Connecting Custom Services to the YAWL Engine

Beta – 7 Release

QUT Faculty of Information Technology

Document Control

Date	Author	Version	Change
25 Feb 2005	Marlon Dumas, Tore Fjellheim, Lachlan Aldred	0.1	Initial Draft
3 March 2006	Lachlan Aldred	0.1.1	General Revisions

Preface

This document describes the key concepts behind designing and building custom Web services for YAWL.

Contents

Introduction

An important point of extensibility of the YAWL system is its support for interconnecting external applications with the workflow execution engine using a service-oriented approach. This enables running workflow instances and external applications to interact with each other in order to delegate work or to signal the creation of work items or a change of status of existing work items. This document briefly describes the mechanisms for achieving this interconnection.

External applications, exposed as services, can interact with the YAWL engine in two different ways:

1. Directly, by means of XML messages exchanged over HTTP. Services connected in this way are called *custom YAWL services*.
2. Indirectly, through the YAWL Web Services Invoker (YAWL WSInvoker); the YAWL engine communicates with the YAWL WSInvoker which then invokes an operation of an external service using SOAP (for example).

The former mechanism is more general than the latter. However, the latter mechanism may be easier to use in the case where the execution of a task instance requires a service operation to be invoked in a synchronous way and the service interface has been described in WSDL. The YAWL Engine Beta 3 comes with two sample external services: A “Time Service” which illustrates the former mechanism, and the “Barnes & Noble” service bridge, corresponding to the latter mechanism.

The following sections briefly explain each of these mechanisms in turn.

Custom YAWL Services

Custom YAWL services interact with the YAWL engine through XML/HTTP messages processed by certain endpoints, some located on the YAWL engine side and others on the service side. In principle, custom YAWL services can be developed in any programming language and can be deployed in any platform capable of sending and receiving HTTP messages. Custom YAWL services are registered with the YAWL engine by specifying their location, in the form of a “base URL”. Once registered, a custom service may receive XML messages from the engine at endpoints identified by URLs derived from the base URL provided at registration. On the other hand, the custom service can send XML messages to the YAWL engine at endpoints identified by URLs that the custom service is assumed to know. At present, the URL encoding used, the

types of HTTP operations used, and the schema and semantics of the exchanged messages are not yet documented. However, a collection of Java classes included in the YAWL engine distribution can be used to implement the required endpoints without requiring any knowledge about the URL encoding and XML formatting used. Specifically, the following classes can be reused to develop custom YAWL services:

- `au.edu.qut.yawl.engine.interfce.InterfaceB_EnvironmentBasedServer` (Note: this class is a Java servlet)
- `au.edu.qut.yawl.engine.interfce.InterfaceB_EnvironmentBasedClient`
- `au.edu.qut.yawl.engine.interfce.InterfaceBWebsideController`

A custom YAWL service can be developed by extracting these classes (and their dependent classes) from the YAWL engine distribution and following the steps outlined below. Before applying these steps, it is important first to understand the purpose of the interactions in which the YAWL engine and a YAWL custom service can engage. Four of these interactions are particularly relevant here:

1. The interaction that the engine initiates to indicate that a new work item has been created (i.e. it is enabled). This message contains the identifier of the work item and related data.[1]
2. The interaction that the service initiates to check-out the work item (i.e. to move the work item from the "enabled" to the "executing" states).
3. The interaction that the engine may initiate to indicate that a work item in the "in progress" state should be cancelled.[2]
4. The interaction that the custom service initiates to check a work item back into the engine thus indicating that the work item has been completed (i.e. to moves work item from the "in progress" to the "completed" state).

Step 1 – Initial Setup:

Create a class which extends the InterfaceBWebSideController abstract class. You will need to implement three methods:

- *public void* ***handleEnabledWorkItemEvent****(WorkItemRecord workItemRecord)*

[1] In this document, we only cover "push-style" YAWL services, whereby the interactions to announce a new work item and to cancel a work item are initiated by the engine. YAWL Beta 3 also supports a "pull-style" interaction, whereby the responsibility of finding out which work items are available is left to the service (i.e. the service must initiate all interactions). The YAWL worklist is an example of a "pull-style" YAWL service (but we do not document it here).

