Worklet Selector
Custom YAWL Service

Manual

Beta – 7 Release
Preface

This manual contains instructions for installing & using the *Worklet Dynamic Process Selection Custom Service for YAWL*.

Each section describes one part in the process of setting up and using the worklet service. It is probably best to work through the manual from start to finish the first time it is read. This manual focuses on the practical use of the worklet service. For those interested, a more technical description of the inner operations of worklets and the rule sets that support them can be found [here](#) or a more concise version [here](#).

All of the example specifications, rule sets, and so on referred to in this manual can be found in the “worklet repository” distributed with the service as part of the YAWL Beta 7 release.

This icon indicates a hands-on method or instruction.
# Contents

Document Control ........................................................................................................................................ ii
Preface ......................................................................................................................................................... ii
Contents .................................................................................................................................................... iii

1. Welcome to the YAWL Worklet Service .......................................................................................... 1
   What is a Custom YAWL Service? ................................................................................................. 1
   What is the YAWL Worklet Service? ......................................................................................... 1

2. Installation ............................................................................................................................................... 3
   Worklet Installation Package ........................................................................................................ 3
   Installation .......................................................................................................................................... 3
   Installing the Rules Editor ............................................................................................................. 6

3. Creating Worklet Enabled Specifications ..................................................................................... 7
   Top-level or Parent Specifications ............................................................................................... 7
   Worklet Specifications .................................................................................................................. 9

4. The Worklet Rules Editor .................................................................................................................. 12
   Worklet Rule Sets ......................................................................................................................... 12
   The Rules Editor .......................................................................................................................... 13
   Browsing an Existing Rule Set ................................................................................................... 14
   Adding a New Rule ....................................................................................................................... 16
   Dynamic Replacement of an Executing Worklet ....................................................................... 19
   Creating a New Rule Set ............................................................................................................. 21

5. Walkthrough – Using the Worklet Service .................................................................................... 23
   A. Worklet-Enabled Atomic Task Example ............................................................................. 23
   B. Worklet-Enabled Multiple Instance Atomic Task Example ............................................. 28
1. Welcome to the YAWL Worklet Service

What is a Custom YAWL Service?

An important point of extensibility of the YAWL system is its support for interconnecting external applications and services 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 workitems or a change of status of existing workitems.

Custom YAWL services interact with the YAWL engine through XML/HTTP messages via certain endpoints, some located on the YAWL engine side and others on the service side. 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 send and receive XML messages to and from the engine.

More specifically, Custom YAWL services are able to check-out and check-in workitems from the YAWL engine. They receive a message when an item is enabled, and therefore can be checked out. When the Custom YAWL service is finished with the item it can check it back in, in which case the engine will set the work item to be completed, and proceed with the execution.

What is the YAWL Worklet Service?

The Worklet Dynamic Process Selection Service for YAWL provides the ability to substitute a workitem in a YAWL process with a dynamically selected "worklet" - a discrete YAWL process that acts as a sub-net for the workitem and so handles one specific task in a larger, composite process activity.

An extensible repertoire (or catalogue) of worklets is maintained. Each time the service is invoked for a workitem, a choice is made from the repertoire based on the data within the workitem, using a set of rules to determine the most appropriate substitution.

The workitem is checked out of the YAWL engine, and then the selected worklet is launched as a separate case. The data inputs of the original workitem are mapped to the inputs of the worklet. When the worklet has completed, its output data is mapped back to the original workitem, which is then checked back into the engine, allowing the original process to continue.

Worklets can be substituted for atomic tasks and multiple-instance atomic tasks. In the case of multiple-instance tasks, a worklet is launched for each child workitem. Because each child workitem may contain different data, the worklets that substitute for them are individually selected, and so may all be different.
The repertoire of worklets can be added to at any time, as can the rules base used for the selection process. Thus the service provides for dynamic ad-hoc change and process evolution, without having to resort to off-system intervention and/or system downtime, or modification of the original process specification.
2. Installation

Worklet Installation Package

Within the YAWL web release YAWL-Beta-7-Executable-WebServer-Version.zip there is a file called workletSelector.war – this is the worklet web application component. In addition to the custom service files, workletSelector.war contains a file called workletRepository.zip, which contains a required set of file directories where worklets are stored, logs are written to and so on.

Installation

Open workletSelector.war in an unzipping tool (e.g. WinZip). Locate and extract the workletRepository.zip to a temporary directory.
First, the repository should be installed. This needs to be done first because you need to let the service know where to find it (i.e. the path to the repository). The repository stores various components needed for the successful operation of the service. The repository has this structure:

- The logs folder is where the service writes log files during its operation. In particular, the workletSelector.log contains a description of all stages of each operation. The eventLog.csv file logs key events of the service. All are plain text files.

