Usability Extensions for the Worklet Service

Michael Adams

Queensland University of Technology, Brisbane, Australia.
`mj.adams@qut.edu.au`

Abstract. The YAWL Worklet Service is an effective approach to facilitating dynamic flexibility and exception handling in workflow processes. Recent additions to the Service extend its capabilities through a programming interface that provides easier access to rules storage and evaluation, and an event server that notifies listening servers and applications when exceptions are detected, which together serve enhance the functionality and accessibility of the Service's features and expand its usability to new potential domains.

Keywords: YAWL, Worklet, Exception Handling, Flexibility

1 Introduction

Since its introduction, the YAWL Worklet Service has proven to be an effective approach to facilitating dynamic flexibility and exception handling in otherwise static process instances [2, 3]. The Service provides an extensible *repertoire* of self-contained selection and exception-handling processes, coupled with an extensible *ripple-down* rules set. At runtime, the Service provides functionality that allows for the substitution of a task with a sub-process, dynamically selected from its repertoire, depending on the rules set and the context of the particular work instance. In addition, exception-handling functionality dynamically detects and mitigates exceptions as they occur, using the same repertoire and rule set framework[1].

Now, recent additions to the service provide new methods of interaction that extend its capabilities through the introduction of an *application programming interface* (API) that allows external custom services and applications to interact with it directly, and an *event server* that allows listener services to be notified when a worklet is selected or an exception raised during a process execution. Of particular interest are the ways in which the service can now be more easily introduced into learning situations.

2 The Worklet Gateway Client API

The `WorkletGatewayClient` is a client side class that can be used by custom services [5] to interact with the Worklet Service via an API. It provides two main functionalities:

[1] See [1] for more details on the design and implementation of the Worklet Service

– creating, accessing, updating and evaluating Ripple Down Rules (RDR) directly, circumventing the need to work with the Windows-based Rules Editor application (see Section 3); and
– adding and removing worklet event listeners (see Section 4).

An instance of `WorkletGatewayClient` can be used by custom services and applications to first create a session with the Worklet Service, then use the returned *session handle* in subsequent API method calls. Any service or application registered with the YAWL Engine is trusted by the Worklet Service, and so the same credentials can used to establish the connection (see Listing 1.4 for an example of how to connect and use the API).

3 Working with Ripple Down Rules

Since the primary aim of the Worklet Service is to support processes enacted by the YAWL engine, each rule set is typically uniquely identified for a particular specification by its `YSpecificationID` object. However, to generalise RDR support for other custom services and applications, rule sets can now also be uniquely identified by any unique string value. Thus, for each client method there are variations to support the fact that either a specification identifier or a generalised name may be used to identify rules.

Listing 1.1. Examples of using the Worklet Gateway Client 'get' methods

```java
public void example throws IOException {
    YSpecificationID specID = new YSpecificationID(null, "0.1",
            "someSpecification");
    String processName = "someProcess";
    RdrMarshal marshal = new RdrMarshal();
    WorkletGatewayClient client = new WorkletGatewayClient();
    String handle = client.connect("admin", "YAWL");

    // get a Rule Set
    String ruleSetXML = client.getRdrSet(specID, handle);
    if (! successful(ruleSetXML)) throw new IOException(ruleSetXML);
    RdrSet ruleSet = marshal.unmarshalSet(ruleSetXML);

    // get a Rule Tree
    String ruleTreeXML = client.getRdrTree(processName, null,
            RuleType.CasePreconstraint, handle);
    if (! successful(ruleTreeXML)) throw new IOException(ruleTreeXML);
    RdrTree ruleTree1 = marshal.unmarshalTree(ruleTreeXML);
    ruleTreeXML = client.getRdrTree(specID, "Treat", RuleType.ItemSelection,
            handle);
    if (! successful(ruleTreeXML)) throw new IOException(ruleTreeXML);
    RdrTree ruleTree2 = marshal.unmarshalTree(ruleTreeXML);

    // get a Rule Node
    String ruleNodeXML = client.getNode(specID, null,
            RuleType.CasePreconstraint, 1, handle);
    if (! successful(ruleNodeXML)) throw new IOException(ruleNodeXML);
    RdrNode node1 = marshal.unmarshalNode(ruleNodeXML);
    ruleNodeXML = client.getNode(processName, "Archive",
            RuleType.ItemSelection, 4, handle);
    if (! successful(ruleNodeXML)) throw new IOException(ruleNodeXML);
    RdrNode node2 = marshal.unmarshalNode(ruleNodeXML);
}
```

Get Methods Firstly, there are a number of *get* methods, which can be used to retrieve an individual rule node, a rule tree (a set of connected nodes for a particular rule type and task id), or an entire rule set (the set of rule trees for a particular specification or process). Each of the *get* methods, and in fact all of the client methods, return a String value representing a successful result or an appropriate error message. A successful result for a *get* method contains an XML String representation of the object retrieved, which can be unmarshalled into the appropriate object using the `RdrMarshal` class, as shown in the examples in Listing 1.1.

