BEA WebLogic® Integration

Using Worklist

Version 10.2
Revised: March 2008

Contents

1. About Worklist

2. Creating and Managing Worklist Task Plans

3. Using and Customizing User Portal

4. Using Worklist Controls

5. Worklist Event-based Services

6. Extending the Capabilities of your Application with Custom Modules

7. Using Worklist Console

About Worklist

WebLogic Integration Worklist provides mechanisms to interact with human actors from WebLogic platform applications. Worklist is intended to complement BEA WebLogic Integration™ Java Process Definition (JPD) component. JPD is intended to handle automated business processes and does not directly concern itself with the nuances of human interaction. Worklist extends the reach of the JPD component to include human actors where required.

JPD and Worklist components available with WebLogic Integration enables integration of diverse systems, applications, and human participants. Worklist enables human interaction, via the capability of assigning tasks to human users, with business processes. Based on the assigned task, human users can perform actions on the tasks, which can trigger new task assignments to other users or system events. This process flow depends on the higher level business processes.

Worklist may also be used in a stand-alone fashion. In this case, the Worklist User Portal becomes the primary interface for the user into Worklist. Worklist manages tasks and updates them as human users complete the steps needed to complete them.

Worklist enables modeling of task flow via a task plan. The main features of Worklist are:

- A task plan which matches “real world” tasks that spans multiple logical steps and actions required to complete the task. For more information, see Chapter 2, “Creating and Managing Worklist Task Plans”
- A web interface, called Worklist User Portal, that is modular and contains portlets. For more information, see Chapter 3, “Using and Customizing User Portal”. This user portal can also be customized and tailored to meet your specific requirements. For more information, see Customizing Worklist User Portal.

- Task assignment and load balancing functionalities that can be used to schedule and assign tasks to appropriate human users.

A task plan can include one or more steps. Each step represents a distinct phase in the completion of the task. Human actors move tasks through their defined steps by taking actions on the task at each step. Each step defines one or more actions from which the human actor can choose. Actions include "work actions" that have an associated step to which the task will move when the action is taken. This model allows a task to move through a set of defined steps in a manner dictated by the actions taken on the task.

Tasks can be assigned to and claimed by a human actor, and can maintain a number of system and user-defined properties representing data for the task. Human actors specify and alter properties (both user and system) as the steps in a task are completed.

The steps and user properties for a task are defined by a Task Plan and apply to all instances of tasks of that type. Task plans are defined by a Worklist designer prior to the creation of any instance of the task plan.

Key Terms in Worklist

Term	Definition
Task	Represents a unit of work requiring one or more human actors to assist in its completion.
Task Plan	A description of the steps, properties, task assignment rules, and so on that are common to a class of tasks for a given purpose. For more information, see Chapter 2, "Creating and Managing Worklist Task Plans".
Task Plan Host Application	An application that hosts or deploys task plan modules and uses Worklist via a Worklist system instance. In most cases, a Worklist application is also a task plan host application.
Task Plan Path	A hierarchical name representing a task plan's location in the abstract file system for the task plan registry being used by a Worklist system instance. This path is of the form /Folder1/Folder2/.../FolderN/TaskPlanFileName (minus .task extension) and is derived from the location of the `.task` file in task plan host application, relative to the application's root folder.
Task Instance	A task instance created from a task plan. For more information, see "Creating a Task Instance" in Worklist User Portal.

Term	Definition
Step	A phase in the overall completion of a task. The steps for a task are defined by the task plan for the task. Task assignment can occur at any step in a task. For more information, see “Steps” on page 2-11.
Action	A discrete operation that can alter the properties or working state of a task. Some actions (work actions) cause a transition to another step defined for the task (can be to the same or current step), whereas others (assign, return actions) simply change the working state of the task. For more information, see “Actions and Connections” on page 2-12.
User Property	A named or typed data value associated with a task plan. A task plan may have any number of properties to hold data that is required to define the state of a task. All steps in a task plan share access to the same set of user properties. For more information, see “User Properties” on page 2-8.
System Property	A named or typed data value defined by Worklist for all tasks. Some system properties can be edited while others cannot be edited. For more information, see “System Properties” on page 2-8.
Assignee	A human actor or group of actors that have been designated as candidates to work on a task or tasks of a given task plan.
Claimant	A single human actor who has assumed responsibility for completing a step for a given task. An user with sufficient privileges can claim a task on behalf of another user from Worklist Console. For more information, see “Claiming a Task for a User” in Worklist Administration.
Administrative State	A value that describes how the Worklist system is currently treating the task instance. Active tasks are allowed to operate normally, whereas suspended tasks cannot be interacted with (except to resume them). Tasks in an error state can be interacted with in a limited way, or have their error state cleared. For more information, see “Administrative and Working States” on page 2-3.
Working State	A value describing the current association of a task with human actors who can work on the task. Working state can be “Unassigned”, “Assigned”, or “Claimed”. For more information, see “Administrative and Working States” on page 2-3.
Worklist Application	An application that makes use of Worklist via a Worklist system instance. In most cases, a Worklist application hosts its own Worklist system instance, its own task plans, and its own Worklist user portals. In this case, the Worklist application is called a stand-alone Worklist application. For more information, see “Worklist Application” on page 2-9.

Term	Definition
Stand-alone Worklist Application	A Worklist application that hosts its own Worklist system instance, its own task plans, and its own Worklist user and manager portals. By this definition, a stand-alone Worklist application is a Worklist host application, task plan host application, and Worklist portal host application. This is the most common type of application for Worklist, and the type of application that Worklist documentation advocates using in most cases.
Worklist Host Application	An application that hosts a Worklist system instance. In most cases, a Worklist application is also a Worklist host application.
Worklist Portal Host Application	An application that hosts or deploys an instance of the Worklist portals. In most cases, a Worklist application is also a Worklist portal host application.
Worklist System Instance	An instance of the Worklist API and its implementation. A Worklist system instance is configured to connect to a database instance hosting the Worklist system tables. The tasks being managed by a Worklist system instance are the tasks contained in the database instance. In addition, a Worklist system instance can be configured with its own security policies, runtime configuration, and so on.
Worklist Portal Instance	An instance of the Worklist user portal. This is a web application that present a WebLogic Portal user interface for interacting with a Worklist system instance and the tasks it is managing.

Worklist Scenario

In Worklist, a task plan can be modeled using the Workshop for WebLogic design-time environment and then be deployed to run on the server. Once the task plan is deployed, the task instances can be created either by authorized systems or human entities using Worklist Console. *Task Instances* or tasks are based on the task plan and actions performed by employees or the system.

Worklist User Portal provides facilities for Worklist users (not administrators/designers) and allows them to see the tasks assigned to them or the groups they belong to. Worklist User Portal allows users to claim tasks, take actions on them, and alter their properties.

The various stages in a typical Worklist scenario are:

1. Designers create a task plan using the design-time environment.

To address your business specific scenario using Worklist, the first step is to design the tasks required by your organization and then use the Task Plan Editor to create and edit task plans. The Task Plan Editor gives a graphical representation of the steps and actions designed for the task. Since task plans are XML documents, you can also view the task plan details by opening it with the XML Editor or the Text Editor in the Workshop for WebLogic Platform IDE.

The Task Plan Editor provides a drag-and-drop editor for defining the steps of a task plan, and then defining the properties of individual steps and actions in that model. In addition, other Views in the Task Plan perspective are provided that allow you to edit the task plan configuration. For more information, see "About Perspectives" on page 1-6.

After defining the task plan (`.task` file) using the Task Plan Editor, deploy it to the server in order to use it.

For more information, see Chapter 2, "Creating and Managing Worklist Task Plans".

2. Authorized entities create a task instance of the task plan using the Worklist User Portal. For more information, see Chapter 3, "Using and Customizing User Portal".

3. Users work on tasks associated with them from

   – Worklist User Portal or a custom Task user interface (if a custom UI is developed for a step in the task plan or the entire task plan).

     Worklist User Portal allows users to see the tasks assigned to them, claim tasks, take actions on tasks, and update the worklist properties (values) associated with the actions they perform. Worklist User Portal is used by Worklist users, not administrators or designers.

     If Worklist User Portal does not meet the needs of a given task plan, a custom task user interface can be designed and developed by a Developer and associated with the entire task plan or just a step in the task plan. This ensures that the required user interface can be designed without forcing the implementation of a completely new web user interface. For more information, see Customizing the Worklist User Portal.

   – A JPD business process (using methods in Worklist Task or Task Batch control). For more information, see Chapter 4, "Using Worklist Controls".

4. Authorized entities manage and monitor Worklist tasks, users, and the business calendar. Worklist Console provides a navigation menu containing several modules for performing focused operations. For more information, see *Using Worklist Console*.

For a sample Worklist scenario, see *Tutorial: Building a Worklist Application*

About Perspectives

The window layout in the Workshop for WebLogic Platform IDE (Interactive Development Environment) is called a perspective and can be extensively customized. Perspectives are intended to provide related tools for performing specific tasks with specific resources. The initial perspective is called the Workshop perspective (shown in the upper right corner of the window). Several other useful perspectives are provided. You can switch perspectives at any time by choosing Window > Open Perspective.

The Workshop perspective is the standard perspective for developing Java Platform, Enterprise Edition (Java EE) Version 5 (Java EE) enterprise applications with Workshop for WebLogic. Note the Package Explorer view at the left which allows you to move through the projects/folders/files of your workspace.

Information panes (Views) in the workbench can be moved, torn off, displayed side by side, stacked, minimized or maximized. Each file displayed in the editor can be maximized or minimized to icons at the edge of the window (fast views). Menu bars and tool bars can be added, removed or customized.

You can create and save your own perspectives. Workshop for WebLogic will also change the perspective when you perform other tasks. For example, a different perspective is typically used when creating a page flow or creating task plans.

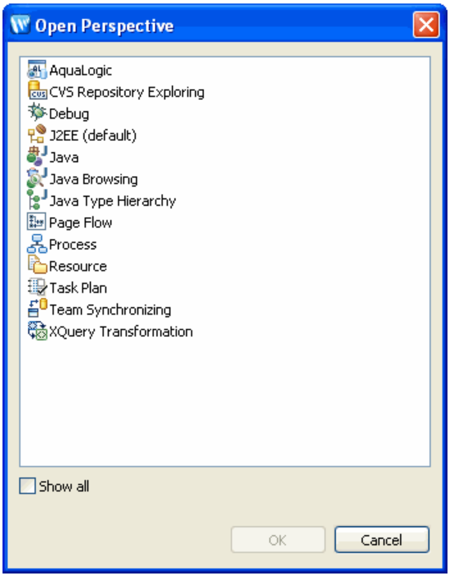
For more information about Eclipse IDE features, access the Workbench User Guide from the Online Help available from Workshop for WebLogic Platform IDE.

Selecting a Perspective

To select a perspective:

1. Click the ⊞ icon at the top right corner of the screen. The Select Perspective dialog appears (see Figure 1-1).

Figure 1-1 Select Perspective Dialog Box



Open Perspective

- AquaLogic
- CVS Repository Exploring
- Debug
- J2EE (default)
- Java
- Java Browsing
- Java Type Hierarchy
- Page Flow
- Process
- Resource
- Task Plan
- Team Synchronizing
- XQuery Transformation

□ Show all

2. Select the perspective and click OK. Although, you can work in any required perspective, Task Plan, Process, and Page Flow are perspectives that are relevant to working with Worklist.

Task Plan Perspective

The Task Plan perspective contains the following Views:

- Package Explorer
- Palette
- Editor
- User Properties
- Properties
- Problems
- Outline
- Console

Process Perspective

The Process perspective contains the following Views:

- Package Explorer
- Node Palette
- Properties
- Annotation
- Data Palette
- Problems
- Navigator
- Outline
- Console

Page Flow Perspective

The Page Flow perspective contains the following Views:

- Console
- JSP Data Palette
- JSP Design Palette
- Merged Projects
- Navigator
- Outline
- Page Flow Editor
- Page Flow Explorer
- Problems
- Properties
- Servers

For more information, see The Page Flow Perspective in *BEA Workshop for WebLogic Platform*.

CHAPTER 2

Creating and Managing Worklist Task Plans

This section provides information about the following:

About Tasks and Task Plans

In Worklist, a task represents an activity that is assigned to a human user. The human user oversees the progress of the task and ensures that it is completed in a correct and timely manner. A task can include one or more steps, and one or more exit points (called terminal steps). Each step represents a distinct phase in the completion of the task. Human actors move tasks through their defined steps by taking actions on the task at each step. Each step defines one or more actions from which the human actor can choose. An action called a "work action" has an associated step to which the task will move when the action is taken. This model allows a task to move through a set of planned steps in a manner dictated by the actions taken on the task.

A terminal step indicates the end of the effective lifetime of the task. Terminal steps can only be reached via a work action taken by a user. Terminal steps can be marked to indicate normal completion (via Complete Step) or abnormal termination (via Abort Step). A task plan must have at least one terminal step.

Tasks can be assigned to and claimed by a human user. Tasks are associated with a set of properties representing task data. Human users can specify and alter properties as the steps in a task are completed.