- The rules folder contains the sets of rules using during the selection process. Rules files have an XML format. Each YAWL specification that uses the service will have a corresponding rules file of the same name, except with an extension of xrs (XML rule set).

- The rulesEditor folder contains a tool that allows you to add rules to a rule set for a specification (see Appendix A for more details).

- The selected folder contains a set of files, in XML format, that are essentially log files that capture the results of each selection process. These files have an xws extension (XML Worklet Selection). These files are used by the rules editor to enable easy addition of new rules for a specification.

- The worklets folder contains the worklet specification files. These files are YAWL specifications that are run as substitutes for workitems as required.

Several of the folders contain examples.

To install the repository, simply extract the contents of workletRepository.zip to a folder of your choice. If you choose to install it to C:\Worklets, it will make the next stage of the process a little easier.

Once the repository is installed, reopen the workletSelector.war file in an unzip tool.
Find and open the file `workletSelector.properties` in a text editor. It contains one line:

```plaintext
repositoryDir=C:/Worklets/repository/
```

Change the path to where you installed the repository. Save and close the text editor. Your unzipping tool may ask you if you want to update the changed file – answer ‘Yes’. If not, copy the changed file into the war file.

Once that is done, refer to the YAWL Installation manual on how to copy the war file into the webapps directory of your Tomcat installation.

If Tomcat is not running, start it now. The service will automatically be installed. Congratulations – the Worklet Service installation is complete!
Installing the Rules Editor

The Rules Editor allows for the addition of new rules to the rule sets of specifications. It is a .NET based application, so has the following requirements:

- Operating System: Windows 98SE or better
- The Microsoft .NET framework (any version). If you don’t have the framework installed, it can be downloaded from Microsoft.

The Rules Editor tool can be found in the `rulesEditor` folder of the worklet repository. It can be run directly from there – no further installation is required.

For more details on the Rules Editor and how to use it, refer to Section 4. The Worklet Rules Editor.
3. Creating Worklet Enabled Specifications

Fundamentally, a worklet is simply a workflow specification that has been designed to perform one part of larger or ‘parent’ specification. However, it differs from a decomposition or sub-net in that it is dynamically assigned to perform a particular task at runtime, while sub-nets are statically assigned at design time. So, rather than being forced to define all possible “branches” in a specification, the worklet service allows you to define a much simpler specification that will evolve dynamically as more worklets are added to the repertoire for a particular task.

The first thing you need to do to make use of the service is to create a number of YAWL specifications – one which acts as the top-level (or manager or parent) specification, and one or more worklets which will be dynamically substituted for particular top-level tasks at runtime.

The YAWL Editor is used to create both top-level and worklet specifications. A knowledge of creating and editing YAWL specifications, and the definition of data variables and parameters for tasks and specifications, is assumed. For more information on how to use the YAWL Editor, see the YAWL Editor User Manual.

Before opening the YAWL Editor, make sure that the worklet service is correctly installed and that Tomcat is running (see the Worklet Service Installation Manual and/or the YAWL Engine Installation Manual for more information).

First, a top-level specification needs to be defined.

Top-level or Parent Specifications

To define a top-level specification, open the YAWL Editor, and create a process specification in the usual manner. Choose one or more tasks in the specification that you want to have replaced with a worklet at runtime. Each of those tasks needs to be associated via the YAWL Editor with the worklet service.

For example, Figure 1 shows a simple specification for a Casualty Treatment process. In this process, we want the Treat task to be substituted at runtime with the appropriate worklet based on the patient data collected in the Admit and Triage tasks. That is, depending on each patient’s actual physical data and reported symptoms, we would like to run the worklet that best handles the patient’s condition.

Worklets may be associated with an atomic task, or a multiple-instance atomic task. Any number of worklets can be associated with an individual task, and any number of tasks in a particular specification can be associated with the worklet service.
Here, we want to associate the Treat task with the worklet service. To do that, right click on the task, then select Task Decomposition Detail from the popup menu. The Update Task Decomposition dialog is shown (Figure 2). This dialog shows the variables defined for the task – each one of these maps to a net-level variable, so that in this example all of the data collected from a patient in the first two tasks are made available to this task. The result is that all of the relevant current case data for this process instance can be used by the worklet service to enable a contextual decision to be made. Note that it is not necessary to map all available case data to a worklet enabled task, only that data required by the service to make an appropriate decision. How this data is used will be discussed later in this manual.
The list of task variables in Figure 2 also show that most variables are defined as ‘Input Only’ – this is because those values will not be changed by any of the worklets that may be executed for this task; they will only be used in the selection process. The last three variables are defined as ‘Input & Output’, so that the worklet can “return”, or map back to these variables, data values that are captured during the worklet’s execution.