[2] In the current "work item lifecycle" model of YAWL does not contemplate the possibility of resuming previously cancelled work items.

This method is invoked when a new work item is created and thus placed in the enabled state (i.e. when the interaction #1 above occurs).

- *public void* ***handleCancelledWorkItemEvent****(WorkItemRecord workItemRecord)*
  This method is invoked when an enabled or "in progress" work item is cancelled by the engine (e.g. due to a cancellation occurring in the YAWL workflow). In principle, after this method has been invoked, the custom service should not check-in/check-out the item, and if the work item is in progress, processing should be stopped.[3]
- *public void* ***setRemoteAuthenticationDetails****(String userName, String password, String httpProxyHost, String proxyPort)*
  This method is used for connecting to the engine over a firewall, through a proxy server.

Step 2 – Implementing method handleEnabledWorkItemEvent:

It is your job as developer of the custom YAWL service to ensure that a work item is checked out of the engine, and once the task is complete to return the result back into the engine. The way you achieve this is up to you. One useful technique is to implement the *handleEnabledWorkItemEvent* method so that it checks out the enabled work item, performs the desired task, and then checks the work item back into the engine. The following explanation assumes that this is what you intend to do.

This method should first try to check if there is a connection from the custom service to the engine. This is done by calling the *checkConnection* method of the superclass.

```
checkConnection(sessionHandle)
```

If this method returns 'false', the method implementation should call the *connect* method, with a username and password acceptable to the engine. This returns a *sessionHandle* that allows this service to connect to the engine.

After the connection has been established, the method *checkOut* may be invoked to check out the enabled work item. This method requires the enabled workItem's ID which is retrieved by using its *getID* method. The method *checkOut* returns a diagnostic string. In order to determine if the checking out was successful one can pass the diagnostic string into the *successful* method. If it returns true then a new work item was created in the engine with state 'executing'. Hence processing can start.

[3] Note that the current version of YAWL does not support any notion of "atomic execution", so no "rollback" is required when a work item is cancelled.

Each enabled workItem decomposes into one or more child tasks. Therefore checking out the enabled work item will put one of the children into the state 'executing'. This work item is checked out of the engine. However the remaining children of the enabled work item will not be checked out yet. Hence they will exist in a state in between 'enabled' and 'executing'. This is the state 'fired'. The state of 'fired' means that the task is not yet checked out of the engine (i.e. 'executing') but it has moved beyond the state of enabled[4]. Hence it may be necessary to check out the 'fired' children in additionally. This can be done by calling the method *getChildren* which returns a list (class *java.util.List*) of *WorkItemRecord* objects. This list will contain the children of the enabled work item, and one of those children will be in the state 'executing'. The others will be only fired. It is recommended that you check out all children in the "fired" state.

```
List children =
  getChildren(enabledWorkItem.getID(), sessionHandle);
  for (int i = 0; i < children.size(); i++) {
         WorkItemRecord itemRecord =
               (WorkItemRecord) children.get(i);
         if(WorkItemRecord.statusFired.equals(
               itemRecord.getStatus())) {
                 checkOut(itemRecord.getID(),
_sessionHandle);
         }
 }
```

After checking out all child workItems, the custom service can execute the actual work. For this purpose, the custom service will probably need to retrieve the input Data (if any) from the work item. The code snippet shown below (extracted from the Time Service) illustrates how this can be done:

```
List executingChildren =
getChildren(enabledWorkItem.getID(), _sessionHandle);
for (int i = 0; i < executingChildren.size(); i++) {
      WorkItemRecord itemRecord =
      (WorkItemRecord) executingChildren.get(i);
      Element datalist = itemRecord.getDataList();
      String dataText =
      datalist.getChildren().get(0).getText();
}
```

Looking at this code, it can be seen that first, all the children of the work Item are obtained. Then for each child we retrieve the datalist. The datalist is an Element (encoded in JDOM) which can be parsed

[4] The YAWL paper (http://www.citi.qut.edu.au/pubs/technical/yawlrevtech.pdf) discusses the full semantics of the YAWL language. For more information about enabled tasks, fired tasks, and executing tasks refer to the sections about multiple instances of an atomic task.

to locate the correct data. In the example of the Time Service, there is only one input is a time duration. After the data has been retrieved, the execution can start. When the execution completes, the system should call the *checkIn* method on the child which has finished.