The steps and properties for a given type of task are defined by a Task Plan and apply to all instances of tasks of that type. All tasks of a given type are then governed by the plan defined for that type of task. Task plans are defined by a designer prior to the creation of any task instance of the type being governed by the plan.

All task instances, regardless of their associated task plan, have properties (called system properties) like:

- Unique identifier and human-readable name
- Administrative State (Active, Error, Suspended, Completed, Aborted)
- Working State (Unassigned, Assigned, Claimed)
- Due Date
- Claimant

In addition, the task plan associated with the task instance defines other elements of a task instance. These include:

- User Properties (Name, Type, Default Value, etc.)
- Steps (Name, Assignee List, Available Actions, etc.)

- Policies (Who can create and administer instances of this task plan)

Administrative and Working States

The Administrative state of a task is a value that describes how the Worklist system is currently treating the task instance.The various administrative states are:

Table 2-1 Administrative States

Administrative State	Description
Aborted	Task instance completed abnormally. No further actions are allowed on steps, and all properties are read-only. The task can be reactivated using the `Reactivate` global action.
Active	Task instance is active and can be interacted with. You can take actions on your steps, set properties, and so on.
Completed	Task instance completed normally. No further actions are allowed on steps, and all properties are read-only. The task can be reactivated using the `Reactivate` global action.
Error	Task instance has encountered some technical error and cannot proceed without administrative intervention. No actions are allowed on steps, but task properties may still be modified. Other types of interaction are allowed on this task only after using the `ClearError` global action.
Suspended	Task instance is not yet complete (or aborted) but temporarily cannot be interacted with (except to resume it). No actions are allowed on steps, and properties are read-only.

The Working state of a task is a value that describes the current association of the task with human actors who can work on the task. The various working states are:

Table 2-2 Working States

Working State	Description
Assigned	Task instance is associated with a list of candidate users who may potentially work on the task. No user has taken responsibility for completing the task.

Table 2-2 Working States

Working State	Description
Claimed	Task instance is associated with a single human user who has taken responsibility for completing the task. This user may later choose to complete the task or return it (putting it back into the Assigned working state).
Unassigned	Task instance is not associated with any human user who can work on the task

The coresponding database numeric value of the worklist task states are:

- Assigned: 0
- Claimed: 1
- Started: 2
- Suspended: 3
- Completed: 4
- Aborted: 5

Global Actions

A number of global actions have been defined that affect the administrative state of the task. These actions may be taken on any task instance regardless of task plan. The actions that effect administrative state are:

Table 2-3 Global Actions

Operation	Description
Abort	Forcefully stop a task. This signifies that the task should be cancelled and should not complete. This operation is performed by the claimant, an administrator, or the task owner. State of the task becomes Aborted.
Assign	Causes a task to move to the ASSIGNED state. The Assignees list must be set when this operation is performed and specify which users can claim the task. This operation can be performed on tasks in a final state, such as COMPLETED or ABORTED. This allows you to work on the task again. This operation can unassign or reassign a task. Assignment can be performed on a single task instances multiple times throughout its life cycle. When assigning a task, an algorithm must be specified to determine how to set the Assignees List. To learn about the algorithms, see Assignment Algorithms. This operation is performed by the task owner, task creator, an assignee, or an administrator.
Claim	Causes a task in the ASSIGNED state to become CLAIMED. A user that is on the Assignees List is set as the claimant of the task. This signifies that a user on the Assignees List has marked ownership of the task and intends to complete it. This operation is performed by a user who wishes to become the claimant for a task, or by an administrator or task owner on behalf of another user.
Clear Error	Force a task instance that has been set into the error state back into the active state. This action will fail if the current user is not an administrator or owner for the task.
Complete	Causes a task in the STARTED state to become COMPLETED. It signifies that the claimant has finished the work required to complete the task, or as much of the work as is possible to do is finished. This operation is performed by the claimant, or by an administrator or task owner on behalf of the claimant.
Create	Creates a new task instance in the ASSIGNED state.
Delete	Delete a task instance. This action will fail if the current user is not an administrator or owner for the task.

Table 2-3 Global Actions

Operation	Description
Reactivate	Bring a task back from a terminal state. This action will fail if the current user is not an administrator/owner for the task. If the task was administratively completed or aborted, the value of the current step will be the step the task was on when the task was aborted or completed. In this case, this step will again be the current step after reactivation of the task instance. If the task completed or aborted as a result of a user action, the final action or step fields will be used to put the task instance back at the step from which the final action was taken.
Resume	Cause a SUSPENDED task to return to the state it was in prior to its suspension. This operation is performed by an administrator or by the task owner.
Return	Remove the current claimant for the task, and force the task back into the Assigned working state. This action will fail if the current user is not an administrator/owner for the task.
Set Error	Force a task instance into the Error Administrative state. This action will fail if the current user is not an administrator or owner for the task.
Start	Causes a task in the CLAIMED state to become STARTED. It signifies that the claimant is starting to work on the task. This operation is performed by the claimant, or by an administrator or task owner on behalf of a claimant.
Stop	Causes a task in the STARTED state to return to the CLAIMED state. It signifies that the claimant is stopping work on the task, possibly temporarily. They can start it again when they are ready to continue working. This operation is performed by the claimant, or by an administrator or task owner on behalf of the claimant.
Suspend	Causes a task to become SUSPENDED. It signifies that the task no longer progresses and should not be worked on, possibly temporarily. The task can be resumed (using the resume operation) when work should continue. This operation is performed by an administrator or task owner.

Worklist Properties

Every task has a number of built-in properties, called the system properties. System properties that can be edited may be specified when you define the task plan, steps and actions in the task plan. In addition, you can define properties that are relevant to your task plan. These are called user properties.

Task Plan Properties

When you design and develop the task plan, you can specify the following properties:

Table 2-4 Task Plan Properties

Property	Description
Completion Due Date	The business date and time by which this task must be completed. This is specified as a time interval, and an optional business calendar name. If there is no business calendar name given, the Worklist global system calendar will be used in the date calculation. The business calendar is used to calculate the absolute date by which the task must be completed. The interval may define a number of days, hours and/or minutes in the D days H hours M minutes format, where D is number of days, H is number of hours, and M is number of minutes. The completion due date of the task is generated based on the interval, and the user or business calendar. For example, if the interval is specified as 10 days and the business calendar is selected. If current date is 1, and the next 5 days are busy based on selected business calendar, then the completed due date for the task would be 15.
Description	A description of the task plan
Owner Name	Name of the User who created the task.
Time Estimate	The estimated time (specified as a business calendar interval, for example, 3min2hour1day or 10 years 5 hours 3 seconds) required to complete the task. This information is used by the default Worklist assignment handler for load balancing. If specified, this value is used as a default value for the time estimate setting on all the steps this task contains.
Version	The version of the task plan. Default value is 1.0.

Table 2-4 Task Plan Properties

Property	Description
Terminal Task Retention	The duration (specified as a business calendar interval, for example, 3min2hour1day or 10 years 5 hours 3 seconds) for which Completed or Aborted tasks should be retained in the runtime store before becoming eligible for purging. For more information, see "Purging Tasks" in Worklist Administration.
Threshold Priority	An integer value that, if specified, conditionally enables the use of user availability information when performing load balancing during task assignment. Availability checking is performed only for those task instances that have a Priority value greater than or equal to the given threshold value.

System Properties

Every task has a number of built-in properties (defined by Worklist and not any individual task plan) that may be set from a constructor or action taken from a step. These properties (called system properties) may be referenced in the property name list of an action or constructor by prefixing the name of the system property with a system defined prefix.

Note: This prefix is locale-sensitive. In the US English locale, the prefix is 'sys:'.

Some of the system properties that can be edited are Task Name, Comment, Priority, Task Completion Due Date, Task Time Estimate, Current Step Completion Due Date, and Current Step Time Estimate.

User Properties

The user properties for a task consist of any number of name/type pairs that define data elements that are maintained for any instance of the task plan. These data elements represent the data passed between participants in completing a given task. A task plan may define any user properties that make sense to help accomplish the goal of that type of task.

At runtime, Worklist administrators (and/or the Worklist API and task control) can modify the properties (both user and system) of a task instance (including adding new user properties) as needed. This is done by creating a copy of the task plan for the task, adding the user property, and applying the new task plan to the task instance using the `Set Task Plan` global action. This ability allows Worklist administrators to respond to special circumstances that may arise on a per-task instance basis.

User properties may be defined with:

- A name (unique within this task plan, maximum of 100 characters in length). See the Requirements for Names Used in a Task Plan section for general guidelines on names.
- Description
- Default value
- Data type

  Worklist defines some system data types, and the system will support the addition of user-defined data types (plugged in by implementing an SPI layer for the new data type). The system data types are:

  - String
  - Integer
  - Float
  - Boolean
  - URL
  - Date/Time as java.util.Date
  - Custom class as JavaBean
  - List
  - Typed XML as XMLBean class (extends XmlObject)

  TaskMessage with MIME Type and Value (MIME Type can be changed dynamically, and value is interpreted according to the MIME Type).

Worklist Application

A Worklist application consists of an EAR and a Web project, which contain all the files and directories that relate to a single unit of work. Optionally, there can be a Utility folder for the Worklist Application that contains the WebLogic System Integration and Control Schemas.

The EAR project corresponds to the Enterprise Application. You can build and deploy this project to create the entire process flow of the enterprise. The task plan that you create is stored under the EARContent folder in the <EAR project name> folder.

The Web project corresponds to the Web application of Worklist that acts as the user interface for the system. The Web project is a part of the EAR project.

Note: `<web project name>` in this document refers to the name of this folder.

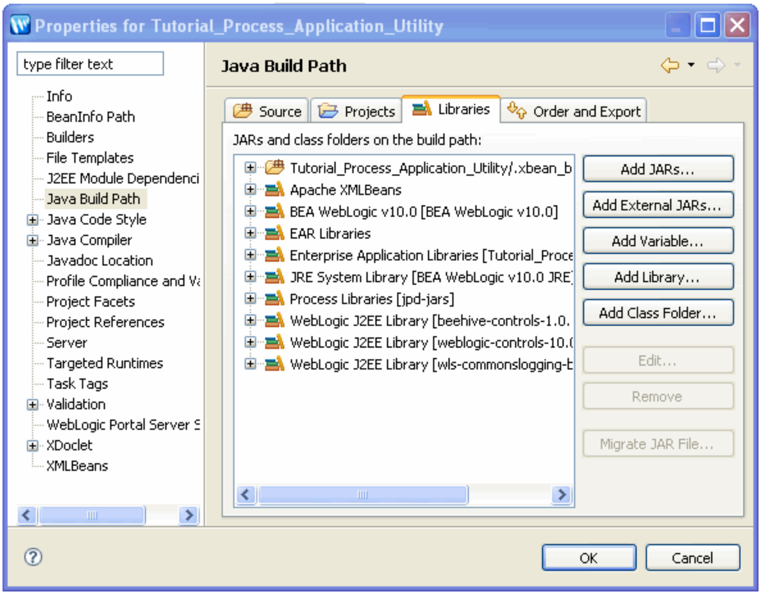
Worklist allows developers to define custom plugin classes of various types. For example, developers can define custom assignment handlers by implementing `com.bea.wli.worklist.api.config.AssignmentHandler`, or implement a custom task event listener by implementing `com.bea.wli.worklist.api.events.TaskEventListener`.

The custom plugin classes must be placed in a utility project so that they are available from the host application's class loader. In order to write these classes, developers need access to Worklist API classes. To implement custom plugin classes, create a utility project to host the plugin code, and then manually add the `worklist-client.jar` library to the build path of the project.

To add the `worklist-client.jar` library to the project build path:

1. In Package Explorer view, right-click on the Utility folder and select Properties. The Properties for <utility folder name> dialog appears.
2. Select Java Build Path in the left navigation pane and select the Libraries tab (see Figure 2-1).

Figure 2-1 Add worklist-client.jar


Properties for Tutorial_Process_Application_Utility
type filter text
Info
BeanInfo Path
Builders
File Templates
J2EE Module Dependenci
Java Build Path
Java Code Style
Java Compiler
Javadoc Location
Profile Compliance and Va
Project Facets
Project References
Server
Targeted Runtimes
Task Tags
Validation
WebLogic Portal Server S
XDoclet
XMLBeans
Java Build Path
Source
Projects
Libraries
Order and Export
JARs and class folders on the build path:
Tutorial_Process_Application_Utility/.xbean_b
Apache XMLBeans
BEA WebLogic v10.0 [BEA WebLogic v10.0]
EAR Libraries
Enterprise Application Libraries [Tutorial_Proce
JRE System Library [BEA WebLogic v10.0 JRE]
Process Libraries [jpd-jars]
WebLogic J2EE Library [beehive-controls-1.0.
WebLogic J2EE Library [weblogic-controls-10.0
WebLogic J2EE Library [wls-commonslogging-b
Add JARs...
Add External JARs...
Add Variable...
Add Library...
Add Class Folder...
Edit...
Remove
Migrate JAR File...
OK
Cancel

3. Click Add External JARs and navigate to the location of the `worklist-client.jar` file (WebLogic installation directory/weblogic92/integration/lib). Select the file and click Open.
4. Click OK.