Note also that the dialog has a section at the bottom called *YAWL Registered Service Detail*. It is here that the task is associated with the worklet service by choosing the worklet service from the list of available services. Note that list of services will only be seen if Tomcat is currently running and it has those services installed. Note also that the description shown of the worklet service will be the description entered into the YAWL Engine when the service was registered (Figure 3 – see the worklet installation manual for more details) and so may differ from that shown in Figure 2.

That is all that’s required to make the top-level specification worklet enabled. Next, we need to create one or more worklet specifications to execute as substitutes for the worklet-enabled task.

**Worklet Specifications**

When the *Casualty Treatment* top-level specification is executed, the YAWL Engine will notify the worklet service when the worklet-enabled *Treat* task becomes enabled. The worklet service will then examine the data in the task and use it to determine which worklet to execute as a substitute for the task. Any or all of the data in the task may also be mapped to the worklet case as input data. Once the worklet instance has completed, any or all of the available output data of the worklet case may be mapped back to the *Treat* task to become its output data, and the top-level process will continue.

A worklet specification is a standard YAWL process specification, and as such is created in the YAWL Editor in the usual manner. Each of the data variables that are required to be passed from the parent task to the worklet specification need to be defined as net-level variables in the worklet specification.

Figure 4 shows a simple example worklet to be substituted for the *Treat* top-level task when a patient complains of a fever.
In itself, there is nothing special about the TreatFever specification. Even though it will be considered by the worklet service as a member of the worklet repertoire and may thus be considered a “worklet”, it is a standard YAWL specification and as such may be executed directly by the YAWL engine without any reference to the worklet service.

As mentioned previously, those data values that are required to be mapped from the parent task need to be defined as net-level variables in the worklet specification. Figure 5 shows the net-level variables for the TreatFever task.

Note the following:

- Only a sub-set of the variables defined in the parent Treat task are defined here (see Figure 2). It is only necessary to map from the parent task those variables that contain values to be displayed to the user, and/or those variables that the user will supply values for to be passed back to the parent task when the worklet completes.

- The definition of variables is not restricted to those defined in the parent task. Any additional variables required for the operation of the worklet may also be defined here.
Only those variables that have been defined with an identical name and data type to variables in the parent task and with a *Usage* of ‘Input Only’ or ‘Input & Output’ will have data passed into them from the parent task when the worklet is launched.

Only those variables that have been defined with an identical name and data type to variables in the parent task and with a *Usage* of ‘Output Only’ or ‘Input & Output’ will pass their data values back to the parent task when the worklet completes.

In Figure 5, it can be seen that the values for the *PatientID*, *Name* and *Fever* variables will be used by the *TreatFever* worklet as display-only values; the *Notes*, *Pharmacy* and *Treatment* variables will receive values during the execution of the worklet and will map those values back to the top-level *Treat* task when the worklet completes.

The association of tasks with the worklet service is not restricted to top-level specifications. Worklet specifications also may contain tasks that are associated with the worklet service and so may have worklets substituted for them, so that a hierarchy of executing worklets may sometimes exist. It is also possible to recursively define worklet substitutions – that is, a worklet may contain a task that, while certain conditions hold true, is substituted by an instance of the same worklet specification that contains the task.

Any number of worklets can be created for a particular task. For the *Casualty Treatment* example, there are five worklets in the repertoire for the *Treat* task, one for each of the five conditions that a patient may present with in the *Triage* task: Fever, Rash, Fracture, Wound and Abdominal Pain. Which worklet is chosen for the *Treat* task depends on which of the five is given a value of *True* in the *Triage* task.

How the worklet service uses case data to determine the appropriate worklet to execute is described in the next section.
4. The Worklet Rules Editor

Worklet Rule Sets

A specification may contain a number of tasks, one or more of which may be associated with the worklet service. For each specification that contains a worklet-enabled task, the worklet service maintains a corresponding set of rules that determine which worklet will be selected as a substitute for the task at runtime, based on the current case data of that particular instance.

Each worklet-enabled task in a specification has its own discrete rule set. The rule set or sets for each specification are stored as XML data in a disk file that has the same name as the specification, with an “.xrs” extension (XML Rule Set). All rule set files are stored in the rules folder of the worklet repository. For example, the file Casualty_Treatment.xrs contains the worklet rule set for the Casualty_Treatment.xml YAWL process specification. Figure 6 shows an excerpt from that file.

```
<?xml version="1.0" encoding="utf-8"?>
<spec>
  <task name="Treat">
    <ruleNode>
      <id>0</id>
      <parent>-1</parent>
      <trueChild>1</trueChild>
      <falseChild>-1</falseChild>
      <condition>True</condition>
      <conclusion>null</conclusion>
      <cornerstone/>
      <description>root level default node</description>
    </ruleNode>
    <ruleNode>
      <id>1</id>
      <parent>0</parent>
      <trueChild>-1</trueChild>
      <falseChild>2</falseChild>
      <condition>Fever = true</condition>
      <conclusion>TreatFever</conclusion>
      <cornerstone/>
      <description>Fever = true</description>
    </ruleNode>
  </task>
</spec>
```

*Figure 6: Excerpt of rule set file Casualty_Treatment.xrs*

Notice that the file specifies a set of ruleNodes for the Treat task. The second ruleNode contains a condition “Fever = True” and a conclusion of “TreatFever”. Thus, when the condition “Fever = True” evaluates to true, the worklet TreatFever is chosen as a substitute for the Treat task. Notice also that each rule node (except the first) has a parent, and may have two child nodes, a true child and a false child.

