringListBean;
```

### Initialize the Web Service Client

To invoke PAPI Web Service you have to create a `Service` object. The constructor of a service object receives a `URL` object that contains the URL of the WSDL. In this example the URL of the WSDL is passed as an argument to the main method.

```
public class PapiWsJaxWsExample {
    private static final String SECURITY_NAMESPACE =

"http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd";

    public static void main(String[] args) {
        try {
            String endPoint = args[0];
            QName qName = new QName("http://bea.com/albpm/PapiWebService",
"PapiWebService");
            Service service = PapiWebService_Service.create(new URL(endPoint),
qName);
```

To be able to invoke operations over PAPI Web Service you have to obtain a `PapiWebService` object. `PapiWebService` exposes all the operations that you can invoke remotely over PAPI Web Service.

```
        PapiWebService papiWebServicePort =
service.getPort(PapiWebService.class);
```

If there is a problem accessing the WSDL endpoint the `URL` constructor throws a `MalformedURLException`, so you need to call it inside a try-catch block.

```
        //... Configure Authentication
        //... Operate with PAPI Web Service
    } catch (MalformedURLException e) {
        System.out.println("Could not connect to the web service endpoint");
        e.printStackTrace();
    }
}
```

## Configure Authentication

Before invoking any operation over the web service the client has to authenticate itself. This example shows you how to use JAX-WS with Username Token Profile and HTTP Basic authentication. Usually you choose one of these two mechanisms because PAPI Web Service only uses the second one in case the first authentication mechanism fails. For more information on how authentications mechanisms work in PAPI Web Service, see [PAPI Web Service Security Authentication](#) on page 26.

The code for this authentication mechanism is divided into two methods, one for each mechanism. These methods should be invoked before invoking any operation over the web service.

```
addUsernameTokenProfile(papiWebServicePort);
addHttpBasicAuthentication(papiWebServicePort);
```

The method `addUsernameTokenProfile()` configures Username Token Profile authentication. Some of the operations performed in this method throw `SOAPException`, so you need to call them inside a try-catch block.

This example adds the header programmatically but you can also configure Username Token Profile using Web Services Interoperability Technologies (WSIT) from Metro Web Services stack. For information about WSIT, see <http://wsit.dev.java>.

```
private static void addUsernameTokenProfile(PapiWebService papiWebServicePort)
    throws SOAPException {
    try {
        SOAPFactory soapFactory = SOAPFactory.newInstance();
        QName securityQName = new QName(SEcurity_NAMESPACE, "Security");
        SOAPElement security = soapFactory.createElement(securityQName);
        QName tokenQName = new QName(SEcurity_NAMESPACE, "UsernameToken");
        SOAPElement token = soapFactory.createElement(tokenQName);
        QName userQName = new QName(SEcurity_NAMESPACE, "Username");
        SOAPElement username = soapFactory.createElement(userQName);
        username.addTextNode("test");
        QName passwordQName = new QName(SEcurity_NAMESPACE, "Password");
        SOAPElement password = soapFactory.createElement(passwordQName);
        password.addTextNode("test");
        token.addChildElement(username);
        token.addChildElement(password);
        security.addChildElement(token);
        Header header = Headers.create(security);
        ((WSBindingProvider) papiWebServicePort).setOutboundHeaders(header);
    } catch (SOAPException e) {
        System.out.println("Could not configure Username Token Profile
authentication");
        e.printStackTrace();
    }
}
```

The method `addHttpBasicAuthentication()` configures HTTP Basic authentication by obtaining the request context and adding it the username and password properties.