Now, your utility project should have access to Worklist API classes needed to write the plugin implementation.

There are three Worklist application types:

- Simple Client

  This is a standalone Worklist application that hosts its own Worklist system instance, its own task plans, and its own Worklist user and manager portals. By this definition, a stand-alone Worklist application is a Worklist host application, task plan host application, and Worklist portal host application. The simple client uses the Worklist API to interact with Worklist.

- Worklist System Host

  This is an application that hosts a Worklist system instance. In most cases, a Worklist application is also a Worklist host application. A Worklist system host defines task plans and uses the Worklist User Portal to interact with Worklist.

- Process Host

  This is an application that hosts or deploys an instance of the Worklist portals. In most cases, a Worklist application is also a Worklist portal host application. Process hosts define WebLogic Integration processes and task plans and use Worklist controls to interact programmatically with Worklist and the User Portal to interact with Worklist for human actions.

Overview of Creating Task Plans

The steps and properties for a given type of task are defined by a Task Plan and apply to all instances of tasks of that type. Every step (other than terminal steps including Completed Step and Aborted Step) of a task plan can include actions, which allow the transition of the task instance from one step to another. The actions can be taken by authorized employees or system actors.

Steps

A step is a distinct phase in the life cycle of a task. A Worklist designer defines steps within the scope of a task plan. A step represents a simple set of actions that can be taken to move a task along in its processing. Terminal steps (Complete Step or Abort step) allow a step to be terminated. A step has elements like:

- Assignment Instructions – Optional instructions on how to assign the task (at this step) to a human user who can help complete the task. Instructions include an assignee list (optional, zero or more users/groups that may claim the step), a candidate list handling value that controls how the candidates for claiming the task are prioritized, etc., and a flag for enabling availability checking on users during the assignment process.
- Actions (one or more actions the actor may take on the step)

You can select the steps from the Palette View, drag and drop it into the Task Plan Editor. You can then actions to the step and also set the properties of the step. For more information, see "Adding Steps and Actions to a Task Plan" on page 2-28.

Actions and Connections

Actions represent work a human user does or decisions that are made by a human user in a step in the task instance. Users take actions that are defined for the current step in the task instance. An action definition lives within a step, which in turn lives within a task plan. The various types of actions in Worklist are:

- Work Action
- Assign Action
- Return Action
- Assign to Next User

You can select any of these steps from the Palette View in the Task Plan Perspective, drag and drop the actions within the step in the task plan. For more information, see "Adding Steps and Actions to a Task Plan" on page 2-28.

Work Action

Work Actions represent work the user has done or decisions they have made with respect to this task. A work action causes the transition of a task from one step to another step. The step from which the transition occurs is the step that contains the action. The step or terminal state to which the transition occurs is named or identified within the action itself. Information about the work done or decision taken is captured in properties for the task. Work actions define the data required to complete the action (if any), and a description of what the action means, and under what circumstances the human actor should take the action.

Work actions can specify a number of properties that must be updated or specified when the action is taken. This is the primary mechanism for updating the properties for a task.

Assign Action

Assign actions cause the task on which they are taken to be reassigned according to assignment instructions defined for the action.

Return Action

Return actions cause the task to be "returned" to the Assigned Working state so that they may be claimed by a different user at a later date.

Assign to Next User

AssignToNextUser actions works in conjunction with the "Iterate List" candidate list handling and assigns the task to (and automatically claims it for) the 'next' user in the candidate list. For more information, see "Candidate List Handling" on page 2-14.

Connections

Connections in the Palette View are used to specify the navigation of the steps and actions in the task plan. The connections connect the connector with the first step in the task plan and also the order in which the actions should be done.

Constructors

A task plan defines one or more constructors that are used to initialize new instances of the task plan. A constructor defines:

- A name for the constructor (required, and maximum length of 100 characters).
- A set of property names indicating which properties of the task must be provided on creation of a new instance of this task plan (optional). Each property may have an associated description with respect to this constructor. Properties can be user or editable system properties.
- The name of a step to be made the start step for tasks initialized through this constructor (required).

The properties defined for a constructor follow the same rules for properties listed on a work action. That is, the constructor can reference properties defined on the task plan or the system editable task properties.

Note: A task plan may define constructors that define no required properties. In addition, there can be any number of such constructors.

Candidate List Handling

A task may be explicitly assigned to task or implicitly assigned to a task via a Assignee list. For information about assigning a task from the Worklist Console, see “Assigning a Task to User or Group” in Worklist Administration. For information about assigning a task from the Task Plan Editor, see “Assignment Instructions” on page 2-31.

The assignment instructions contain the following:

- A list of assignees (can be users or groups)
- A flag indicating how the candidate list (resulting from expanding groups in the assignee list to their individual user members and combining with users in the list) will be handled.
- A flag indicating if availability checking should be applied if we are asked to perform load balancing during assignment

The assignee list is evaluated to generate a list of candidate claimants for the task. The candidate list is then evaluated. The possible scenarios of candidate list evaluation are given below. These scenarios are based on the type of candidate list handling chosen. Valid values for candidate list handling are:

- None (no action is taken)
- Default (simple handling)
- Load Balancing (advanced handling)
- Interactive (allows a human to guide the assignment process)
- Iterate List (allows successive assignment to each user on the assignee list)

Table 2-5 Assignee List and Candidate Handling

Assignee List Contents	Candidate List Handling	Result
No assignees given in step	None, Default, Load Balancing, Interactive, or Iterate List	No action is taken.

Table 2-5 Assignee List and Candidate Handling

Assignee List Contents	Candidate List Handling	Result
A single candidate user	None	None
	Default	The task is automatically claimed for the user (and the user becomes the claimant). The claimant can later return the task if needed (at which point the task must be reassigned to choose another user as claimant). No automatic claim action is performed when a task is returned.
	Load Balancing	Same as Default
	Interactive	This flags the Worklist client to use the Worklist API to perform interactive assignment. That is the Worklist User Portal or Custom Task UI (if defined) are used to assign the task to a user.
	Iterate List	Handled the same as Default handling

Table 2-5 Assignee List and Candidate Handling

Assignee List Contents	Candidate List Handling	Result
Multiple candidate users	None	None
	Default	No workload calculation will be performed for the list of candidate users, and the task will not be claimed in the name of any user. All the users on the candidate list will have an equal opportunity or responsibility to claim the task from the list of tasks they see as assigned to them. An exception to this rule is made if the task is currently claimed, and the claimant is among the candidate users for this step. In this case, the current claimant is chosen as the new claimant for the task.
	Load Balancing	A load balancing algorithm is applied to choose a single user from the candidate list to work on the task. The task is automatically claimed for the selected user (and the user becomes the claimant). The claimant can later return the task if needed (at which point another user in the candidate list can claim the task or the task can be reassigned). No automatic load balancing or claim action is performed when a task is returned. If the task is currently claimed, and the claimant is among the candidate users for this step, the current claimant is chosen as the new claimant for the task. In this case, the load balancing algorithm is bypassed.
	Interactive	This flags the Worklist client to use the Worklist API to perform interactive assignment. That is the Worklist User Portal or Custom Task UI (if defined) are used to assign the task to a user.
	Iterate List	Initially claim the task for the first user in the candidate list, and then subsequently (after an AssignToNextUser action is taken) claim the task for the next user in the candidate list.

Load Balancing

When an assignee list yields a list of candidate users that contains more than one candidate, and the candidate list handling type is set to “Load Balancing”, Worklist chooses a user based on that user's workload and, optionally, availability. This behavior is called “load balancing”.

Since load balancing can be somewhat costly (especially when evaluating work load for a large list of candidate users), Worklist allows the task administrator to specifically enable load balancing on a step by step basis. Load balancing may be enabled by specifying the candidate list handling type as `LoadBalancing` on the assignment instructions in the `WorklistTaskAdmin.assignTask()` call, or on any step that specifies assignment instructions.

By default, load balancing does not take into account the user's availability (based on business calendar). However, if this is required, checking can be turned on within the load balancing process. Availability is calculated as a “score” where higher score values indicate greater availability

Steps in Creating a Task Plan

1. Create a task plan folder.
2. Create a task plan.
3. Add one or more steps to the task plan.
4. Define the system properties of each step in the task plan.
5. Add one or more actions for each step in the task plan.
6. Define the system properties for each action in all the steps in the task plan.
7. Define any user properties for the task plan, if required.
8. Use Connectors to specify the sequence in which the steps and actions should be executed.
9. Create a Constructor for the task plan.
10. Validate the task plan.

Before you Begin

Before you start using Worklist, make sure that you:

- Create a domain

- Configure a server
- Start BEA Workshop for WebLogic

Creating a Domain

You have the option to create a domain to:

- Configure a domain to run WLI (business process) and worklist
- Configure a domain to run only for worklist

Configuring a Domain to Run WLI Business Process and Worklist

You may create this domain using the Configuration Wizard. For more information, see Creating a New WebLogic Domain in *Creating WebLogic Domains Using the Configuration Wizard*.

Configuring a Domain to Run Worklist

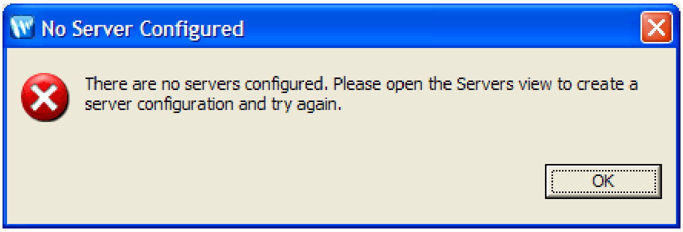
You may create this domain for running worklist application independently, without business process support. To configure the domain perform the following steps:

1. Create a WebLogic Server domain in the Configuration Wizard.
2. From the **Start menu**, click **All Programs** > **BEA** > **Tools** > **Configuration Wizard** to start the BEA WebLogic Configuration Wizard.
3. Select **Extend an existing WebLogic domain** and click **Next**. This displays the Select a WebLogic Domain Directory Source page in the Configuration Wizard dialog box.
4. Select the WebLogic domain and click **Next**.
5. Select **Extend my domain using an existing extension templete** and click **Browse**.
6. Select **workshop_wl_10.2.jar** and click **Ok**.
7. Click **Broswe**, and select the **p13n.jar**.
8. Repeat the above steps and select **wli_worklist.jar.**
9. Accept the default settings in the Customize JDBC and JMS File Store Settings page and click **Next**.
10. In the Extend WebLogic Domain page, click **Extend**.
11. Click **Done**, and the domain is configured.

Configuring a Server

You need to configure a server on which you can deploy the worklist applications.

Note: If a server is not configured, and if you try to start Worklist Console from Workshop for WebLogic Platform, you get the following error message:



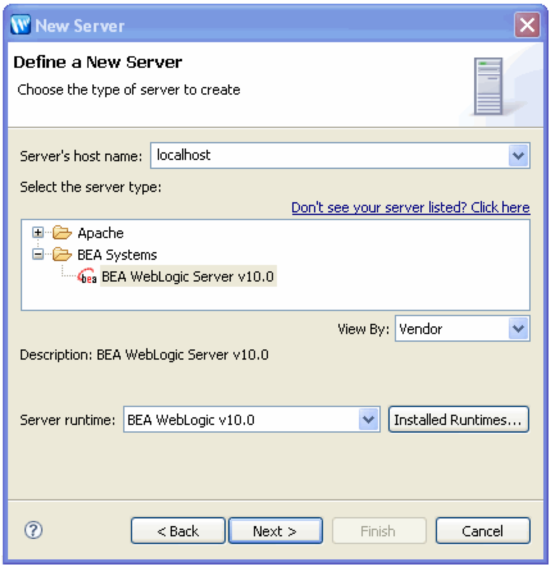
There are no servers configured. Please open the Servers view to create a server configuration and try again.

To configure a server:

1. Select File > New > Other. The Select a Wizard dialog appears.

2. Click the ⊞ icon next to Server and select Server. Click Next. The Define a New Server dialog appears (see Figure 2-2).

Figure 2-2 Define a New Server


New Server
Define a New Server
Choose the type of server to create
Don't see your server listed? Click here
Apache
BEA Systems
BEA WebLogic Server v10.0
Installed Runtimes...
< Back
Next >
Finish
Cancel

Server's host name: localhost

Select the server type: BEA WebLogic Server v10.0

View By: Vendor

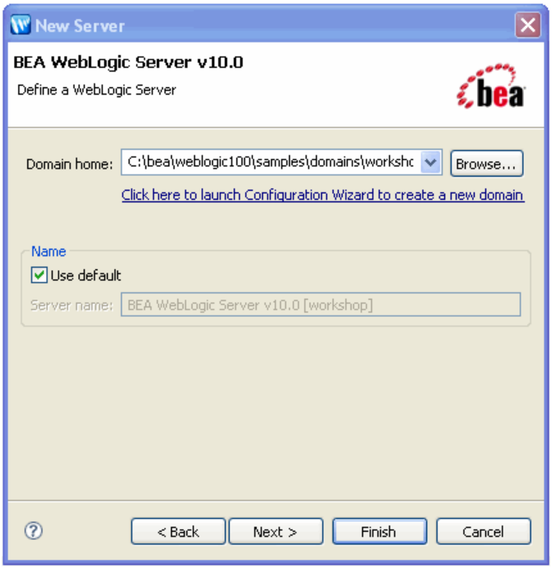
Description: BEA WebLogic Server v10.0

Server runtime: BEA WebLogic v10.0

3. Select the host name of the server and server type. Retain default values for the rest of the fields. Click Next.

   The Define a WebLogic Server dialog appears (see Figure 2-3).