A worklet rule set is a set of modified Ripple-Down Rules (RDR), which maintains a rule hierarchy in a binary-tree structure. The tree is traversed from the root node along the branches, each node having its condition evaluated. If a
node’s condition evaluates to \textit{True}, and it has a true child, then that child node’s condition is also evaluated. If a node’s condition evaluates to \textit{False}, and there is a false child, then that child node’s condition is evaluated. When a terminal node is reached, if its condition evaluates to \textit{True}, then that conclusion is returned as the result of the tree traversal; if it evaluates to \textit{False}, then the last node in the traversal that evaluated to \textit{True} is returned as the result. The root node (Rule 0) of the tree is always a default node with a default \textit{True} condition and conclusion, and so can only have a true branch.

Of course, to maintain a rule set of any complexity by directly editing the XML formatted file would be daunting, to say the least. To make things much easier, a Rules Editor tool is available, and can be found in the rulesEditor folder of the worklet repository. It can be run directly from there – no further installation is required (depending on the requirements below).

\section*{The Rules Editor}

The Rules Editor allows for the addition of new rules to existing rule sets of specifications, and the creation of new rule sets. It is a .NET based application, so has the following requirements:

\begin{itemize}
  \item Operating System: Windows 98SE or later.
  \item The Microsoft .NET framework (any version). If you don’t have the framework installed, it can be downloaded from Microsoft.
\end{itemize}

When the Rules Editor is run for the first time, the following dialog is displayed:

\begin{figure}[h]
  \centering
  \includegraphics[width=0.5\textwidth]{Rules_Editor_First_Time_Configure.png}
  \caption{Rules Editor First Time Use Message}
\end{figure}

Clicking OK shows the Configuration dialog (Figure 8), where the paths to resources the Rules Editor uses need to be specified. Some default paths are shown, but can be modified directly or by using the browse buttons where available. The following paths must be specified:

\begin{itemize}
  \item \textbf{Worklet Repository}: the path where the worklet repository was installed. The default path shown assumes the Rules Editor was started from the rulesEditor folder of the repository. If it was started from another location, specify the actual path to the repository by editing the path or browsing to the correct location.
\end{itemize}
- **YAWL Editor**: the path to the folder that contains the YAWL Editor (i.e. the file `YAWLEditor1.3.jar`).

- **Worklet Service URI**: the URI to the worklet service. The default URI assumes it is installed locally. If it is remote to the computer running the Rules Editor, then that URI should be entered, ensuring it ends with “/workletSelector”.

Some checks will be made to make sure the paths are valid and you will be asked to correct any that are not. Once the configuration is complete, the main screen will appear. This screen allows you to view each node of the rule set for each worklet-enabled task in a particular specification; to add new rules to the current rule set; and, to create a new rule set for a new specification.

### Browsing an Existing Rule Set

To load a rule set into the Rules Editor, click on the *File* menu, then select *Open*.... The File Open Dialog should open with the *rules* folder of the repository selected. Select the file you wish to open, and then click OK.

Figure 9 shows the main screen with the rule set for the *Casualty Treatment* specification loaded. On this screen, you may browse through each node of the rule set tree and view the various parts of each node.
The screen’s title bar shows the name of the specification associated with the currently loaded rule set.

Under the menu bar, the Task drop-down list contains the name of each worklet-enabled task in the specification. If there is more than one task in the list, selecting a task name will show the rule set for that task.

The RDR Tree panel on the top-left shows all of the rule nodes in this set in their binary tree structure. To make things a little clearer, those nodes on the True branch of a parent have a green icon, those on the False branch of a parent node have a red icon. By clicking on individual nodes, the details of the node are displayed to the right and below the tree panel.

The Selected Node panel on the screen bottom shows the details of the selected node. At runtime, if the condition evaluates to true, and there are no more child nodes on the true branch of this node, then the worklet named will be returned as the result of this tree traversal.

The Cornerstone Case panel on the top right shows the complete set of case data that, in effect, caused the creation of the currently selected rule (see Adding a new rule below for more details). In Figure 9, the Cornerstone Case data shows that the variable Fever had a value of true, while the variables Rash, Wound and Fracture have value of false.

