



A UML state machine diagram is faintly visible in the background. It features several states represented by circles, some of which are double circles indicating initial or final states. Transitions are shown as arrows between these states, including a horizontal transition from a state on the left to a final state on the right, and several curved transitions connecting other states in a complex flow.

# FUEGO

## PAPI-WEBSERVICE

Fernando Dobladez  
*ferd@fuego.com*

August 24, 2004

### Abstract

This document describes *PAPI-WS*: a simplified version of **PAPI** (Fuego's Process-API) exposed as a web service.

PAPI-WS is provided in Fuego 5, and consists of a SOAP/HTTP web service API that allows for external systems to interact with Fuego business processes.

The reader should be familiar with how Fuego processes are designed. Knowledge of how to consume a web service (from any programming language or tool) is needed in order to use PAPI-WS.

## Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Functionality</b>	<b>3</b>
<b>3</b>	<b>Deployment</b>	<b>4</b>
<b>4</b>	<b>How to Use PAPI-WS</b>	<b>5</b>
4.1	First Example . . . . .	6
4.2	Types and Operations . . . . .	7
4.2.1	<code>createSession</code> and <code>createNESession</code> . . . . .	9
4.2.2	<code>close</code> . . . . .	9
4.2.3	<code>createInstance</code> . . . . .	9
4.2.4	<code>getInstancesByView</code> . . . . .	11
4.2.5	<code>runActivity</code> . . . . .	11
4.2.6	<code>runGlobalActivity</code> . . . . .	12
4.2.7	<code>getVariables</code> . . . . .	14
4.2.8	<code>sendNotification</code> . . . . .	14

## 1 Introduction

The Java-based **PAPI** is a powerful programming interface that provides full interaction with processes running with a Fuego Engine.

For instance, Fuego Portal interaction with the Engine is implemented on top of **PAPI** exclusively. This means all the functionality that the Portal provides to interact with Fuego processes can be achieved programmatically via **PAPI**.

As powerful as it is, **PAPI** has some considerations:

- It is a Java-based API. Utilizing from non-Java applications requires the creation of custom *wrappers* on top of **PAPI**, or some kind of *bridge* to marry it with non-Java technology. This means more development and deployment complexity.
- The use of PAPI requires some initialization and infrastructure support code that adds complexity even to the achieve simple tasks (i.e. creating a new process instance).

For these reasons, Fuego includes the **PAPI-WS** web service. PAPI-WS exposes some of the functionality that **PAPI** provides. It is simpler and language-independent.

## 2 Functionality

Following is a summary of the operations that the current **PAPI-WS** provides:

- `createInstance` – Create a new process instance. String arguments can be sent to the `begin` activity. *create-Instance added in version 5.0-SP1*
- `getVariables` – Get the list of variables of a particular instance and their current values.
- `sendNotification` – Notify a particular instance. The notification message can be sent to a regular *wait* activity or to one that allows for interruptions.

- `runGlobalActivity` – Execute a *global* or *global creation* activity. String arguments can be sent to the activity. Also, any output arguments are returned from the activity. In the case of activities of type *global creation*, information about the newly created instance is also returned.
- `getInstancesByView` – Get the list of instances that are visible through a particular *view* (as defined in the Fuego Portal and Portal Administrator).
- `runActivity` – Execute an *interactive* (non-global) activity for an specific instance. Only activities without client-side code can be run.

As designed, **PAPI-WS** is a small API which only exposes the most common operations available from **PAPI**.

### 3 Deployment

The actual **PAPI-WS** web service is bundled within the Fuego Portal web application.

In order to make the **PAPI-WS** available, the Fuego Portal must be deployed and running.

In Fuego Studio, this simply translates to opening a project and starting the Engine from the menu (*Run*→*Start Engine...*). Similarly, when a project is started in Fuego Express, the embedded Portal and **PAPI-WS** are also started and ready to use.

Once the Portal (and therefore, the **PAPI-WS** service) are running, the WSDL<sup>1</sup> description of the web service will reside on:

```
http://hostname:port/Portal/webservices/ProcessService?WSDL
```

Where:

**hostname** is the server hostname where the Fuego Portal is running.

---

<sup>1</sup>Web Service Description Language

**port** is the TCP port the Portal is listening on. Fuego Studio defaults to 9595.

**Portal** is the path to the Fuego Portal application. In the case of Studio and Express, this will be the name of the desired Project.