Figure 2-3 Define a WebLogic Server



New Server

BEA WebLogic Server v10.0

Define a WebLogic Server

Domain home: C:\bea\weblogic100\samples\domains\workshc

Click here to launch Configuration Wizard to create a new domain

Name

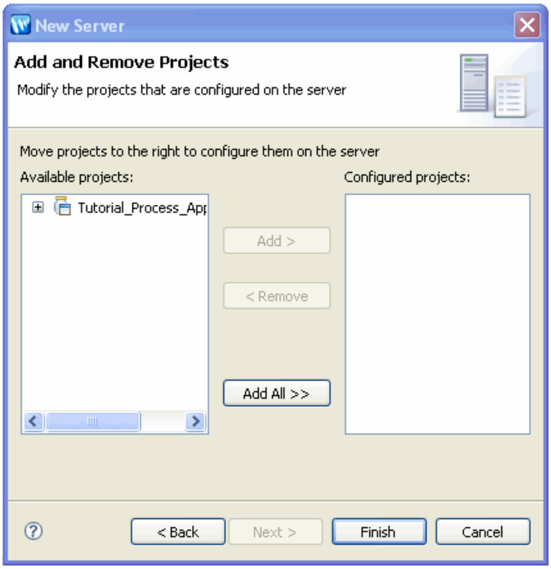
☑ Use default

Server name: BEA WebLogic Server v10.0 [workshop]

4. Enter the path to the domain home. If required, click Browse to locate the domain. Click Next.

   The Add and Remove Projects dialog appears (see Figure 2-4).

Figure 2-4 Add and Remove Projects



New Server

Add and Remove Projects

Modify the projects that are configured on the server

Move projects to the right to configure them on the server

5. Select any projects in the Available Projects list and click Add.

6. Click Finish.

The server is configured.

Configure a domain to run only for worklist

Starting Workshop for WebLogic Platform

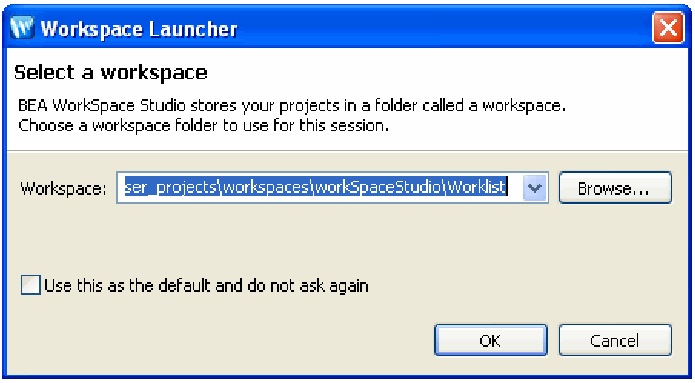
To start the application:

1. Select **Start** > **AllPrograms** > **BEA** > **WorkSpace Studio 1.1**.

   The Workspace Launcher dialog appears (see Figure 2-5).

2. Specify the folder in which all your projects should be stored. This folder is called the Workspace.

Figure 2-5 Workspace Launcher



Select a workspace

BEA WorkSpace Studio stores your projects in a folder called a workspace.
Choose a workspace folder to use for this session.

Workspace: ser_projects\workspaces\workSpaceStudio\Worklist

□ Use this as the default and do not ask again

3. Click **OK**.

   The **BEA WorkSpace Studio** IDE is displayed.

Creating a Worklist Application

A Worklist application consists of an EAR and a Web project, which contain all the files and directories that relate to a single unit of work.

The EAR project corresponds to an Enterprise Application. You can build and deploy this project to create the entire process flow of the enterprise.

The Web project corresponds to the Web application of Worklist that acts as the user interface for the system. The Web project is a part of the EAR project.

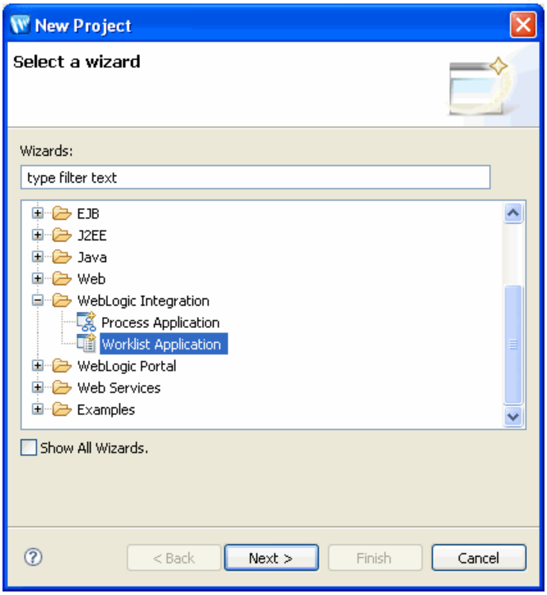
To create a new Worklist application:

1. In BEA WorkSpace Studio, click **File** > **New** > **Project**.

   The **New Project** dialog appears.

2. Click ⊞ next to WebLogic Integration and select **Worklist Application** (see Figure 2-6).

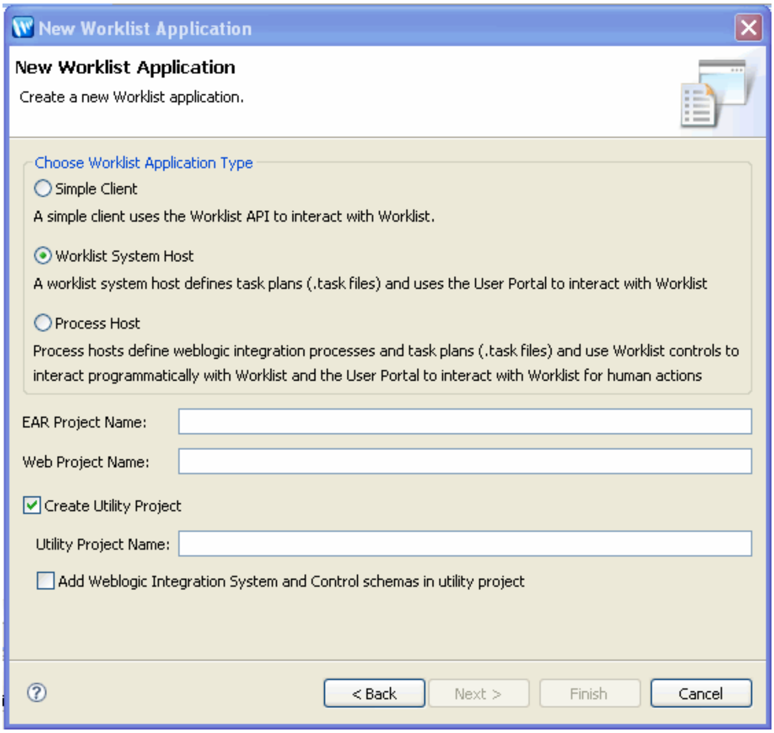
Figure 2-6 New Worklist Project



3. Click **Next**.

   The **New Worklist Application** dialog appears (see Figure 2-7).

Figure 2-7 New Worklist Application Dialog Box



New Worklist Application

New Worklist Application

Create a new Worklist application.

Choose Worklist Application Type

○ Simple Client

A simple client uses the Worklist API to interact with Worklist.

● Worklist System Host

A worklist system host defines task plans (.task files) and uses the User Portal to interact with Worklist

○ Process Host

Process hosts define weblogic integration processes and task plans (.task files) and use Worklist controls to interact programmatically with Worklist and the User Portal to interact with Worklist for human actions

EAR Project Name:

Web Project Name:

☑ Create Utility Project

Utility Project Name:

□ Add Weblogic Integration System and Control schemas in utility project

4. Select the Worklist application type. For more information, see "Worklist Application" on page 2-9.

5. Specify the name of the EAR project.

6. Specify the name of the Web project.

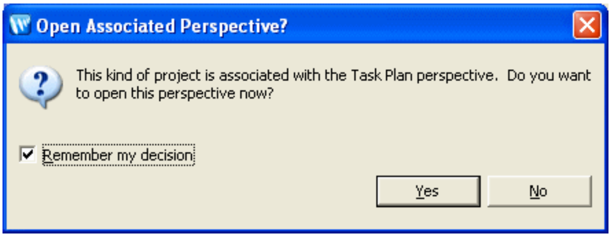
7. If required, select the Create Utility Project check box and specify the Utility project name. This project will contain all the WebLogic Integration schemas.

8. Select the **Add WebLogic Integration System and Control schemas in utility project** check box, this will create the WorklistEvent filter.

9. Click **Finish**.

   If Task Plan perspective is not the current perspective, the Open Associated Perspective dialog appears (see Figure 2-8).

Figure 2-8 Open Associated Perspective? Confirmation Box



Open Associated Perspective?

This kind of project is associated with the Task Plan perspective. Do you want to open this perspective now?

☑ Remember my decision

10. Select the **Remember my decision** check box and click **Yes**.

    The Worklist application is created. In the Package Explorer pane, three project folders (one folder each for the EAR project, Web project and the Utility project) are automatically created and displayed.

The Task Plan perspective is now associated with the Worklist application. The Task Plan Task Plan icon appears on the top right corner of the BEA WorkSpace Studio window.

Creating a Task Plan

To create a task plan

1. In the Package Explorer pane, right-click the `<EAR project folder name>\EarContent` folder, and select **New** > **Folder**.

   The **New Folder** dialog appears.

2. Enter a folder name for the task plan and click **Finish**.

   This task plan folder is created under the `<EAR project folder name>\EarContent` folder.

3. Select the task plan folder and press Ctrl+N.

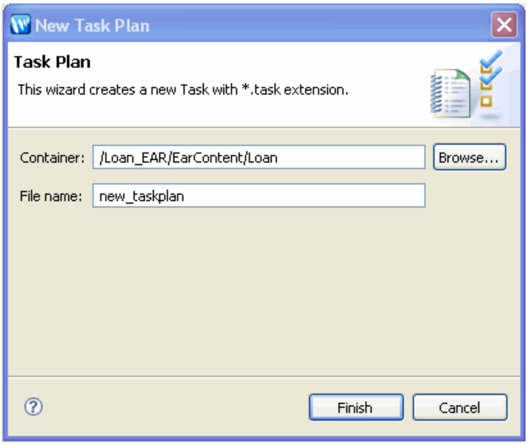
   The New dialog appears.

4. Click ⊞ next to WebLogic Integration and select Task Plan.

5. Click Next.

   The **New Task Plan** dialog appears (see Figure 2-9).

Figure 2-9 New Task File Dialog



Task Plan

This wizard creates a new Task with *.task extension.

Container: /Loan_EAR/EarContent/Loan

File name: new_taskplan

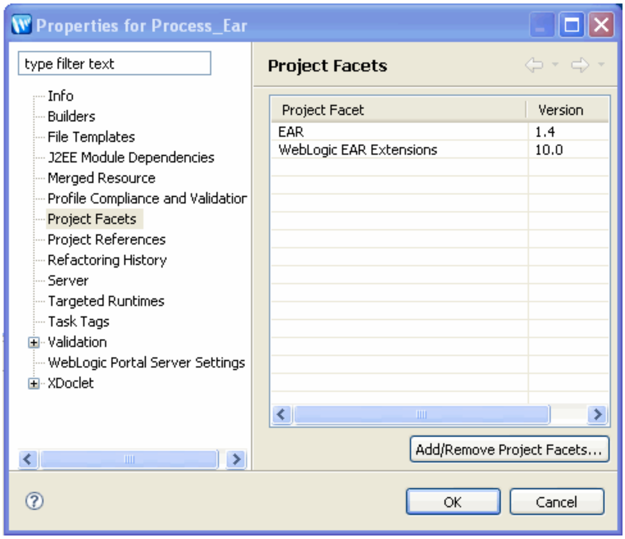
6. Specify the path to the task plan folder in the Container field.
7. Enter the name of the task plan.
8. Click **Finish** to proceed.

The task plan is created and opens in console where you can define the steps and actions for the task. Task plan files (.task files) are placed directly in a WebLogic EAR project under its Content folder. This folder name can vary depending on the project's configuration, but is generally named EARContent.

To create a task plan to a Process Application

1. Create a Process Application.
2. Add Worklist facets to your Process Application by selecting **Project > Properties > Project Facets**, from the **BEA WorkSpace Studio** menu (see Figure 2-10).