Figure 9: Rules Editor Main Screen
**Adding a New Rule**

There are occasions when the worklet returned for a particular case, while the correct choice based on the current rule set, is an inappropriate choice for a particular case. For example, if a patient in a *Casualty Treatment* case presents with a rash *and* a heart rate of 190, while the current rule set correctly returns the *TreatRash* worklet, it may be desirable to treat the racing heart rate before the rash is attended to. In such a case, as the worklet service begins an instance of the *TreatRash* process, it is obvious that a new rule needs to be added to the rule set so that cases that have such data (both now and in the future) will be handled correctly.

To add a new rule to a rule set, it is first necessary to open the rule set in the Rules Editor (as described above). Then, click *Rules* on the top menu, then *Add...* to open the (initially blank) *Add New Rule* screen.

Every time the worklet service selects a worklet to execute as a substitute for a specification instance’s workitem, a file is created that contains certain descriptive data about the selection process. These files are stored in the *selected* folder of the worklet repository. The data stored in these files are again in XML format, and are named according to the following format:

\[\text{Caseid-Taskid-} \text{workletname}.xws\]

For example: 4.3-3_Treat - TreatRash.xws (xws for Xml Worklet Selection). The case id and task id refer to the substituted work item, not the worklet case instance.

Thus to add a new rule after an inappropriate selection, the particular file for the case that was the catalyst for the rule addition must be located and loaded into the Rules Editor.

From the *Add New Rule* screen, click the *Open...* button to load the selection information from file. The File Open dialog that displays should open in the *selected* folder of the repository. Select the appropriate file for the case in question then click OK. For example, selecting 4.3-3_Treat - TreatRash.xws means that case instance 4.3 had selected, for the *Treat* task, the worklet *TreatRash*. Note that the file chosen must be a selection for an instance of the specification that matches the specification rule set loaded on the main screen (in other words, you can’t attempt to add a new rule to a rule set that has no relation to the xws file opened here). If the specifications don’t match, an error will display.

Figure 10 shows the Add New Rule screen with the file 4.3-3_Treat - TreatRash.xws loaded. The *Cornerstone Case* panel shows the case data that existed for the creation of the original rule that resulted in the selection. The *Current Case* panel shows the case data for the current case – that is, the case that is the catalyst for the addition of the new rule.

The *New Rule Node* panel is where the details of the new rule are added. Notice that the id’s of the parent node and the new node are shown as read only – the Rules Editor takes care of where in the rule tree the new rule node is to be placed, and whether it is to be added as a true child or false child node.
Since we have the case data for the original rule, and the case data for the new rule, to define a condition for the new rule it is only necessary to determine what it is about the current case that makes it require the new rule to be added. That is, it is only where the case data items differ that distinguish one case from the other, and further, only a subset of that differing data is relevant to the reason why the original selection was inappropriate.

For example, there are many data items that differ between the two case data sets shown in Figure 10, such as PatientID, Name, Sex, Blood Pressure readings, Height, Weight and Age. However, the only differing data item of relevance here is HeartRate – that is the only data item that, in this case, makes the selection of the TreatRash worklet inappropriate.

![Figure 10: Add New Rule Screen](image)

Clicking on the line “HeartRate = 190” in the Current Case panel copies that line to the Condition input in the New Rule Node panel. Thus, the condition for the new rule has been easily created.

Note that it is not necessary to define the rule as “Rash = True & HeartRate = 190”, as might first be expected, since this new rule will be added to the true branch of the TreatRash node. By doing so, it will only be evaluated if the condition of its parent, “Rash = True”, evaluates to True. Therefore, any rule nodes added to the true branch of a parent become exception rules of the parent. In other words, this particular tree traversal can be interpreted as: “if Rash is True then return TreatRash except if HeartRate is also 190 then return ???” (??? = whatever worklet we decide to return for this rule – see more below).
Now, the new rule is fine if, in future cases, a patient’s heart rate is exactly 190, but what if it is 191, or 189, or 250? Clearly, the rule needs to be amended to capture all cases where the heart rate exceeds a certain value; say 175.

While selecting data items from the Current Case panel is fast and easy, it is often the case that the condition needs to be further modified to correctly capture relevant cases. The Condition input allows direct editing of the condition.

Conditions are expressed a strings of operands and operators, and sub-expressions may be parenthesised. The following operators are supported:

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Operators</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>* /</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>2</td>
<td>+ –</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>3</td>
<td>= &gt; &lt; != &gt;= &lt;=</td>
<td>Comparison</td>
</tr>
<tr>
<td>4</td>
<td>&amp;</td>
<td>Logical AND</td>
</tr>
<tr>
<td></td>
<td>]</td>
<td>Logical OR</td>
</tr>
<tr>
<td></td>
<td>!</td>
<td>Logical NOT</td>
</tr>
</tbody>
</table>

All conditions must finally evaluate to a Boolean value.