Evaluation Methods The rule sets maintained by the Worklet Service can be evaluated against a given data set via the client API at any time. Like the *get* methods, there are variations that accept a specification identifier or a generalised name, and if evaluating an item-level rule tree, a task id. There are three possible results when a rule set is evaluated:

1. No rule tree exists for the given specification id or process name, and/or task id, and rule type (as required). An error message to that effect is returned.
2. A rule tree exists for the given parameters, but none of its nodes' conditions were satisfied. An error message to that effect is returned.
3. A rule tree exists for the given parameters, and the conditions of at least one its nodes were satisfied. An `RdrConclusion` object is returned.

For a rule set associated with a YAWL specification, an `RdrConclusion` object represents a set of *primitives*, each comprising an *action* and a *target*, and provides methods for iterating through the primitives set. Of course, if the Worklet Service is being used merely as an RDR store/evaluator, and thus there is no intention to have the rules evaluated by the Worklet Service in relation to a YAWL process instance, then the conclusion of a rule node may contain any data of relevance to an external service or application. For example, the conclusion of an *ItemSelection* rule node contains one primitive, with an action of 'select' and a target naming the worklet to select. The conclusion of an exception rule type may contain any number of primitives (see the Worklet Service chapter of the YAWL User Manual for more information). An example of an *evaluate* method call is shown in Listing 1.2.

Listing 1.2. Example using the Worklet Gateway Client 'evaluate' method

```java
public void example throws IOException {
    YSpecificationID specID = new YSpecificationID(null, "0.1",
            "someSpecification");
    RdrMarshal marshal = new RdrMarshal();
    WorkletGatewayClient client = new WorkletGatewayClient();
    String handle = client.connect("admin", "YAWL");

    // evaluate a Rule Type against a data set
    String dataStr = "<data><Age>40</Age><Fracture>true</Fracture></data>";
    Element data = JDOMUtil.stringToElement(dataStr);
    String conclusionXML = client.evaluate(specID, "Treat", data,
            RuleType.ItemSelection, handle);
    if (! successful(conclusionXML)) throw new IOException(conclusionXML);
    RdrConclusion conclusion = marshal.unmarshalConclusion(conclusionXML);
}
```

Adding a Rule Node RDR are structured in such a way that rule nodes can be added at any time, but never deleted, since deleting a node would 'break' the tree's internal structure. The client API provides a method to add a rule node to a rule set for a specification or process, with similar variations to those mentioned above for the different rule types. An example of adding a node is shown in Listing 1.3.

Listing 1.3. Example using the Worklet Gateway Client to add a new Rule Node

```
 1  public void example throws IOException {
 2      YSpecificationID specID = new YSpecificationID(null, "0.1",
 3              "someSpecification");
 4      RdrMarshal marshal = new RdrMarshal();
 5      WorkletGatewayClient client = new WorkletGatewayClient();
 6      String handle = client.connect("admin", "YAWL");
 7
 8      // add a Rule Node
 9      String cornerStr = "<data><Age>40</Age><Fracture>true</Fracture></data>";
10      Element eCornerstone = JDOMUtil.stringToElement(cornerStr);
11      RdrConclusion conclusion = new RdrConclusion();
12      conclusion.addPrimitive(ExletAction.Suspend, ExletTarget.Case);
13      conclusion.addCompensationPrimitive("myWorklet");
14      conclusion.addPrimitive(ExletAction.Continue, ExletTarget.Case);
15      RdrNode node = new RdrNode("Age=>75", conclusion, eCornerstone);
16      String newNodeXML = client.addNode(specID, "Treat",
17              RuleType.ItemPreConstraint, node, handle);
18      if (! successful(newNodeXML)) throw new IOException(newNodeXML);
19      node = marshal.unmarshalNode(newNodeXML);
20  }
```

When adding a rule node, values for *condition*, *conclusion* and *cornerstone* are all that is required, as shown in line 15 of Listing 1.3. The Worklet Service will determine the appropriate location in its rule tree for the new node[2]. The *cornerstone* data is particularly important when adding a node to an existing tree, as it is used to determine the proper location of the node. If no rule set currently exists for the specification id or process name given, then one is created. Further, if no rule tree currently exists for the specification id or process name, and rule type (and task id if required) given, then one is created, and the node is added as the first node in the tree (after the root node).

User-defined Functions The condition defined for each rule node is an expression that must evaluate to a final boolean (true/false) value. Expressions can be formed using the usual numeric, comparison and logical operators, and operands consisting of literal values, and/or the values referenced by the name of case and/or workitem data variables. Alternately, an XQuery expression may be used, which is useful if you need to query case or workitem data stored as a complex data type.

In addition, conditional expressions support the use of *user-defined functions* that are defined to evaluate the available data in a particular way, and in turn may be inserted into complex, conjunctive expressions. These functions can pass arguments consisting of literal values, and case or workitem data variable names,

[2] Interested readers can find details of how the Service determines the location of a new rule within the tree on pages 59–60 of [1]

which will be substituted with their actual runtime values before being passed to the function[3].