When the execution finishes. The system must call the *"checkIn"* method on the child which has finished. This method takes four parameters.

1) The identifier of the work item which was checked out
2) The input data of this work item
3) The output data of this work item
4) The session handle.

The input data of the work item is stored in the work item itself. The output data must be created by the service. These elements are described as XML Strings where the root element is the decompositionID and the inner elements describe the data. The decomposition ID is found in the task information. This can be retrieved by calling:

```
getTaskInformation(  specificationID,
                     taskID(),
                     _sessionHandle);
```

An example of checking in a work item is shown below. The example is from the Time Service.

```
TaskInformation taskinfo =
getTaskInformation(itemRecord.getSpecificationID(),
                                itemRecord.getTaskID(),
                                       _sessionHandle);

String start = "<" + taskinfo.getDecompositionID() +
">";
String end = "</" + taskinfo.getDecompositionID() +
">";

XMLOutputter outputter = new XMLOutputter();

checkInWorkItem(itemRecord.getID(),
outputter.outputString(itemRecord.getDataList()),
                        start+end,
                        _sessionHandle);
```

Step 3 (optional) – Implement handleCancelledWorkItemEvent()

Sometimes, a work item in the "executing" state (i.e. work items that have been checked out but not yet checked in) needs to be cancelled

because the corresponding task is in the cancellation set of another task, and this latter task moves to the “completed” state. In this situation, the method *handleCancelledWorkItemEvent* will be invoked so that the custom YAWL service can optionally stop executing the work item. The custom YAWL service could optionally do nothing:

```
public String
handleCancelledWorkItemEvent(WorkItemRecord
workItemRecord) {
  return "Cancelled";
}
```

Step 4 – Deploy the custom service in a Java servlet container

Once we have subclassed *InterfaceBWebSideController* and implemented its abstract methods we need to create a web application that is able to receive events from the engine when tasks become enabled and/or cancelled. The YAWL Servlet *InterfaceB_Server_WebSide* is able to act as the event handler for messages coming from the YAWL engine. However to pass these events successfully to your code it needs to be told exactly which class (which is a subclass of *InterfaceBWebSideController*) to load. We can describe this for Tomcat by using the *web.xml* format shown later.

In the case of Tomcat, the code for the custom YAWL service will need to be grouped in a “war” file and deployed in the *webapps* directory of the Tomcat installation. In addition, a deployment descriptor (web.xml file) needs to be included inside a directory labelled ‘WEB-INF’ (for details, see Tomcat’s documentation).

As an example, the *web.xml* file should be something like:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application
2.3//EN"  "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
    <!-- General description of your web application -->
    <display-name> ... </display-name>
    <description> ...  </description>

    <context-param>
        <param-name>InterfaceB_BackEnd</param-name>
        <param-value>
            http://localhost:8080/yawl/ib</param-value>
        <description>
            The URL of the engine's Interface B.
        </description>
    </context-param>
```

```
    <context-param>
        <param-name>InterfaceBWebSideController</param-name>
        <param-value>
            com.company.SubClassOfInterfaceBWebsideController
                </param-value>
        <description>
            The class name of the Interface B
            Server implementation.
        </description>
    </context-param>

    <servlet>
        <servlet-name>InterfaceB_Servlet</servlet-name>
        <description>
            Listens to notification of work items from the
            engine.  Shouldn't need to change this.
        </description>
        <servlet-class>
au.edu.qut.yawl.engine.interfce.InterfaceB_EnvironmentBasedSe
rver</servlet-class>
        <load-on-startup>2</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>InterfaceB_Servlet</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>