To make the condition for the new rule more appropriate, the condition “HeartRate = 190” should be edited to “HeartRate > 175”.

After defining a condition for the new rule, the name of the worklet to be executed when this condition evaluates to true must be entered in the Worklet field of the New Rule Node panel (refer Figure 10). This input is a drop-down list that contains the name of all the worklets in the worklets folder of the worklet repository. An appropriate worklet for this rule may be chosen from the list, or, if none are suitable, a new worklet specification can be created.

Clicking the New... button next to the worklets list will open the YAWL Editor so that a new specification can be created. When defining the new worklet, bear in mind that to pass data from the original work item to the worklet, the names and data types of the variables passed must match those of the work item and be created as net-level variables in the worklet specification. Also, all new worklets must be saved to the worklets folder of the repository so that the worklet service can access it.

When the new worklet is saved and the YAWL Editor is closed, the name of the newly created worklet will be displayed and selected in the worklet drop-down list. Figure 11 shows the New Rule Node panel after the definition of the example new rule has been completed. The Description field is optional, but recommended.
Once all the fields for the new rule are complete and valid, click the Save button to add the new rule to the rule tree.

**Dynamic Replacement of an Executing Worklet**

Remember that the creation of this new rule was triggered by the selection and execution of a worklet that was deemed an inappropriate choice for the current case. So, when a new rule is added, you are given the choice of replacing the executing (inappropriate) worklet instance with an instance of the worklet defined in the new rule.

After the Save button is clicked, a message similar to the Figure 12 is shown, providing the option to replace the executing worklet, using the new rule. The message also lists the specification and case id’s of the original work item, and the name and case id of the running worklet instance.

If the Yes button is clicked, then in addition to adding the new rule to the rule set, the Rules Editor will contact the worklet service and request the change. For this process to succeed, the following must apply:

- Tomcat is currently running and the worklet service is correctly installed;
- The Service URI specified in the Rules Editor configuration dialog is valid; and
The worklet originally chosen is currently running.

A message dialog will be shown soon after with the results of the replacement process sent from the worklet service to the Rules Editor, similar to Figure 13.

If the No button is clicked, then the new rule is simply added to the rule set.

Figure 13: Result of Replace Request Dialog

Figure 14 shows the main Rules Editor screen with the new rule added in the correct place in the tree, with the current case data becoming the Cornerstone Case for the new rule.

Figure 14: Main Screen after Addition of New Rule
Creating a New Rule Set

When a new specification has been created that contains a worklet-enabled task, or a task in an existing specification is modified to become worklet-enabled, a new rule set needs to also be created to accommodate the task.

To create a new rule set, click the File menu then select New…. (If there is a rule set currently opened, it must first be closed by clicking File then Close before a new rule set can be created). Figure 15 shows the Create New Rule Set screen.

![Create New Rule Set Screen](image)

Figure 15: The Create New Rule Set Screen

On this screen:

- The Process Identifiers panel is where the names of the specification and worklet-enabled task are defined. These names must exactly match the name of the specification that contains the task this rule set is being created for, and the name of the task itself.

- In the Add Cornerstone Data panel, a set of cornerstone data for the new first rule can be specified. Add a variable name to the Data Label input, and give it a value in the Data Value input, then click the Add button to add it to the set of Cornerstone Case data.

- The New Rule Node 1 panel is virtually identical to the panel on the Add New Rule screen. Here a condition and optional description can be entered, and the worklet for the new rule created or selected from the list.
It is only possible to add one new rule to a newly created rule set using the Rules Editor (in addition to the automatically created root node). This is to protect the integrity of the rule set. Since each subsequent rule is added because of an exceptional case or where the selected worklet does not fit the context of a case, the preferred method is to create a rule set with one rule, and then add rules as they become necessary via the Add New Rule process described earlier. In this way, added rules are based on real case data and so are guaranteed to be valid. In a similar vein, there is no option to modify or delete a rule node within a rule set tree, since to allow it would destroy the integrity of the rule set, because the validity of child rule nodes depend on the conditions of their parents.

After all required fields have been entered, click the Create button to create and save the new rule set. Figure 16 shows the new rule set in the main Rules Editor screen as a result of the creation.

Figure 16: Example of a Newly Created Rule Set
5. Walkthrough – Using the Worklet Service

The worklet repository that comes with the worklet service release contains two example specifications with worklet-enabled tasks, each with an associated rule set and a number of associated worklets. This section will step through the execution of these two examples. A knowledge of how to use the YAWL system is assumed. Before we begin, make sure the worklet service is correctly installed and operational, and then log into the YAWL system.

The easiest and best way to ensure the worklet service in ‘on-the-air’ is to go to the YAWL Administrate page and click on the worklet service’s URI link. If all’s well the service’s welcome page will be displayed (Figure 17).