For example, after opening a project named *IssueTracking* in Fuego Studio and starting the Engine on the default port 9595, the location of the WSDL descriptor would be:

```
http://localhost:9595/IssueTracking/webservices/ProcessService?WSDL
```

## 4 How to Use PAPI-WS

This document will not go into the details on how to consume a web service. There are different tools and different languages that can be used to call a web service.

It is highly recommended to use a tool that will inspect the WSDL descriptor and generate local stubs/skeletons that can be used to invoke the web service as if it were a local API.

This will make the client code easier to write and maintain, since most of the technical problems are solved by such a tool, and having locally generated skeletons allows for doing type-checking on the part of the compiler.

**Note:** *Fuego itself is a good example of such tool. When a web service is cataloged in Fuego Studio, an object is created for the service and for every type described in the WSDL descriptor. Then, using the web service from a Fuego process feels no different than using a set of local Fuego Objects. All the communication, the marshaling of objects to/from XML, and all the dirty work is automatically solved under the hood.*

Therefore, the explanations on this document will assume the use of this type of tool and will concentrate on the semantics of the *PAPI-WS* operations. They will ignore the technicalities around web services. For code examples, Java will be used, but the code can be easily extrapolated to any other language or tool.

### 4.1 First Example

Following is a simple, but complete, example of a Java application that uses PAPI-WS to list the instances visible through a particular view.

The example uses auto-generated classes that provide access to the web service.<sup>2</sup>

*PAPIWSExample.java*

```
package fuego.papi.ws.example;

import fuego.papi.ws.client.*;

public class PAPIWSExample
{
    public static void main(String[] args) throws Exception
    {
        // Get a handle to the ProcessService
        ProcessServiceServiceLocator locator = new
            ProcessServiceServiceLocator();
        ProcessService service = locator.getProcessService();

        String mySession = null;

        try {
            String userid    = args[0];
            String password = args[1];
            String view      = args[2];

            // Establish a PAPI session
            mySession = service.createSession(userid, password);

            // Get the list of instances on the specified view
            // and display them
            InstanceInfo[] instances = service.getInstanceByView(
                mySession, view);

            for(int i=0; i < instances.length; i++)
                System.out.println "["+i+"] "+instances[i].getId());

        }
        finally {

            // make sure the session is discarded
            if(service != null && mySession != null)
```

<sup>2</sup>Apache Axis 1.1 was the tool used to generate client classes from the WSDL descriptor.  
See: <http://ws.apache.org/axis>

```
        service.close(mySession);  
    }  
}  
}
```

The example shows the basic steps that any application needs to follow in order to use the **PAPI-WS** service:

1. **Establish a session with the Fuego Engine.** For this, a valid user-id and password is required
2. **Perform any operation provided by the service.** All operations require a valid open session to run. Any number of operations can be run using the same open session.
3. **Close the session.** It is very important to close the session after its last use in order to avoid *leaking* resources.<sup>3</sup>

The above example is a command-line application. It expects the user-id, password and name of the *view* as arguments. This shows a sample run:

```
java fuego.papi.ws.example.PAPIWSExample idamay idamay  
unified_inbox  
  
[0] /HelloWorld#Default-1.0/1/0  
[1] /HelloWorld#Default-1.0/2/0
```

The sample execution displayed the *id* of two instances pertaining to the HelloWorld process.

Note that in Fuego Studio the password for any user is always the user-id itself. Also, the example uses the *unified\_inbox* view, which is the default *inbox* that is created when a project is deployed.

## 4.2 Types and Operations

The **PAPI-WS** WSDL defines the following complex types, which are used as *value objects*, that is, to send and receive information to/from the service:

**ProcessService** This is the service itself. It contains the set of available operations (methods).

---

<sup>3</sup>The example takes advantage of the `finally` block that Java provides, which will guarantee that the session is closed even in case of exceptions.

**KeyValuePair** This type is a simple structure that contains a key and a value. It is used when named arguments need to be sent or received.

**InstanceInfo** A type that contains information about a process instance.

**GlobalExecutionResult** The execution of a *global* activity returns an object of this type. It includes the `InstanceInfo` of the created instance (in the case of a *global creation* activity) and a set of arguments as an array of `KeyValuePair`.