Figure 2-10 Project Facets



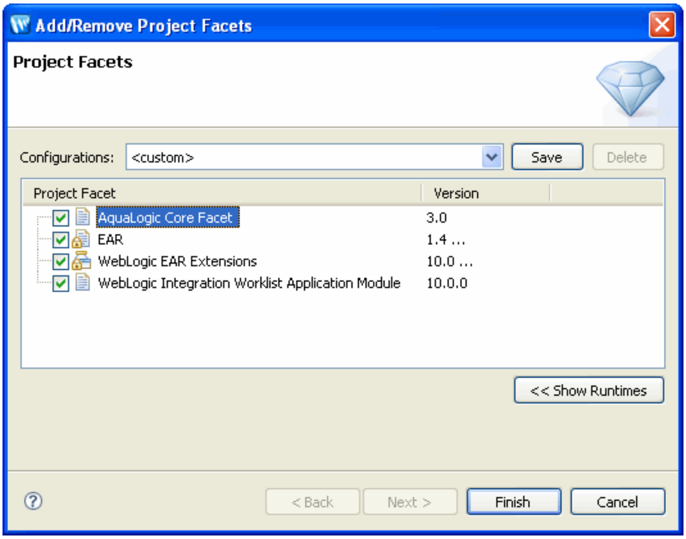
Project Facet	Version
EAR	1.4
WebLogic EAR Extensions	10.0

3. Click **Add/Remove Project Facets**.

   The **Add/Remove Project Facets** dialog appears.

4. Check the **AquaLogic Core Facet** and **WebLologic Integration Worklist Application Module** check box under Project Facet to enable Worklist facet (see Figure 2-11).

Figure 2-11 Add/Remove Project Facets



Configurations: <custom>

Project Facet	Version
☑ AquaLogic Core Facet	3.0
☑ EAR	1.4 ...
☑ WebLogic EAR Extensions	10.0 ...
☑ WebLogic Integration Worklist Application Module	10.0.0

5. Click **Finsh**.

6. Click **Ok**.

   The facets are added to your project.

7. Complete the Steps in Creating a Task Plan.

Adding Steps and Actions to a Task Plan

A task plan is a collection of steps that define the actions a user needs to do when working on a task. To add steps and actions to a task plan:

1. Double-click on the task plan file (`*.task` file) to add the steps.

2. From the Palette View, click Step and then drag and drop it anywhere in the <taskname>.task tab.

   The default name for a new step is **`Step`***#*, where *#* is an incremental numeric value that starts with 1 and changes depending on the number of existing steps in the task plan.

3. Select the step to set its system properties in the Properties View. For more information, see "Setting the System Properties of a Step" on page 2-31.

4. Repeat step 2 and step 3 to define the required number of steps to your task plan.

5. From the Palette View, click Action and then drag and drop it into a step.

   The default name for a new action is **`Action`**#, where # is an incremental numeric value that starts with 1 and changes depending on the number of existing actions in the step.

6. Select the action to set its system properties in the Properties View. For more information, see "Setting the System Properties of an Action" on page 2-34.

   Note: If required, you can create user properties for the task plan and define these user properties for any of the actions. For more information, see "Creating User Properties of the Task Plan" on page 2-35.

7. Repeat step 5 and step 6 to define the required number of actions to your step.

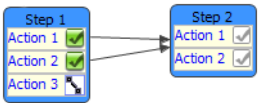
8. Repeat step 5 and step 6 to define all the actions for all the steps in the task plan.

9. Select Abort Step and Complete Step options in the Palette View and place then anywhere in the task plan tab.

10. Click **Connections** in the Palette View and click the source action that you want to connect to the target step or action. Press and hold the Mouse and point to the target step or action. You will see a line that originates from the source action. Release the Mouse cursor at the target step or action. The source and target are not joined by the connecting line.

    After specifying the connection, a green tick appears against the source Action as shown in the following figures. Connecting an action to another step or action specifies the order in which steps and actions in the task plan should be executed (see Figure 2-12).

Figure 2-12 Connecting a Source Action to a Target Action



Note: You can also use self-transition to ensure that after the current action is complete, the actions returns to the step. In Figure 2-12, after Action 3 in Step 1 is completed, control returns to Step1. Double-click an Action to enable self-transition.

11. Repeat step 10 to connect all the actions in the task plan.

12. Save the task plan.

All Steps and actions are added to the task plan. You need to define a constructor for the task plan.

Defining a Constructor for the Task Plan

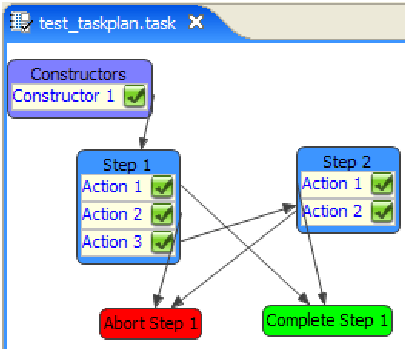
In a task plan, there is at least one constructor that defines how a task instance comes into existence. A constructor for a task plan lists the initial data to be provided for the creation of a task instance as well as the resulting step of the task instance. Each constructor needs to have a step associated with it. There may be more than one constructor for a task plan.

Note: Constructors can be invoked by authorized users or system actors, so that the task instances can be created either by human data entry or system execution in a process.

To define a constructor:

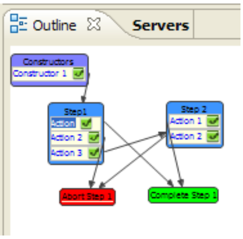
1. Click **Constructor** on the Palette tab and drop it in the Constructor container of the task plan tab.
2. Specify the system properties for the constructor. For more information, see "Setting the System Properties of an Action" on page 2-34.
3. If required, specify any user properties defined for the task plan. For more information, see "Creating User Properties of the Task Plan" on page 2-35.
4. Click **Connection** in the Palette View to connect the Constructor to the required step in the task plan (see Figure 2-13).

Figure 2-13 Task Plan



You can also view the task plan in the Outline View (see Figure 2-14).

Figure 2-14 Task Plan in Outline View

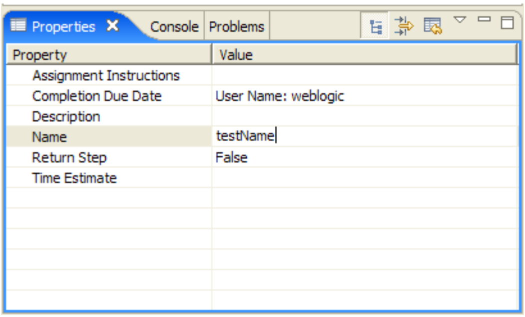


Setting the System Properties of a Step

You can set any of the following properties for a step:

- User or group to whom the step should be assigned
- Date on which the step should be completed
- Description associated with the step
- Name of the step
- Return Step can be set to True or False. If True, the Return action causes the task to be *returned*, thus returning it to the Assigned Working state so that it may be claimed by a different user at a later date.
- Estimated time taken to complete the step in the task.

Figure 2-15 System Properties of a Step



Property	Value
Assignment Instructions	
Completion Due Date	User Name: weblogic
Description	
Name	testName
Return Step	False
Time Estimate	

Assignment Instructions

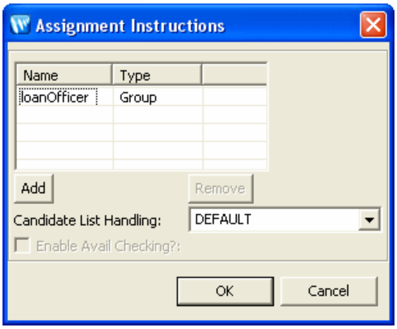
Steps define assignment instructions to control who can claim instances of its parent task plan, when those instances are on this step. At any given time, a task instance may be claimed by at most one human actor. Thus, the assignment instructions for the current step (if present) effectively become the assignment instructions for the whole task instance.

The actor that claims a task instance is called the 'claimant' for the task instance. This actor may be designated by name, by group or by role. A human actor must be either one of the named actors, or belong to one of the named groups, or be granted the named role, in order to claim the step.

To set the system properties of a step, do any of the following:

1. To assign the current step to any existing user or group, select Assignment Instructions and click ... in the Value column. The Assignment Instructions dialog appears.
   a. Click Add and enter the user or group name in the Name column.
   b. In the Type column, select User or Group.
   c. In the Candidate List Handling field, select the type of handling to be applied while assigning the step to the specified users or groups. For more information about candidate list handling, see "Creating a Task Plan" on page 2-25.
   d. Select or clear the Enable Avail Checking check box only when you select Load_Balancing as the Candidate List Handling method for the step.
   e. Click OK. The specified user or group is assigned to the current step in the task based on the candidates associated with the step will be handled.

Figure 2-16 Assignment Instructions



Name	Type	
loanOfficer	Group	

Candidate List Handling: DEFAULT

☐ Enable Avail Checking?:

2. Select Completion Due Date and click .... The Completion Due Date dialog appears.
   a. Enter the number of days, hours, and minutes required to complete the current step in the task.

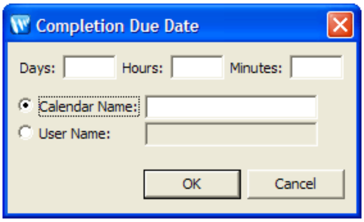
b. Specify a calendar to associate a business calendar with the due date. The number of days required to complete the days is calculated based on the number of free and busy days in the calendar.

c. Select a user name to use the calendar associated with user to calculate the due date.

d. Click OK.

The interval may define a number of days, hours and/or minutes in the D days H hours M minutes format, where D is number of days, H is number of hours, and M is number of minutes. The completion due date of the task is generated based on the interval, and the user or business calendar.

Figure 2-17 Completion Due Date for the Step

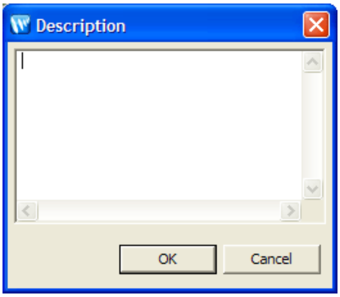


3. Select Description and click …. The Description dialog appears (see Figure 2-18).

   a. Enter a brief description for the step.

   b. Click OK.

Figure 2-18 Step Description

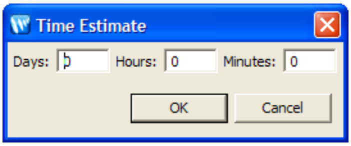


4. Select Name and enter a meaningful name for the step.

5. Select Return Step and select True or False. If True, the task is *returned*, thus returning it to the Assigned Working state so that it may be claimed by a different user at a later date.

6. Select Time Estimate and click . The Time Estimate dialog appears.
   a. Estimate the days, hours and minutes required to complete the step.
   b. Click OK.

Figure 2-19 Time Estimated for the Step



Days: 5 Hours: 0 Minutes: 0

The system properties are now set. If required, click the icon to restore the default system properties for the step. You can restore to the default values only if you did not save the task plan.

Setting the System Properties of an Action

You can set the following system properties for any action:

Table 2-6 System Properties of an Action

Property Name	Description
Name	Name of the action.
Description	A brief description of the action.
Next Step	The next step that should be done after completing the current action.
Current Step Claim Due Date	The date by which the current step should be claimed by any user.
Current Step Completion Due Date	The date by which the current step should be completed.
Is Owner Group	When set to true implies that the value specified in the Owner property refers to a group.
Owner	The owner of the task.
Priority	The priority of the task. This is an integer value that, if specified, conditionally enables the use of user availability information when performing load balancing during task assignment. Availability checking is performed only for those task instances that have a Priority value greater than or equal to the given threshold value.

Table 2-6 System Properties of an Action

Property Name	Description
Step Time Estimates	The estimated time (specified as a business calendar interval, for example, 3min2hour1day or 10 years 5 hours 3 seconds) required to complete the current step in the task. This information is used by the default Worklist assignment handler for load balancing. If specified, this value is used as a default value for the time estimate setting on the current step in this task.
Task Completion Due Date	The business date and time by which this task must be completed. This is specified as a time interval, and an optional business calendar name. If there is no business calendar name given, the Worklist global system calendar will be used in the date calculation. The business calendar is used to calculate the absolute date by which the task must be completed. The interval may define a number of days, hours and/or minutes in the D days H hours M minutes format, where D is number of days, H is number of hours, and M is number of minutes. The completion due date of the task is generated based on the interval, and the user or business calendar. For example, if the interval is specified as 10 days and the business calendar is selected. If current date is 1, and the next 5 days are busy based on selected business calendar, then the completed due date for the task would be 15.
Task Name	Name of the task.
Task Time Estimate	The estimated time (specified as a business calendar interval, for example, 3min2hour1day or 10 years 5 hours 3 seconds) required to complete the task. This information is used by the default Worklist assignment handler for load balancing. If specified, this value is used as a default value for the time estimate setting on all the steps this task contains.

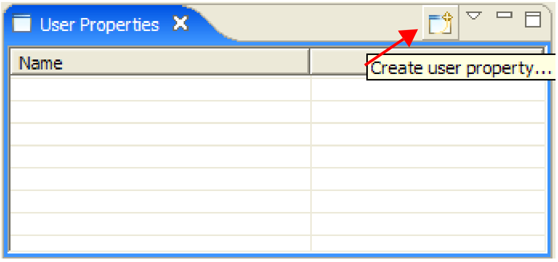
Creating User Properties of the Task Plan

User properties are business-specific data elements of a task plan. User properties are global and apply to all the steps throughout the life cycle of the task plan. In WLI 10.2, Enumeration (List) Data type user properties is available to task properties.