**Figure 17: Welcome Page for the Worklet Service**

A. Worklet-Enabled Atomic Task Example

The *Casualty Treatment* specification used in the previous sections of this manual is an example of a specification that contains an atomic task (called *Treat*) that is worklet-enabled. We’ll run the example specification to see how worklets operate.

Navigate to the YAWL Administrate page and upload the *Casualty Treatment* specification from the *worklets* folder of the worklet repository. Then, go to the Workflow Specifications page and launch a *Casualty Treatment* case.

The case begins by requesting a patient id and name – just enter some data into each field then click *Submit* (Figure 18).
Go to the *Available Work* page, and the first task in the case (*Admit*) will be listed as an available workitem. Make a note of the case number. Check out the *Admit* workitem, then go to the *Checked Out Work* page, select the workitem and click the *Edit Work Item* button.

The *Admit* workitem simulates an admission to the Casualty department of a hospital, where various initial checks are made of the patient. You’ll see that, in addition to the patient name and id specified earlier, there are a number of fields containing some medical data about the patient. Each field has some default data (to save time), but you may edit any fields as you wish (Figure 19). When done, click the *Submit* button.

Go back to the *Available Work* page and check out the next workitem, *Triage*. Then go to the *Checked Out Work* page and edit the workitem.
The *Triage* task simulates that part of the process where a medical practitioner asks a patient to nominate their symptoms. You’ll see that the patient’s name and id have again been displayed for identification purposes, in addition to 5 fields which approximate the problem. One field should be set to *True*, the others to *False*.

Let’s assume the patient has a fever. Set the *Fever* field to *True*, the rest to *False*, and then submit it (Figure 20).

![Figure 20: Editing the Triage Workitem (detail)](image)

There is nothing special about the first two tasks in the process; they are standard YAWL tasks and operate as expected. However, the next task, *Treat*, has been associated with the worklet service. The *Treat* task simulates that part of the process that follows the collection of patient data and actually treats the patient’s problem.

Of course, there are many medical problems a patient may present with, and so there are just as many treatments, and some treatment methods are vastly different to others. In a typical workflow process, this is the part of the process where things could get very complicated, particularly if we tried to build every possible treatment as a conditional branch into the process model.

The worklet service greatly simplifies this problem, by providing a extensible repertoire of discrete workflow processes (worklets) which, in this example, each handle the treatment of a particular medical problem. By examining the case data collected in the earlier tasks, the worklet service can launch, as a separate case, the particular treatment process for each case.

This method allows for a simple expression of the task in the ‘parent’ process (i.e. a single atomic *Treat* task signifies the treatment of a patient, whatever the eventual treatment process may be) as well as the ability to add to the repertoire of worklets at any time as new treatments become available, without having to modify the original process.
When the *Triage* workitem is submitted, the next task in the process, *Treat*, becomes enabled. Because it is worklet-enabled, the worklet service is notified. The worklet service checks to see if there is a set of rules associated with this workitem, and if so the service checks out the workitem.

When this occurs, the YAWL Engine marks the workitem as executing (externally to the Engine) and waits for the workitem to be checked back in. In the meantime, the worklet service uploads the relevant specification for the worklet chosen as a substitute for the workitem and launches a new case for the specification. When the worklet case completes, the worklet service is notified of the case’s completion, and the service then checks the original workitem back into the Engine, allowing the original process to continue.

We have completed editing the *Triage* workitem and clicked the *Submit* button. Go to the *Available Work* page. Instead of seeing the next workitem listed (i.e. *Treat*), we see that *Test Fever*, the first workitem in the *TreatFever* process, is listed in its place (Figure 21). The *TreatFever* process has been chosen by the worklet service to replace the *Treat* workitem based on the data passed to the service.

Note that the case id for the *Test Fever* workitem is different to the case id of the parent process. Go to the *Workflow Specifications* page to see that a *Casualty Treatment* case is still running, and that the *TreatFever* specification has been loaded and it also has a case running (Figure 22).

Go back to the *Available Work* page, check out the *Test Fever* workitem, and then edit it in the *Checked Out Work* page. Note that if you are logged in as ‘admin’ you’ll also see the checked-out *Treat* workitem on this page – this workitem should not be edited as it has been checked out by the worklet service. Any workitems checked out by external services will only appear here if you are logged in as ‘admin’.

The *Test Fever* workitem has mapped the patient name and id values, and the particular symptom – fever – from the *Treat* workitem checked out by the worklet service. In addition, it has a *Notes* field where a medical practitioner can enter...
observations about the patient’s condition (Figure 23). Enter some information into the Notes field, and then submit it.

Figure 22: TreatFever Specification Uploaded and Launched

Check out the next workitem, Treat Fever, and then edit it. This workitem has two additional fields, Treatment and Pharmacy, where details about how to treat the condition can be entered (Figure 24). Enter some data here, and then submit it.