For work item level rules, a special data structure, called `this`, can be passed as an argument to a user-defined function (e.g. `myFunc(this)`). The structure is assigned an XML string representing the complete work item record, so that specification and task descriptors, and state and timing values may be accessed within the user-defined function.

4 Worklet Event Listeners

Each time the Worklet Service selects a worklet for a work item, or raises an exception for a work item or case, details of the event are announced to registered listener classes. Listeners can use these event announcements to perform additional processing as required, and make it possible to create *exception handling service chains*.

An abstract class called `WorkletEventListener` is the base class for all worklet listeners. This class is a `HttpServlet` that handles all of the mechanics involved in receiving notifications from the Worklet Service and transforming them into five abstract method calls that are to be implemented by extending classes:

– **caseLevelExceptionEvent** is called each time a case level exception is raised, and passes descriptors for case-level data, the type of exception raised, and the `RdrNode` that evaluated to true and so triggered the raising of the exception.
– **itemLevelExceptionEvent** is called each time a item level exception is raised, and passes descriptors as above in addition to the work item record that was the 'subject' of the exception being raised.
– **selectionEvent** is called each time a worklet selection occurs, i.e. when a worklet is substituted for a work item, passing descriptors for case-level data, the work item that the worklet was selected for, and the rule node that evaluated to true and so triggered the selection.
– **constraintSuccessEvent** is called each time a case or work item that has pre- or post-constraint rules defined has had those rules evaluated and the case or work item passes the constraints (i.e. no rules were satisfied and so no exception was raised).
– **shutdown** is called when the Worklet Service is shutting down, and allows implementers to take any necessary action.

Once the listener class implementation is complete, the extra servlet definition section needs to be added to the web.xml of the implementing (listener) service, and then at runtime the listener is to be registered with the Worklet Service, as shown in Listing 1.4.

[3] See the YAWL User and Technical Manuals for details on the definition of user-defined functions

Listing 1.4. Registering a listener with the Worklet Service

```
public boolean registerWorkletListener() {
    WorkletGatewayClient client = new WorkletGatewayClient();
    try {
        String url = "http://localhost:8080/myService/workletlistener";
        String handle = client.connect("admin", "YAWL");
        return successful(client.addListener(url, handle));
    }
    catch (IOException ioe) {
        log.error("Failed␣to␣␣register␣listener:␣" + ioe.getMessage());
    }
    return false;
}
```

5 Conclusion

The additions to the Worklet Service outlined above allow it to be more easily integrated into new and existing projects. Since rules can now be easily added and evaluated directly via the API, there is no longer the requirement to use the Windows-based Rules Editor for that purpose, and therefore it is no longer necessary to undertake the learning curve associated with using the Rules Editor before the benefits provided by the Service can be realised. In addition, the ability to define and evaluate rules sets for environments external to YAWL process executions broadens the application of the Service's functionality to much wider domains. Finally, the ability for external applications and services to be notified when an exception or worklet selection has occurred during a process execution, and when a case or work item has passed a constraint evaluation, opens the Service up to new areas of functional support, particularly to the facilitation of exception handling service chains.

An example of a novel use of these extensions can be seen in the development of *blended workflow*, which integrates two specifications for the same workflow process, one structured (YAWL) and one ad-hoc (goal-based), allowing users to switch between two views of the same process instance [4].

References

1. M. Adams. *Facilitating Dynamic Flexibility and Exception Handling for Workflows*. PhD thesis, Queensland University of Technology, Brisbane, Australia, 2007.
2. M. Adams, A.H.M. ter Hofstede, W.M.P. van der Aalst, and D. Edmond. Dynamic, extensible and context-aware exception handling for workflows. In R. Meersman and Z. Tari, editors, *Proceedings of the 15th International Conference on Cooperative Information Systems (CoopIS'07)*, volume 4803 of *Lecture Notes in Computer Science*, pages 95–112, Vilamoura, Portugal, November 2007. Springer.
3. M. Adams, A.H.M. ter Hofstede, D. Edmond, and W.M.P. van der Aalst. Worklets: A service-oriented implementation of dynamic flexibility in workflows. In R. Meersman and Z. Tari et al., editors, *Proceedings of the 14th International Conference on Cooperative Information Systems (CoopIS'06)*, volume 4275 of *Lecture Notes in Computer Science*, pages 291–308, Montpellier, France, November 2006. Springer.

4. D. Passinhas, M. Adams, B. O. Pinto, R. Costa, A. Rito Silva, and A.H.M. ter Hofstedet. Supporting blended workflows. In N. Lohmann and S. Moser, editors, *Proceedings of the Demonstration Track of the 10th International Conference on Business Process Management (BPM 2012)*, volume 940 of *CEUR Workshop Proceedings*, pages 23–28, Tallinn, Estonia, September 2012. CEUR-WS.org.
5. A.H.M. ter Hofstede, W.M.P. van der Aalst, M. Adams, and N. Russell. *Modern Business Process Automation: YAWL and its Support Environment.* Springer, 2010.