To create user properties:

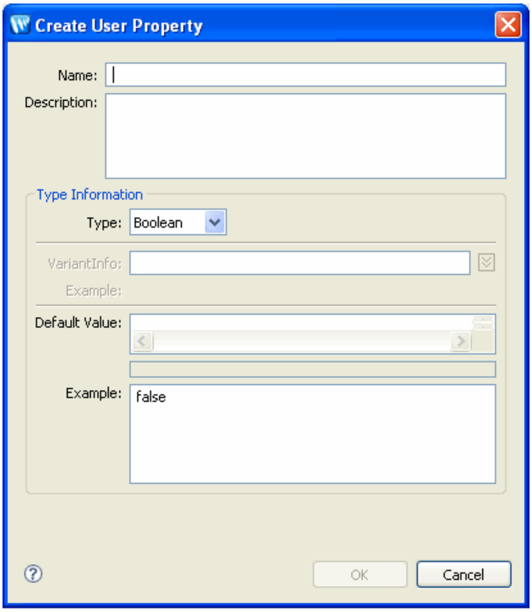
1. From the User Properties View, click the **Create user property** icon (see Figure 2-20).

Figure 2-20 User Properties Tab



The Create User Property dialog appears (see Figure 2-21).

Figure 2-21 Create User Properties for Task Plan



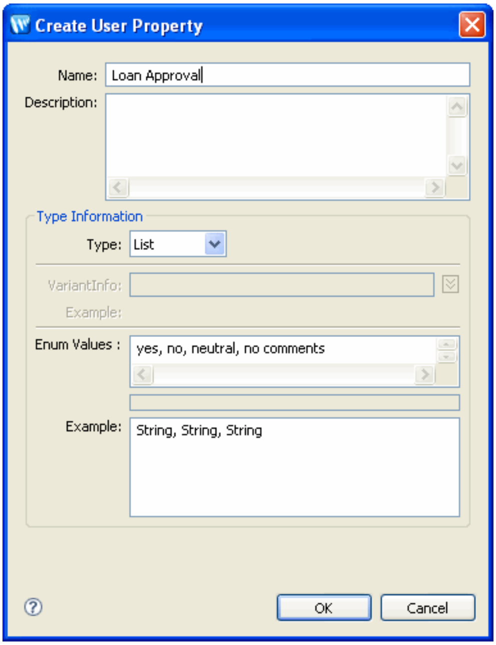
2. Enter the name and description of the user property.

3. Select the data type of the user property from the Type drop-down list.

   The data type can be `Boolean, DateTime, Float, Integer, JavaBean, String, URL, List,` or `XMLBean`. If you select `DateTime, JavaBean, or XMLBean` in the `VariantInfo` field, enter the format in which the property should be specified. For example, `mm/dd/yyyy ‘at’ hh:mm:ss` can be the date and time format.

For creating List data type, the information entered in the Enum Values field should be separated by a (,) (see Figure 2-22). For example: yes, no, neutral, no comments. Once the user property of this type is created , the default value edited in the Properties view for List data type will have similar behavior of other data types.

Figure 2-22 Create User Property (List Data Type)



Name: Loan Approval

Description:

Type: List

Enum Values : yes, no, neutral, no comments

Example: String, String, String

4. If required, enter the default value of the user property.
5. Click **OK**.

The user property is created.

Validating a Task Plan

The final stage in designing and deploying the task plan is to validate if the task plan is working according to the required enterprise model specification.

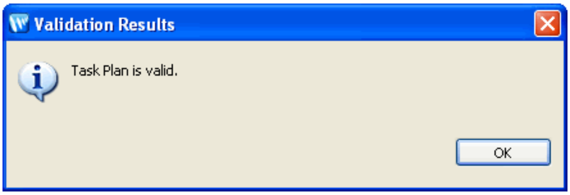
Note: For the task plan to be deployed successfully, version of the task plan should be 1.0. Verify that the task plan version in the Properties View is 1.0.

To validate the task plan:

1. Click Worklist > Validate Task Plan for Runtime.
2. If the task plan is valid, then the Validation Results dialog appears as shown in the following figure.

   If the task plan is invalid, the error details are displayed in the Validation Results dialog (see Figure 2-23). Fix the errors and re-validate the task plan.

Figure 2-23 Validate Task Plan



Task Plan is valid.

3. Click OK to confirm.
4. Select File > Save All to save the application.

If required, you can modify the layout of the task plan by selecting Worklist → Automatic Layout.

Importing a Task Plan

To import an existing task plan into the current workspace:

1. In BEA WorkSpace Studio, select **File > Import**.

   The Import dialog appears.
2. Under Select an import source, expand General and select **Existing Projects into Workspace** as the import source and click **Next**.

   The **Import Projects** dialog appears.
3. Specify the root directory in which the project is located.
4. Select the projects that you want to import.
5. Click **Finish**.

The task plan is imported into the current workspace.

Note: In the Import dialog, you can also select File System as import source. The File System dialog appears. Specify the path to the task plan, select the projects that you want to import, specify the parent folder into which you want to import the file system. Click Finish to import the task plan.

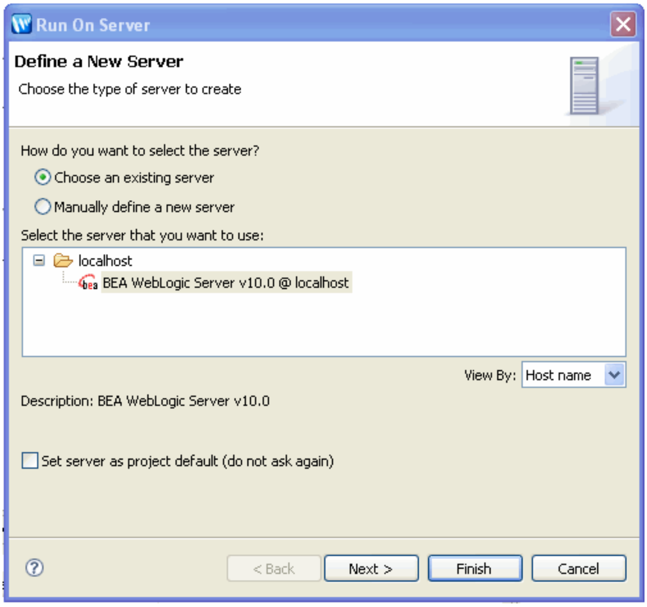
Deploying a Task Plan

Once the task plan is modeled completely, you can deploy it on WebLogic Integration Server.

To deploy the task plan:

1. In the Package Explorer View, right-click on the <web project name> folder and select Run As → Run on Server. The Run on Server dialog appears.
2. Specify the server on which you want to deploy the task plan. You can choose an existing server or manually define a new server. For more information about defining a server, see "Configuring a Server" on page 2-19.

Figure 2-24 Run on Server



Run On Server

Define a New Server

Choose the type of server to create

How do you want to select the server?

● Choose an existing server

○ Manually define a new server

Select the server that you want to use:

- localhost
  - BEA WebLogic Server v10.0 @ localhost

View By: Host name

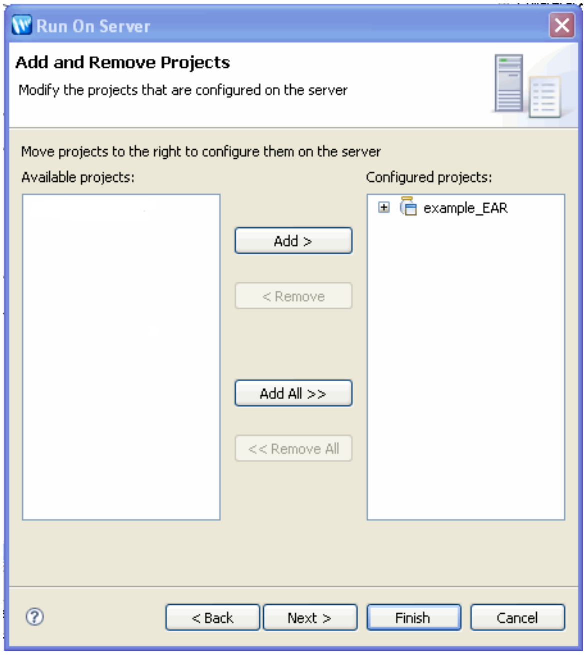
Description: BEA WebLogic Server v10.0

□ Set server as project default (do not ask again)

3. To choose an existing server, from the Select the server that you want to use: list, select the server on which you want to deploy the task plan and click Next. The Add and Remove Projects dialog appears.

4. Select the project in the Available projects list and click Add. The selected project is listed in the Configured projects list.

Figure 2-25 Add or Remove Projects Configured on the Server



5. Click Next. The Select Tasks dialog appears.

6. Click Finish.

The selected project is deployed on the specified server.

It will take some time to deploy the project on the server. After the task plan is deployed successfully, it opens up on the Worklist User Portal within the Workshop for WebLogic environment.

Changing a Task Plan

You can change a task plan as and when required. Make sure that you re-deploy the task plan after you change the task plan.

To change a task plan:

1. From Package Explorer, navigate to the location of the task plan.
2. Right-click on the task plan and select Open With > Task Plan Editor.
3. Make the required changes and save the task plan. For more information, see "Creating a Task Plan" on page 2-25.

The task plan is changed.

Note: After you update the task plan, re-deploy the task plan. For more information, see "Deploying a Task Plan" on page 2-39. The updated changes are only reflected in task instances that you created after you re-deployed the task plan.

Using and Customizing User Portal

Worklist provides a web-based user portal for interacting with tasks. This portal allows users to see the details of a task and take actions on the task to move it through its lifecycle. Each step in a task plan can define unique actions and thus present its own unique considerations to the user.

About Worklist User Portal

Worklist User Portal is an all-purpose user interface that uses the metadata for steps and actions to render an interactive interface at each step. The user interface for a given step and action will differ from the user interface for other steps and actions by virtue of the differences in the metadata for those steps and actions.

Worklist User Portal is a part of the Worklist domain and provides an interface for Worklist users to manage task instances that they are authorized to deal with. This portal provides basic out-of-the-box functionality for creating, monitoring, and updating tasks.

Worklist User Portal can be used by two types of users: an administrator kind of user and a normal user. The functions that these users can do differs based on permissions.

Customizing Worklist User Portal

Worklist User Portal is an all-purpose user interface that uses the metadata for steps and actions to render a usable interface at each step. The user interface for a given step and action will differ from the user interface for other steps and actions by virtue of the differences in the metadata for those steps and actions. For example, when rendering the user interface at a given step, the Worklist User Portal shows the description and actions the task plan designer defined for that step. Worklist User Portal has only this information to work on when rendering the user interface, and cannot account for any more subtle needs a user may have with regard to this step (For example, summary information that would be needed to enter a value for an action's required properties, or calculations that may be applied to individual task properties to derive the result value for a property)

In some cases, like the previous examples, a given step or the actions that can be taken on that step may require a customized user interface. Worklist accounts for this need for extensibility by allowing a web developer to design a custom web user interface to be used for a given step, or an entire task plan. This custom web user interface is called a Custom Task User Interface. Custom Task UI allows a web developer to design custom pages only where needed, and still use the facilities of the default Worklist User Portal everywhere else. This allows a gradual investment in the customization of the user portal, instead of requiring a wholesale replacement of the Worklist User Portal interface.

A custom interface can be designed for just a step in a task plan or for the entire task plan.

Login to Worklist

To login to Worklist User Portal:

1. Open a Web browser and enter the following URL:

   ```
   http://localhost:7001/<web project name>/
   ```

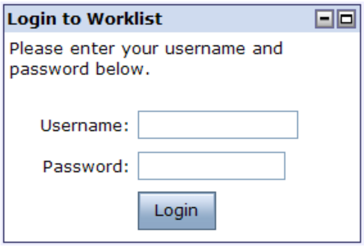
   The Web project corresponds to the Web application of Worklist that acts as the user interface for the system. For more information, see “Worklist Application” on page 2-9. You can use any external browser, for example Internet Explorer, or the default browser that comes with Workshop for WebLogic Platform.

   Worklist User Portal is displayed.

   Note: E-mail notifications regarding tasks associated with you contain the URL to the specific task instances. You can also click on this URL to access the task in the Worklist User Portal.