```
private static void addHttpBasicAuthentication(PapiWebService
papiWebServicePort) {
    Map<String, Object> request =
        ((BindingProvider) papiWebServicePort).getRequestContext();
    request.put(BindingProvider.USERNAME_PROPERTY, "test");
    request.put(BindingProvider.PASSWORD_PROPERTY, "test");
}
```



```
}
```

## Operate with PAPI Web Service

Once PAPI Web Service successfully authenticates the client, the client is ready to perform operations over the web service.

The following code retrieves a list of available processes by invoking the method `processesGetIds()` over a `PapiWebService` object. It then iterates over them using those ids to obtain the instances for each process by invoking the method `processGetInstances()` over the `PapiWebService` object. If there is a problem performing any of the requested operations, this method throws an `OperationException`, so you need to invoke it inside a try-catch block. Finally it iterates over those instances invoking the method `getId()` and prints its result.

```
        try {
            StringListBean processIds =
papiWebServicePort.processesGetIds(true);
            for (String processId : processIds.getStrings()) {
                System.out.println("\nProcess: " + processId);
                InstanceInfoBeanList instances =
                    papiWebServicePort.processGetInstances(processId);
                for (InstanceInfoBean instance : instances.getInstances())
                {
                    System.out.println("-> " + instance.getId());
                }
            }
        } catch (OperationException_Exception e) {
            System.out.println("Could not perform the requested
operation");
            e.printStackTrace();
        }
```

## Complete Code Example

The following class contains all the steps described in this section.

```
package example;

import java.net.MalformedURLException;
import java.net.URL;
import java.util.Map;
import javax.xml.namespace.QName;
import javax.xml.soap.SOAPElement;
import javax.xml.soap.SOAPException;
import javax.xml.soap.SOAPFactory;
import javax.xml.ws.BindingProvider;
import javax.xml.ws.Service;
import com.sun.xml.ws.api.message.Header;
import com.sun.xml.ws.api.message.Headers;
import com.sun.xml.ws.developer.WSBindingProvider;
import stubs.InstanceInfoBean;
import stubs.InstanceInfoBeanList;
import stubs.OperationException_Exception;
import stubs.PapiWebService;
import stubs.PapiWebService_Service;
import stubs.StringListBean;

public class PapiWsJaxWsExample {
```

```

    private static final String SECURITY_NAMESPACE =
"http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd";

    public static void main(String[] args) {

        try {
            ////////////////////////////////////////////////// Initialize the web service client
            //////////////////////////////////
            String endPoint = args[0];
            QName qName = new QName("http://bea.com/albpm/PapiWebService",
"PapiWebService");
            Service service = PapiWebService_Service.create(new URL(endPoint),
qName);
            PapiWebService papiWebServicePort =
service.getPort(PapiWebService.class);

            ////////////////////////////////////////////////// Configure authentication //////////////////////////////////
            addUsernameTokenProfile(papiWebServicePort);
            addHttpBasicAuthentication(papiWebServicePort);

            ////////////////////////////////////////////////// Operate with PAPI Web Service //////////////////////////////////

            try {
                StringListBean processIds =
papiWebServicePort.processesGetIds(true);
                for (String processId : processIds.getStrings()) {
                    System.out.println("\nProcess: " + processId);
                    InstanceInfoBeanList instances =
papiWebServicePort.processGetInstances(processId);
                    for (InstanceInfoBean instance : instances.getInstance())
                    {
                        System.out.println("-> " + instance.getId());
                    }
                }
            } catch (OperationException_Exception e) {
                System.out.println("Could not perform the requested operation");

                e.printStackTrace();
            }
            } catch (MalformedURLException e) {
                System.out.println("Could not connect to the web service endpoint");

                e.printStackTrace();
            }
        }

        private static void addHttpBasicAuthentication(PapiWebService
papiWebServicePort) {
            Map<String, Object> request =
                ((BindingProvider) papiWebServicePort).getRequestContext();
            request.put(BindingProvider.USERNAME_PROPERTY, "test");
            request.put(BindingProvider.PASSWORD_PROPERTY, "test");
        }

        private static void addUsernameTokenProfile(PapiWebService papiWebServicePort)
        {
            try {
                SOAPFactory soapFactory = SOAPFactory.newInstance();
                QName securityQName = new QName(SECURITY_NAMESPACE, "Security");
                SOAPElement security = soapFactory.createElement(securityQName);
                QName tokenQName = new QName(SECURITY_NAMESPACE, "UsernameToken");
                SOAPElement token = soapFactory.createElement(tokenQName);
            }
        }
    }

```