Here is the prototype for each of the operations provided by Process-Service:

```
String ← createSession(String participant, String password)

String ← createNESession(String participant, String password)

void ← close(String sessionId)

InstanceInfo ← createInstance(String sessionId,
                               String processId, String argumentSetName,
                               KeyValuePair[] arguments)

InstanceInfo[] ← getInstancesByView(String sessionId,
                                     String viewId)

InstanceInfo ← runActivity(String sessionId, String instanceId,
                           String activityName)

GlobalExecutionResult ← runGlobalActivity(String sessionId,
                                           String activityId,
                                           KeyValuePair[] arguments)

KeyValuePair[] ← getVariables(String sessionId,
                               String instanceId)

void ← sendNotification(String sessionId, String instanceId,
                        String activityName,
                        String argumentSetName,
                        KeyValuePair[] arguments)
```

Following is a description for each operation and how to use them.



#### 4.2.1 createSession and createNESession

As explained in the above example, the first step before executing any useful operation via **PAPI-WS** is to establish a session.

There two types of sessions in **PAPI**:

**Exclusive:** Which means only one session is allowed for each user at any given time.

**Non-exclusive:** A single user-id can be used to create many simultaneous sessions

The `createSession` call provides an *exclusive* session, and the `createNESession` returns a *non-exclusive*.

Some operations *require* the use of an exclusive session, while others can be executed in the context of a non-exclusive session.

In both cases, the operation expects the participant's id and password as arguments, and returns a character `String` which will uniquely identify the session just created.

**Note:** In Fuego Studio the password for any user is always the user-id itself. Example: if the user id is *joe*, the password will also be *joe*.

Once a session is created, it can be used to call other operations as many times as needed, until it is discarded with the `close` operation.

#### 4.2.2 close

Simply closes the given session, freeing any server-side resources attached to it.

Once the session is closed, the session `String` cannot be used again.

#### 4.2.3 createInstance

This operation creates a new instance into the given process.

Note that this operation *requires* an exclusive session.

The `processId` argument is a string identifying the process in which the new instance will be created. This string is composed as follows:

```
/OrgUnit/ProcessName#Variation-Major.Minor
```

Spaces within the process name should be omitted. The version of the process (starting with the '#' sign) is optional: if omitted, the last deployed version is assumed.

If no organizational unit was specified when deploying the process, then the `processId` will not contain one.

Examples of valid `processId`:

```
/IT/HelloWorld#Default-1.0  
  
/HelloWorld#Default-1.0  
  
/IT/HelloWorld  
  
/HelloWorld
```

The `argumentSetName` argument is a string identifying the argument set.<sup>4</sup>

The list of arguments that can be sent to the process is passed with the `arguments` parameter, as an array of `KeyValuePair`.

Example:

```
...  
// Prepare to send one argument to the instance  
KeyValuePair[] values = new KeyValuePair[1];  
values[0] = new KeyValuePair();  
values[0].setKey("argumentOne");  
values[0].setValue("valueOne");  
  
// Create instance  
String myProcessId = "/HelloWorld";  
String myArgset = "BeginIn";  
  
InstanceInfo info = service.createInstance(mySession,  
                                           myProcessId, myArgset, values);
```

<sup>4</sup>The argument-sets for the process are defined as a property of the `Begin`. In Fuego Studio, right-click on the `Begin` activity, select *argument mapping* and the argument-sets will appear on the left-hand side of the mapping dialog.

```
// display the id of the created instance
System.out.println("New instanceId = "+ info.getId());
...
```

#### 4.2.4 `getInstancesByView`

Returns an array of `InstanceInfo` objects with some information about all the instances sitting on the specified view.

This operation does not require an exclusive session. Either an exclusive or a non-exclusive session can be used with this operation.

The *views* in the Fuego Portal are automatically created when a project is deployed, but can also be created and customized by end users and by administrators using the Portal Admin tool.

It is important to remember that the available views and the list of process instances in each view depends on the user that created the session.

For example, the default *inbox* view (its id is `unified_inbox`) will list all the instances that are pending and can be executed *by the current user* (that is, by the user that established the session). Instances sitting in activities that the current user cannot access will not show up in the `unified_inbox` view of this user.

See [First Example](#) on page 6 for an example of using `getInstancesByView`.

#### 4.2.5 `runActivity`

This operation allows to execute a particular instance that is sitting on an *interactive* activity.

In other words, this operation allows to programmatically *release* an instance from an interactive activity. The activity is effectively executed in the same manner as if the end-user does it from the Fuego Portal.

Note that this operation *requires* an exclusive session.



**Important:** This operation cannot be used if the interactive activity expects some client-side input (like presenting a form to the user). If this is the case, the call to `runActivity` will freeze until the Portal times-out.