2. In the Login to Worklist portlet, enter your user name and password.

Figure 3-1 Login to Worklist User Portal

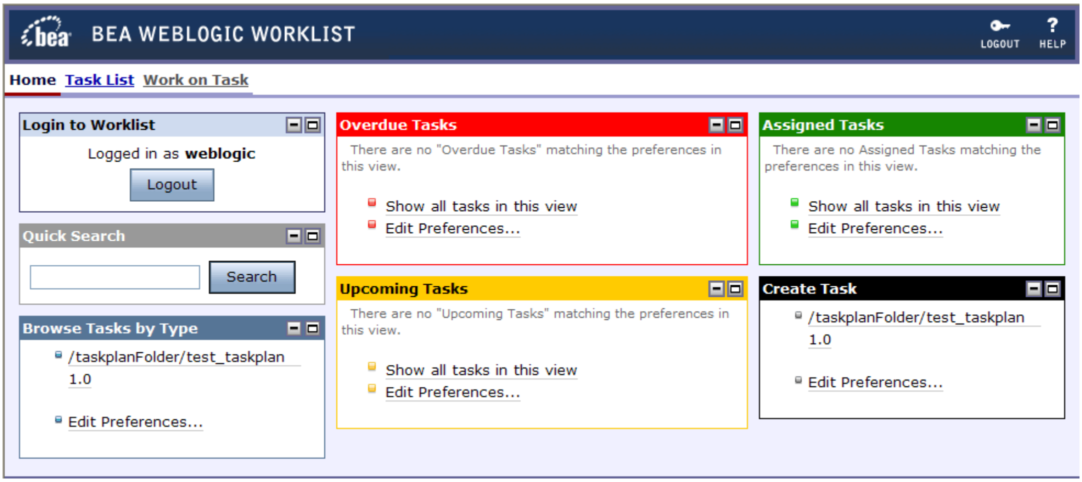


Please enter your username and password below.

3. Click Login.

Assigned, upcoming, overdue tasks associated with you are displayed in the appropriate portlet in the User Portal home page. Task plans based on which you can create task instances are displayed in the Create task portlet.

Figure 3-2 Worklist User Portal Home Page



Logged in as **weblogic**

There are no "Overdue Tasks" matching the preferences in this view.

There are no Assigned Tasks matching the preferences in this view.

There are no "Upcoming Tasks" matching the preferences in this view.

Note: After you are done, click Logout to exit from Worklist User Portal.

Additional Information

For more information about User Portal and customizing the User Portal, see Using Worklist User Portal.

The following table summarizes the tasks associated with using and customizing the user portal.

Modules	Associated Tasks
User Portal	For information about each of the following, see Worklist User Portal in *Using Worklist User Portal*: • Portlets in Worklist User Portal • Creating a Task Instance • Creating a Task Instance Associated with an Interactive Assignee List • Viewing Upcoming Tasks • Viewing Assigned Tasks • Claiming a Task Instance • Working on a Task Instance • Viewing the Task Instance Details • Searching for a Task • Browsing Tasks based on Task Plan • Viewing Overdue Tasks • Task List Page • Deleting a Task Instance
Custom User Portal	For information about each of the following, see Customizing Worklist User Portal in *Using Worklist User Portal*: • Typical Workflow • Implementing the Custom Task UI Page Flow • Creating a Page Flow • Creating Form Beans • Defining Actions on a Page Flow • Defining JSP Pages • Registering the Custom Task UI • Deploying the Custom Task UI

Using Worklist Controls

Java Controls are server-side components managed by the Workshop framework. Controls expose Java interfaces that can be invoked directly from business processes. In other words, controls are the interfaces between your business processes and other resources.

This section describes the built-in controls provided for Worklist that support the integration of business users with business processes. It includes the following topics:

- About Worklist Controls
- Creating Worklist Controls
- Using Task and Task Batch Controls in Business Processes

About Worklist Controls

Worklist provides two Java controls: `TaskControl` and `TaskBatchControl`. You can use Business Process and these Worklist controls to support the integration of human actors and automated actors. Automated actors interact with the JPD and human actors interact with the Worklist User Portal.

As with other built-in controls in Workshop for WebLogic Platform, you use the controls by adding instances of the controls to your business process. Subsequently, you invoke operations on the controls at the point in the business process at which you want to integrate the business-user logic.

The underlying control implementation takes care of most of the details of the interaction for you. Business processes invoke operations on the controls using `Control Send` and `Control Send` with Return nodes. Business processes can block at Control Receive nodes waiting for events to be returned from controls. In other words, Control Receive nodes are triggered by control callbacks. Business processes can use a TaskControl or TaskBatchControl to change a task and also wait for a task (via a TaskControl) to be changed by a human actor. You can extend Worklist controls through Java annotations. Common extensions include implementing callback functions and performing system queries.

Worklist provides the following controls:

- TaskBaseControl—provides the built-in control methods and annotations that are common to both TaskControl and TaskBatchControl.

  Note: This control is a base class for the other Worklist controls and is never used directly by Worklist clients.

- TaskControl—simplifies working with a single task instance via the low-level Worklist API. This control extends TaskBaseControl and thus supports all the methods and annotations supported by TaskBaseControl.
- TaskBatchControl—simplifies working with a batch or group of task instances via the low-level Worklist API. This control extends TaskBaseControl and thus supports all the methods and annotations supported by TaskBaseControl.

These controls provide simplified construction and initialization of Worklist API resources. These controls are extensible controls that allow you to define extension interfaces containing new methods beyond those on the generic control interface. The new methods of these controls can be annotated to attach Worklist behaviors to them.

TaskBaseControl

TaskControl and TaskBatchControl extend from TaskBaseControl, which provides the built-in control methods and annotations that are common to both TaskControl and TaskBatchControl. It provides the following built-in methods:

- String createTask(TaskCreationXMLDocument doc)
- TaskType getTaskType(TaskType.ID taskTypeId)

TaskBaseControl does not allow any control-level annotations. However, it provides the following method-level annotations, which allow you to assign one or more Worklist behaviors to the methods on a control extension:

- create

- assign
- take an action
- get properties (built-in or user-defined)
- set properties (built-in or user-defined)
- set error
- claim
- return
- suspend
- resume
- complete
- abort
- delete

Method-level Annotations for TaskBaseControl

The various annotations are:

Annotation Name	Description	Fields that Support Parameters
TaskCreate Annotation	Causes the method attached to this annotation to create a new task using the configuration values in this annotation. This annotation causes the `setTaskId()` method to be called if the control is a TaskControl extension (giving the ID of the newly created task).	name, comment, description, priority, owner, timeEstimate
TaskPlanID Annotation	The ID of the application hosting the Worklist system instance whose registry contains the task plan for the new task. If not specified, this will default to the name of the application that contains this control instance.	path, version, worklistHostApplicationId
TaskUpdate81x Annotation	Backward compatibility for 81x use only. Causes the method attached to this annotation to update a task using the configuration values in this annotation.	

Annotation Name	Description	Fields that Support Parameters
ContainerHandleAnno Annotation		type, subtype, instance
DateTimeSpec Annotation	Absolute date/time in the default date/time format for the current locale (will be parsed with `DateFormat.getDateTimeInstance().parse().` Mutually exclusive with businessTime. If businessTime is specified, this is ignored. **Note:** This field supports parameters so you can specify a method parameter that holds a pre-parsed or calculated `java.util.Date` or `java.util.Calendar` instance.	absoluteTime
BusinessTimeAnno Annotation	Represents an abstract business date and time by defining a time duration from a (as yet unknown) reference time. This duration may optionally be calculated using a business calendar thus allowing a discontinuous set of time segments to be calculated whose total duration matches the given duration. The end time of the last such segment is taken as the effective business date/time for this BusinessTime. The time segments are placed such that they fall within the boundaries of 'available' time on the business calendar.	duration, calendarName
StepTimeEstimate Annotation	Estimate in `Interval` string format, such as `3d4h5m`	stepName, estimate
AssignmentInstructionsAnno Annotation	Describes instructions for assigning a task to human actors.	
AssignmentInstructions81x Annotation	Describes instructions for assigning a task to human actors in 8.1.x terms.	
AssigneeDefinitionAnno Annotation	Used from within `AssignmentInstructionsAnno` to define the list of assignees.	name

Annotation Name	Description	Fields that Support Parameters
PropertyValue Annotation	Provides a single value for a single property with a given name.	name, value
TaskAssign Annotation	Causes the method for this annotation to stimulate the assignment of one or more task instances given the defined assignment instructions	
TaskAssign81x Annotation	Backward compatibility for 81x only. Causes the method for this annotation to stimulate the assignment of the task instances given the defined assignment instructions.	
TaskStart81x Annotation	Backward compatibility for 81x only. Causes the method for this annotation to take the `Start` action on the CLAIMED step of a task based on the Compatibility 8.1.x task plan.	
TaskStop81x Annotation	Backward compatibility for 81x only. Causes the method for this annotation to take the `Stop` action on the STARTED step of a task based on the Compatibility 8.1.x task plan.	
TaskTakeAction Annotation	Causes the method for this annotation to take the given action on the current step of this task instance. The named action may be a Work action, an Assign action, or a Return action.	name

Annotation Name	Description	Fields that Support Parameters
TaskGetData Annotation	Causes the method for this annotation to retrieve the `TaskData` from one or more task instances for this control. The method attached to this annotation will return a `TaskData` object. Do not use this annotation in conjunction with other annotations that return values, as annotations will compete and collide in returning values; only one return value can be provided from the method. If the return type of the method attached to this annotation is not assignment compatible with `TaskData` a `ClassCastException` will be thrown at runtime. If this control manages more than one task instance, the method attached to this annotation will return an array of type `TaskData[]`.	
TaskGetInfo Annotation	Backward compatibility for 81x only. Causes the method for this annotation to retrieve the `TaskInfo` from one or more task instances for this control. The method attached to this annotation will return a `TaskInfo` object. Do not use this annotation in conjunction with other annotations that return values, as annotations will compete and collide in returning values; only one return value can be provided from the method. If the return type of the method attached to this annotation is not assignment compatible with `TaskInfo` a `ClassCastException` will be thrown at runtime. If this control manages more than one task instance, the method attached to this annotation will return an array of type `TaskInfo[]`.	

Annotation Name	Description	Fields that Support Parameters
TaskGetPropertyNames81x Annotation	Backward compatibility for 81x use only. Causes the method for this annotation to retrieve the names of all properties set on the task instances for this control. The method attached to this annotation will return an array of String containing names of properties. Do not use this annotation in conjunction with other annotations that return values, as annotations will compete and collide in returning values; only one return value can be provided from the method. If the return type of the method attached to this annotation is not assignment compatible with String[] a `ClassCastException` will be thrown at runtime. If this control manages more than one task instance, the method attached to this annotation will return a two-dimensional array where the first dimension indicates the task ID (as a String) of each individual task instance. The second dimension represents the property names array described above.	

Annotation Name	Description	Fields that Support Parameters
TaskGetProperties Annotation	Causes the method for this annotation to retrieve the value of the named properties from the task instance(s) for this control. The method attached to this annotation will return an array of `PropertyInstance` objects, one per property specified. If the property was not found, the array element corresponding to the unknown property will be null. In the special case where only one property name is given, and if the return type of the method is not an array type, the single property value will be returned (not an array) according to the rules below. This annotation should not be used in conjunction with other annotations that return values, as only one return value can be provided from the method (and annotations will compete and collide in returning values). If the return type of the method attached to this annotation is not assignment compatible with `PropertyInstance`, this annotation will cause the value object of the `PropertyInstance` to be extracted. The value object (if assignment compatible) will be stored in the return value. If not, a `ClassCastException` will be thrown at runtime. If this control manages more than one task instance, the method attached to this annotation will return an array of `TaskProperties` objects indicating the task ID (as a String) of each individual task instance. The `TaskProperties` object represents the property value array described above (for a single task). **Note:** For tasks based on the Compatibility 8.1.x task plan, the names given in `propertyNames` will not be represented in the task plan. This is allowed for compatibility reasons only with this task plan. All properties are assumed to be of type String.	propertyNames, sysProperties

Annotation Name	Description	Fields that Support Parameters
TaskGetRequest81x Annotation	Backward compatibility for 81x use only. Causes the method for this annotation to retrieve the value of the request property from the task instance(s) for this control. The method attached to this annotation will return an object representing the given property. The type of that object will be (in precedence order): • A `TaskMessage` if the method return type is assign-compatible with `TaskMessage` • A `byte[]` containing the byte data of the property if the return type is `byte[]` • An `XMLBean` if the return type is assign-compatible with `XmlObject` • A Java object Do not use this annotation in conjunction with other annotations that return values, as annotations will compete and collide in returning values; only one return value can be provided from the method. If this control manages more than one task instance, the method attached to this annotation will return a two-dimensional array where the first dimension indicates the task ID (as a String) of each individual task instance. The second dimension represents the indicated property described above.	

Annotation Name	Description	Fields that Support Parameters
TaskGetResponse81x Annotation	Backward compatibility for 81x use only. Causes the method for this annotation to retrieve the value of the response property from the task instance(s) for this control. The method attached to this annotation will return an object representing the given property. The type of that object will be (in precedence order): • A `TaskMessage` if the method return type is assign-compatible with TaskMessage • A `byte[]` containing the byte data of the property if the return type is `byte[]` • An `XMLBean` if the return type is assign-compatible with `XmlObject` • A Java object Do not use this annotation in conjunction with other annotations that return values, as annotations will compete and collide in returning values; only one return value can be provided from the method. If this control manages more than one task instance, the method attached to this annotation will return a two-dimensional array where the first dimension indicates the task ID (as a String) of each individual task instance. The second dimension represents the indicated property described above.	