```

        QName userQName = new QName(SEcurity_NAMESPACE, "Username");
        SOAPElement username = soapFactory.createElement(userQName);
        username.addTextNode("test");
        QName passwordQName = new QName(SEcurity_NAMESPACE, "Password");
        SOAPElement password = soapFactory.createElement(passwordQName);
        password.addTextNode("test");
        token.addChildElement(username);
        token.addChildElement(password);
        security.addChildElement(token);
        Header header = Headers.create(security);
        ((WSBindingProvider) papiWebServicePort).setOutboundHeaders(header);

    } catch (SOAPException e) {
        System.out.println("Could not configure Username Token Profile
authentication");
        e.printStackTrace();
    }
}
}

```

## Running Java JAX WS Client Example

The example contains all the necessary libraries to generate the stubs, compile and run it. These libraries are contained in the directory lib. It also contains an ant script that contains all the necessary configurations to compile, and run the client.

To run the example you need to install Ant. You can download it from <http://ant.apache.org/bindownload.cgi>

To run this example:

1. Enable PAPI Web Service for an already existing project.

See [Enabling PAPI Web Service](#) on page 21.

2. Locate example file included with your Oracle BPM installation at  
<ORABPM\_HOME>/samples/integration/BPM-PapiWs-JaxWs-Example.zip.
3. Unzip the file. This will generate a directory named ALBPM-PapiWs-JaxWs-Example.
4. Open a command-line session.
5. Change to the generated directory.  
For example: `cd ALBPM-PapiWs-JaxWs-Example.`
6. Type `ant run`.

Executing this task generates the stubs the client code needs to run, using the ant task JAX WS provides for this purpose, and compiles the classes used in the example before running them.

7. Enter PAPI Web Service's endpoint.

You can copy this from the link displayed next to WSDL in the Web Services Console. For information on how to launch the Web Services Console see [PAPI Web Service Configuration Console](#) on page 23.

After executing this procedure the example program runs. You will see a list of the processes deployed in the running process engine and below them a list of all the in-flight instances in each process.

## PAPI Web Service .NET Client Example

This section shows an example of a client developed with .NET Framework. This example uses PAPI Web Service to retrieve a list of process instances visible to the connected user.

## Download

You can download the complete code of this example from [http://edocs.bea.com/albsi/docs60/resources/papi\\_ws/ALBPM-PapiWs-DotNet-Example.zip](http://edocs.bea.com/albsi/docs60/resources/papi_ws/ALBPM-PapiWs-DotNet-Example.zip) . This project was developed using .NET Framework 2.0.5 and Microsoft Web Service Enhancements 3.0.

The syntax used for this example is C#.

## .NET Client Main-Class

This section analyzes the main-class of the .NET client example step by step.

### Programming a .NET Client

The typical steps you have to follow to use PAPI Web Service with JAX-WS stack are:

- Import the required libraries
- Initialize the web service
- Configure authentication
- Operate with PAPI Web Service

### Import the Required Libraries

This example uses classes from `System.Web.Services.Protocols` and `System.Net` packages. You have to import these classes to be able to use them in your code. The following code imports the classes from these packages that are used in this example.

```
using System.Web.Services.Protocols;
using System.Net;
```

The following code imports the classes that the client needs to perform authentication.

```
using Microsoft.Web.Services3.Security.Tokens;
using Microsoft.Web.Services3.Design;
```

You also have to import the automatically generated stubs. In the example project these classes were generated in the package `PAPI_WS2_Sample.papiws`.