    <session-config>
        <session-timeout>30</session-timeout>
        <!-- 30 minutes -->
    </session-config>
</web-app>
```

In this web.xml file, it is important to note that:
1) The servlet class is set to *InterfaceB_EnvironmentBasedServer*. This is the actual servlet which will be started.
2) The *InterfaceBWebSideController* parameter value is the name of the class implementing the *HandleEnabledWorkItemEvent* method discussed in Step 2.
3) The name of the .war file that you wrap your code in will determine the URL that you use when you register your YAWL service with the engine, in the next step. For example if your war file is named 'myCustomSerive.war' then the URL registered in Step 5 could be `http://localhost:8080/myCustomService.`

Step 5 – Register the service with the engine

This can be done from the HTML interface of the YAWL engine, under the "Administration" menu. In this page there is a list of registered custom YAWL services, and a form for adding new ones. During registration, the URL of the servlet that will handle incoming events from the engine will need to be added exactly (see previous

section). In addition to this one may add some documentation regarding what the service actually does (see Figure 1).

Then click on Add YAWL Service:

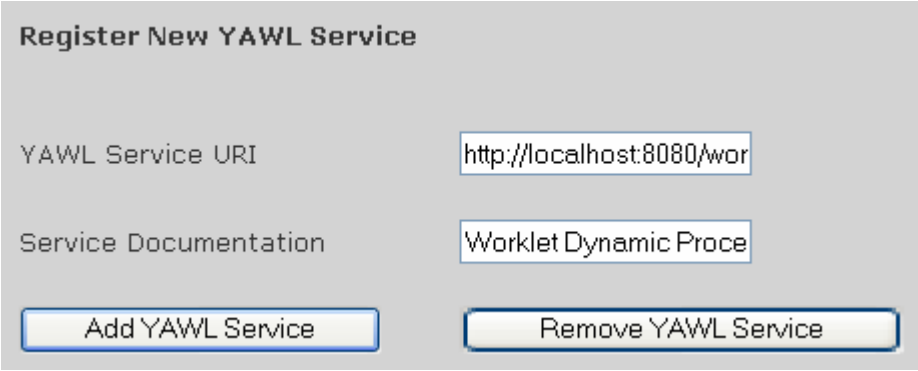

Figure 1: Admin page widget for adding custom YAWL services to engine.

Step 6 – Write and deploy a process definition that uses the service

The YAWL editor can be used to directly link with your custom YAWL services. See the editor manual for details about how to do this.

Example: The YAWL WSInvoker

The YAWL WSInvoker is applicable in the case where an external application (exposed as a service) needs to be invoked by the YAWL engine to perform the work corresponding to a task instance. In this mechanism, a task is decomposed to an invocation to a (web) service operation described in a WSDL interface referenced in the process description. When an instance of the task is enabled, the YAWL WSInvoker retrieves the corresponding WSDL interface and using the Apache Web Services Invocation Framework (WSIF),[5] it analyses it to determine how to invoke the service operation in question. The operation invocation is then performed immediately and accordingly, the task instance is moved from the enabled to the processing state. When the operation invocation completes, the task is moved to the completed state. Note that in the current version of

[5] http://ws.apache.org/wsif

the YAWL WSInvoker, the operation invocation must be synchronous (i.e. request/response), so this mechanism can not be used to connect services to YAWL in an asynchronous way (for this purpose, custom YAWL services must be introduced).

The following task decomposition, extracted from the *Barnes&Noble.xml* in the YAWL distribution, shows how an external web service can be invoked from through the YAWL WSInvoker. When a work item is created for this task, the operation *getPrice* of the Barnes&Noble (SOAP) web service is invoked, with the ISBN as parameter.

```
<decomposition id="Get Price via WS"
xsi:type="WebServiceGatewayFactsType">
  <inputParam name="isbn">
        <type>xs:string</type>
  </inputParam>
  <outputExpression query="/data/return"/>
  <outputParam name="return">
        <type>xs:float</type>
        <mandatory/>
  </outputParam>
  <yawlService id="http://localhost:8080/yawlWSInvoker/">
       <wsdlLocation>
         http://www.xmethods.net/sd/2001/BNQuoteService.wsdl
       </wsdlLocation>
         <operationName>getPrice</operationName>
  </yawlService>
</decomposition>
```

Note that, unlike custom YAWL services, services that are connected to the engine through the YAWLWSInvoker do not need to be registered with the YAWL engine through the Web-based administration interface.

We can note that from the perspective of the YAWL engine, the YAWLWSInvoker is just like any custom service, except that it is part of the distribution of the YAWL engine and it does not need to be registered as other custom services do. In particular, the code of the YAWL WSInvoker (see package au.edu.qut.yawl.wsif and especially class WebServiceController in this package) provides an example of how to implement a (custom) YAWL service (in addition to the “Time Service” example).