Figure 23: Test Fever Workitem (detail)

Figure 24: Treat Fever Workitem (detail)
When the *Treat Fever* workitem is submitted, the worklet case is completed. The worklet service maps the output data from the worklet case to matching variables of the original *Treat* workitem, then checks that workitem back in, allowing the next workitem in the *Casualty Treatment* process, *Discharge*, to execute.

Go to the *Available Work* page, and you’ll see that the *Discharge* workitem is available (Figure 25). Edit it to see that the data collected by the *TreatFever* worklet has been mapped back to this workitem. Submit it to complete the case.

![Figure 25: Discharge Workitem with Data Mapped from TreatFever Worklet](image)

**B. Worklet-Enabled Multiple Instance Atomic Task Example**

This walkthrough takes the *List Maker* example from the YAWL Editor User Manual (pp. 46-47) and worklet-enables the *Verify List* task to show how multiple instance atomic tasks are handled by the worklet service.

The specification is called *wListMaker*. The only change made to the original *List Maker* specification was to associate the *Verify List* task with the worklet service using the YAWL Editor. Figure 26 shows the specification.

![Figure 26: The wListMaker specification](image)
Go to the *Administrate* page and upload the *wListMaker* specification from the *worklets* folder of the worklet repository. Then, go to the *Workflow Specifications* page and launch an instance of *wListMaker*.

When the case begins, enter 3 values for the *Bob* variable, as shown in Figure 27 – you will have to click the *Insert after selected* button twice to get three input fields. Make sure you enter the values “one”, “two” and “three” (without the quotes and in any order). Submit the form.

![Figure 27: Start of wListMaker Case with Three ‘Bob’ Values Entered (detail)](image)

Check out and edit the *Create List Items* workitem. Since the values have already been entered there is no more to do here, so click the *Submit* button to continue.

The next task is *Verify List*, which has been associated with the worklet service. Since this task is a multiple instance atomic task, three child instances of the task are created, one for each of the *Bob* values entered previously. The worklet service will determine that it is a multiple instance atomic task and will treat each child workitem instance separately, and launch the appropriate worklet for each based on the data contained in each. Since the data in each child instance is different in this example, the worklet service starts 3 different worklets, called *BobOne*, *BobTwo* and *BobThree*. Each of these worklets contains only one task.

Go to the *Available Work* page. There are three workitems listed, each one the first workitem of a separate case (see Figure 28).
Go to the Workflow Specifications page to see that the BobOne, BobTwo and BobThree specifications have been uploaded and launched by the worklet service as separate cases (Figure 29 – note the case numbers).

Go back to the Available Work page and check out all three workitems.

Go to the Checked Out Work page. If you are logged in as ‘admin’ you’ll see something like Figure 30. Edit each of the Get_Bob workitems, and modify the values as you wish – for this walkthrough, we’ll change the values to “one – five”, “two – six” and “three – seven” respectively.

As you edit and submit each Get_Bob workitem, notice that the corresponding Verify List workitem is automatically checked in by the worklet service (you’ll only see this if you are logged in as ‘admin’). Since the Bob worklets contains only one task, editing and submitting this workitem also completes the worklet case.
After the third workitem has been edited and submitted, and so the third Verify List workitem is checked back into the Engine by the worklet service, the Engine determines that the Verify List workitem has completed and so the original process continues to its final workitem, Show List.

Check out and edit the Show List workitem to show the changes made in each of the Get_Bob worklets have been mapped back to the original case (Figure 31).

**Figure 30:** The User and Worklet Service Checked Out Items

<table>
<thead>
<tr>
<th>ID</th>
<th>Task Description</th>
<th>Enable Time</th>
<th>Start Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2:3_Verify_list</td>
<td>Verify List</td>
<td>Dec 18 12:36:54</td>
<td>Dec 18 12:36:55</td>
</tr>
<tr>
<td>7.1:2_Get_Bob_Two</td>
<td>Get_Bob_Two</td>
<td>Dec 18 12:36:59</td>
<td>Dec 18 12:36:55</td>
</tr>
<tr>
<td>4.4:3_Verify_list</td>
<td>Verify List</td>
<td>Dec 18 12:36:54</td>
<td>Dec 18 12:36:55</td>
</tr>
<tr>
<td>5.1:2_Get_Bob_One</td>
<td>Get_Bob_One</td>
<td>Dec 18 12:36:57</td>
<td>Dec 18 12:36:55</td>
</tr>
<tr>
<td>4.3:3_Verify_list</td>
<td>Verify List</td>
<td>Dec 18 12:36:54</td>
<td>Dec 18 12:36:55</td>
</tr>
</tbody>
</table>

**Figure 31:** The Show List Workitem Showing the Changes to the Data Values