```
using PAPI_WS2_Sample.papiws;
```

### Initialize the Web Service

The following code instantiates a web service proxy. This proxy will provide access to the operations exposed by PAPI Web Service.

```
papiws.PapiWebServiceWse proxy = new papiws.PapiWebServiceWse();
```

### Configure Authentication

Before invoking any operation over the web service the client has to authenticate. This example uses plain text Username Token Profile authentication. The authentication mechanism has to match the one you define in WSE 3 policy settings.

The following code creates a username token and sets it as the client credentials. Then it sets the client security policy by passing the id of this policy as an argument, to the method `SetPolicy()`.

```
UsernameToken token = new UsernameToken("test", "test",
    PasswordOption.SendPlainText);
proxy.SetClientCredential<UsernameToken>(token);
proxy.SetPolicy("ALBPM_Policy");
```

### Operate with PAPI Web Service

The following code retrieves a list of available processes by invoking the method `processesGetIds` over a `PapiWebServiceWse` object. It then iterates over them using those ids to obtain the instances for each process by invoking the method `processGetInstances()` over the `PapiWebServiceWse` object. If there is a problem performing any of the requested operations, this method throws a `SoapException`, so you need to invoke it inside a try-catch block. It is a good practice to wrap this exception in a user-defined exception. Finally, it iterates over those instances invoking the method `getId()` and prints its result.

```
try {
    //...
    foreach (string processId in processIds)
    {
        Console.Out.WriteLine("\n Process: " + processId);
        instanceInfoBean[] instances = proxy.processGetInstances(processId);
        foreach (instanceInfoBean instance in instances)
        {
            Console.Out.WriteLine(" -> " + instance.id);
        }
    }
}
catch (SoapException e)
{
    OperationException oe = new OperationException(e.Message);
    throw oe;
}
```

### Complete Code Example

The following class is the main class of the .NET PAPI Web Service client example.

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Web.Services.Protocols;
using System.Net;
using PAPI_WS2_Sample.papiws;

using Microsoft.Web.Services3.Security.Tokens;
using Microsoft.Web.Services3.Design;

namespace PAPI_WS2_Sample
{
    class Program
    {
        static void Main(string[] args)
        {
            //Set a custom handler for unhandled exceptions (optional)
            AppDomain.CurrentDomain.UnhandledException +=
```

```

Program.UnhandledExceptionHandler;

        try
        {
            ////////////////////////////////////////////////// Initialize the web service client
            ///////////////////////////////////
            papiws.PapiWebServiceWse proxy = new papiws.PapiWebServiceWse();

            ////////////////////////////////////////////////// Configure authentication ///////////////////////////////////

            UsernameToken token = new UsernameToken("test", "test",
PasswordOption.SendPlainText);
            proxy.SetClientCredential<UsernameToken>(token);
            proxy.SetPolicy("ALBPM_Policy");

            //set timeout and encoding
            proxy.Timeout = 60000;
            proxy.RequestEncoding = Encoding.UTF8;

            ////////////////////////////////////////////////// Operate with PAPI Web Service
            ///////////////////////////////////
            string[] processIds = proxy.processesGetIds(false);

            foreach (string processId in processIds)
            {
                Console.Out.WriteLine("\n Process: " + processId);
                instanceInfoBean[] instances =
proxy.processGetInstances(processId);
                foreach (instanceInfoBean instance in instances)
                {
                    Console.Out.WriteLine(" -> " + instance.id);
                }
            }

            catch (SoapException e)
            {
                OperationException oe = new OperationException(e.Message);
                throw oe;
            }

        }

        static public void UnhandledExceptionHandler(object sender,
UnhandledExceptionEventArgs e)
        {
            Console.Error.WriteLine("Unhandled Exception: \n" +
e.ExceptionObject.ToString());
            Environment.Exit(-1);
        }
    }

    public class OperationException : Exception
    {
        public OperationException(String message) : base(message)
        {
        }
    }
}

```