**PAPI-WS** does not currently provide a mechanism to simulate generic end-user interaction.

The `instanceId` argument is a string that identifies a particular instance. It can be obtained from an `InstanceInfo` object for example, returned by `getInstancesByView`.

The `activityName` argument is a string identifying the activity where the instance stays.

Example:

```
...
String myInstance = "/HelloWorld#Default-1.0/1/0";
String activity = "Check Idea";
service.runActivity(mySession, myInstance, activity);
...
```

#### 4.2.6 runGlobalActivity

Activities of type *global interactive* or *global creation* can be run using this operation.

In the case of a *global creation*, an implicit process instance is created as a side-effect.

This operation does not require an exclusive session.

The `activityId` argument is a string identifying the activity that will be executed. This string is composed as follows:

```
ProcessId/ActivityName
```

Refer to [createInstance](#) on page 9 for the syntax of `ProcessId`. Spaces within the activity name should be omitted.

Examples of valid `activityId`:

```
/IT/HelloWorld#Default-1.0/CreateIdea
```

```
/IT/HelloWorld/CreateIdea  
  
/HelloWorld/CreateIdea
```

Optionally, arguments can be sent to the activity using the `arguments` parameter, which is an array of `KeyValuePair`.

The return value of `runGlobalActivity` is a `GlobalExecutionResult` object. This object includes an `InstanceInfo` that describes the new created instance (if any)<sup>5</sup>, and a set of arguments returned by the activity (those marked of type `out` or `in/out` in the activity).

Example:

```
...  
// Prepare to send two arguments  
KeyValuePair[] values = new KeyValuePair[2];  
values[0] = new KeyValuePair();  
values[0].setKey("argumentOne");  
values[0].setValue("valueOne");  
  
values[1] = new KeyValuePair();  
values[1].setKey("argumentTwo");  
values[1].setValue("valueTwo");  
  
GlobalExecutionResult result =  
    service.runGlobalActivity(mySession, "/HelloWorld/  
        CreateIdea", values);  
  
// display the id of the created instance (if any)  
System.out.println("New instanceId = "+result.getInstanceInfo().  
    getId());  
  
// display the received arguments  
KeyValuePair[] outArgs = result.getArguments();  
System.out.println("Received arguments:");  
  
for (int i = 0; i < outArgs.length; i++)  
    System.out.println(outArgs[i].getKey() + " = "+  
        outArgs[i].getValue());  
  
...
```

---

<sup>5</sup>Only meaningful if the activity is of type *global creation*

#### 4.2.7 `getVariables`

Given a particular `instanceId`, this operation provides the list of the instance's variables and their values. All the values are converted into a `String` before being returned.

This operation does not require an exclusive session.

The `instanceId` argument is a string that identifies a particular instance. It can be obtained from an `InstanceInfo` object, for example, returned by `getInstancesByView`.

The variable names and values are returned as an array of `KeyValuePair`.

```
...
String myInstance = "/HelloWorld#Default-1.0/1/0";

KeyValuePair[] vars = service.getVariables(mySession, myInstance
);

// Display the variables and their values
System.out.println("Variables for instanceId "+myInstance+":");

for (int i = 0; i < outArgs.length; i++)
    System.out.println(outArgs[i].getKey() + " = " +
        outArgs[i].getValue());
...
```

#### 4.2.8 `sendNotification`

This operation allows to send a process notification to activities of type *wait*, including those that allow for interruptions.

This operation does not require an exclusive session.

The `instanceId` argument is a string that identifies a particular instance. It can be obtained from an `InstanceInfo` object, for example, returned by `getInstancesByView`.

The `activityName` argument is a string identifying the *wait* activity where the notification will be sent.

The `argumentSetName` argument is a string identifying the argument set.<sup>6</sup>

---

<sup>6</sup>The argument-sets are defined as a property of the activity. In Fuego Studio, right-

The list of arguments that can be sent with the notification message is passed with the `arguments` parameter, as an array of `KeyValuePair`.

Example:

```
...
// Prepare to send one argument with the notification
KeyValuePair[] values = new KeyValuePair[1];
values[0] = new KeyValuePair();
values[0].setKey("argumentOne");
values[0].setValue("valueOne");

// Send the notification
String myInstance = "/HelloWorld#Default-1.0/1/0";
String myActivityName = "NotificationWait";
String myArgset = "NotificationWaitIn";

service.sendNotification(mySession, myInstance, myActivityName,
    myArgset, values);
...
```

---

click on a *wait* activity, select *argument mapping* and the argument-sets will appear on the left-hand side of the mapping dialog.