Annotation Name	Description	Fields that Support Parameters
TaskSetProperties Annotation	Causes the method for this annotation to set the value of the given properties onto the task instance(s) for this control. The method attached to this annotation will take arguments that can be combined with annotation values to form the effective runtime property value. If the effective runtime value is assignment compatible with the value class of the data type for the property, it will be taken as the new property value. If not, and it is of type String, it will be treated as a serialized default value for the data type, and de-serialized to get the actual runtime value for the property. If this fails, or the original runtime value was not a String, then a `DataTypeException` is thrown at runtime. If the method declares it can throw `DataTypeException`, the exception will be thrown directly. Otherwise, it will be wrapped in an `IllegalArgumentException` and thrown that way.	
TaskSetProperty81x Annotation	Backward compatibility for 81x use only. Causes the method for this annotation to set the value of the given property onto the task instance(s) for this control. The method attached to this annotation will take arguments that can be combined with annotation values to form the effective runtime property value. **Note:** For tasks based on the Compatibility 8.1.x task plan, the name given in name will not be represented in the task plan. This is allowed for compatibility reasons only with this task plan. All properties are assumed to be of type `String`.	

Annotation Name	Description	Fields that Support Parameters
TaskSetRequest81x Annotation	For 81x backwards compatibility only. Causes the method for this annotation to set the value of the indicated property onto the task instance(s) for this control. The method attached to this annotation will take arguments that can be combined with annotation values to form the effective runtime property value. The effective runtime value will be taken as the `data` for the `TaskMessageValue` value for the indicated property. If the method declares it can throw `DataTypeException`, any `DataTypeException` that might occur when trying to serialize or store the value will be thrown directly. Otherwise, it will be wrapped in an `IllegalArgumentException` and thrown that way.	
TaskSetResponse81x Annotation	For 81x backwards compatibility only. Causes the method for this annotation to set the value of the indicated property onto the task instance(s) for this control. The method attached to this annotation will take arguments that can be combined with annotation values to form the effective runtime property value. The effective runtime value will be taken as the `data` for the `TaskMessageValue` value for the indicated property. If the method declares it can throw `DataTypeException`, any `DataTypeException` that might occur when trying to serialize or store the value will be thrown directly. Otherwise, it will be wrapped in an `IllegalArgumentException` and thrown that way.	
TaskRemoveProperties81x Annotation	For 8.1/8.5 backwards compatibility only. Causes the method for this annotation to unset the named properties from one or more task instances for this control.	

Annotation Name	Description	Fields that Support Parameters
TaskSetError Annotation	Causes the method for this annotation to set an error condition on one or more task instances for this control. Methods with this annotation should only be used to indicate technical errors in the processing of a task. Business errors should be accounted for in the step and action model of the task type for this task.	message
TaskClaim Annotation	Causes the method for this annotation to claim one or more task instances for this control.	claimant
TaskClaim81x Annotation	For 8.1/8.5 backwards compatibility only. Causes the method for this annotation to claim one or more task instances for this control, and then take the Claim action on the ASSIGNED step of the task. This has been deprecated. Use custom task plan and `TaskTakeAction` annotation instead.	
TaskReturn Annotation	Causes the method for this annotation to return one or more task instances for this control.	
TaskReturn81x Annotation	For 8.1 and 8.5 backwards compatibility only. Causes the method for this annotation to return one or more task instances for this control, and then take the Return action on the CLAIMED or STARTED step of the task. This has been deprecated. Use custom task plan and `TaskTakeAction` annotation instead.	
TaskSuspend Annotation	Causes the method for this annotation to suspend one or more task instances for this control.	
TaskResume Annotation	Causes the method for this annotation to resume one or more task instances for this control.	
TaskComplete Annotation	Causes the method for this annotation to complete one or more task instances for this control.	

Annotation Name	Description	Fields that Support Parameters
TaskAbort Annotation	Causes the method for this annotation to abort one or more task instances for this control.	
TaskDelete Annotation	Causes the method for this annotation to delete one or more task instances for this control.	

Task Control

`TaskControl` encapsulates low-level access to a single task instance throughout its life cycle. It provides methods to set user-defined and system properties of the task, as well as taking actions on a task.

`TaskControl` provides programmatic and declarative event services as described in the Beehive Controls Development Guide. Control extensions of the `TaskControl` can define their own event methods, and attach listener configuration attributes to them via annotations.

TaskControl extends `TaskBaseControl` (thus providing all its methods). `TaskControl` also provides methods for the following:

- `setTaskId(String taskId)` - can be called only once to associate the control with the task
- `String getTaskId()`
- `public TaskData getTaskData`
- `public TaskDataXMLDocument getTaskDataXMLDocument()`
- `public void updateTask(TaskUpdateXMLDocument doc)`

Method-level Annotations for TaskControl

The `TaskControl` control interface is not intended to be a fully functional replacement for the Worklist API. It is expected that users will define extensions of `TaskControl`, adding any appropriate methods to meet their needs for managing the task.

A `TaskControl` extension can define any number of methods with any legal Java name. Worklist does not assign any special meaning to a method name. Instead, annotations added to the methods of a `TaskControl` extension attach behaviors to those methods. A given method can have multiple annotations, and thus accomplish multiple operations on a task in a single method call.

In addition, annotations may be placed at the top-level of a control extension (before the interface declaration) that sets behavior (default or otherwise) for all methods in the extension.

TaskControl extensions can be annotated with:

- `TaskPlan` (specifies the task plan for the task that will be managed from this control)
- `TaskCreate` (acts as default values for `create` annotations on methods)
- `TaskAssign` (acts as default values for `assign` annotations on methods)

`TaskControl` allows all the method-level annotations from `TaskBaseControl`.

`TaskControl` defines the following additional method-level annotations:

Table 4-1 Method-level Annotations for TaskControl

Annotation Name	Description
TaskPlanAnno Annotation	Defines the task plan to be associated with a control. Either this or the `TaskCreate` annotation is required on a `TaskControl`.
TaskEventAnno Annotation	Callback methods only. Allows a callback method to describe subscriptions to events of a given type, and to define the method signature of that callback to the Worklist event framework. Event header information can be passed to individual parameters in the callback method based on the configuration provided in this annotation. In addition, a full `TaskEvent` instance can also be passed.

TaskBatch Control

The `TaskBatchControl` encapsulates low-level access to a batch of task instances. It does not provide event services (for these, you must use `TaskControl`). Methods on a control extension of `TaskBatchControl` can define custom task selectors that define the batch or group of task instances upon which the method will act.

The `TaskBatchControl` extends `TaskBaseControl`, and thus provides all system methods from `TaskBaseControl`. It does not define any additional system methods.

Control-level Annotations for TaskBatchControl

The `TaskBatchControl` supports the `select` control-level annotations. This acts as a default selector for methods on the control extension.

Method-level Annotations for TaskBatchControl

`TaskBatchControl` supports the following method-level annotation:

Annotation Name	Description	Fields That Support Parameters
TaskQueryAnno Annotation	Defines the query used to fetch the task instances to be manipulated in the batch	All except queryParamName and propertyValue

Merging Parameter Values with Annotation Values

Some annotations support parameter `markers` within them that can be replaced at runtime with the values of parameters provided in a method invocation. This allows dynamic data to be combined with static data from the annotation.

Annotations that support runtime parameter markers are marked with the `@SupportsParameters` annotation.

```
public @interface @SupportsParameters {

    String markerBegin() default "{";

    String markerEnd() default "}";

}
```

Any annotation field of type `String` can be annotated with the `@SupportsParameters` annotation. Any unescaped parameter markers in the annotation string value will be interpreted as replaceable parameters. For example:

```
public @interface TestAnno {

    @SupportsParameters

    String hasParams();

    String noParams();

}
```

and

```
public interface MyTest {

    @TestAnno(hasParams="Param1 has value {param1} and param2 = {param2}")
    public void testHasParams(int param1, String param2);
    @TestAnno(noParams="Param1 has value {param1} and param2 = {param2}")
    public void testNoParams(int param1, String param2);
}
```

In this case, a call to `testHasParams` with `param1=1` and `param2="Two"` yields an effective runtime value for `hasParams` of:

`Param1` has value `1` and `param2 = Two`

whereas a call to `noParams` with `param1=1` and `param2="Two"` yields an effective runtime value for `noParams` of:

`Param1` has value `{param1}` and `param2 = {param2}`

Note: Javadoc does not show meta-annotations such as the `@SupportsParameters` annotation. Refer to the method-level annotations table for the worklist controls for an indication if a given field supports parameters.

Creating Worklist Controls

You can create Task Control and Task Batch Controls and use these controls in a business process.

Creating a Task Control

You can create a task control using any of the following methods:

- Creating from the Select a Wizard dialog and from the Data Palette in the Process perspective. For more information, see "Creating a Task Control from the Select a Wizard Dialog" on page 4-17 and "Creating a Task Control from the Data Palette in Process Perspective" on page 4-19.
- Using the Task Plan Control Generator. For more information, see "Creating a Worklist Control Using Task Control Generator" on page 4-24.
- Manually creating the control. For more information, see "Manually Creating Task Control and Task Batch Control" on page 4-28

Creating a Task Control from the Select a Wizard Dialog

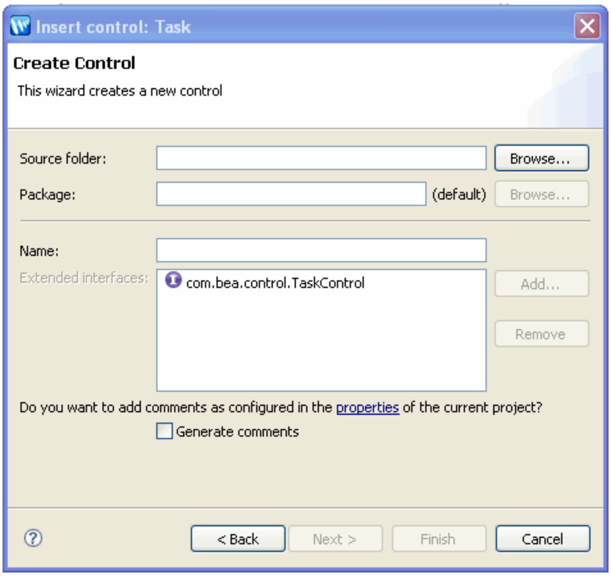
To create a Task Control that will trigger the creation of a task instance:

1. Select the `<web project name>/src` folder and click Ctrl+N**.** The Select a Wizard dialog appears.

   Note: The Web project corresponds to the Web application of Worklist that acts as the user interface for the system. For more information, see "Worklist Application" on page 2-9.

2. Click the ⊞ next to WebLogic Integration Controls and select Task. Click Next**.** The Create Control page of the Insert Control: Task dialog appears.

Figure 4-1 Creating a Task Control



Insert control: Task

Create Control

This wizard creates a new control

Extended interfaces: com.bea.control.TaskControl

Do you want to add comments as configured in the properties of the current project?

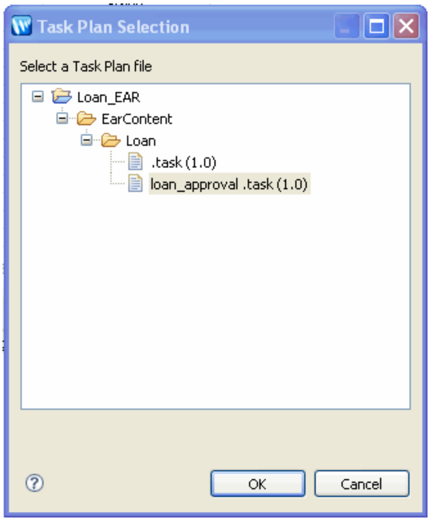
☐ Generate comments

3. Enter the name of the Task Control in the Name field.

4. Click **Next.** The Task Plan page of the Insert Control: Task dialog appears.

5. Click **Browse** to locate the task plan. The Task Plan Selection dialog appears.

Figure 4-2 Selecting the Task Plan



Task Plan Selection

Select a Task Plan file

6. Select the plan and click OK to continue.

7. Click Finish to add the new Task Control to the business process.

8. Select File > Save or use Ctrl+S to save the JPD file.

The task control is created based on the selected task plan. You can see the `<task control name>.java` file in the Package Explorer View.

Note: To use the methods in this control, drag and drop the `.java` file from the Package Explorer View into the Data palette View.

Creating a Task Control from the Data Palette in Process Perspective

To create a task control:

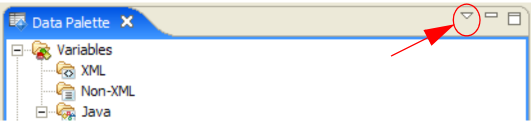
1. Make sure that the Process perspective is available. For more information, see "Selecting a Perspective" on page 1-6.

2. Select the business process file in `<web project name>/src/<process folder name>` and open it with the Process Editor.

   Note: The Web project corresponds to the Web application of Worklist that acts as the user interface for the system. For more information, see "Worklist Application" on page 2-9.

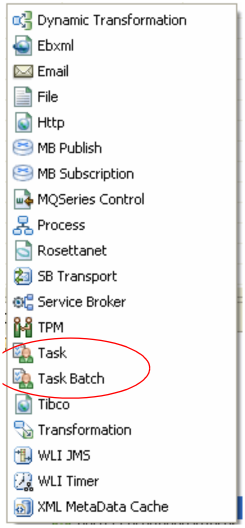
3. Click the down arrow in the Data Palette View.

Figure 4-3 Data Palette View



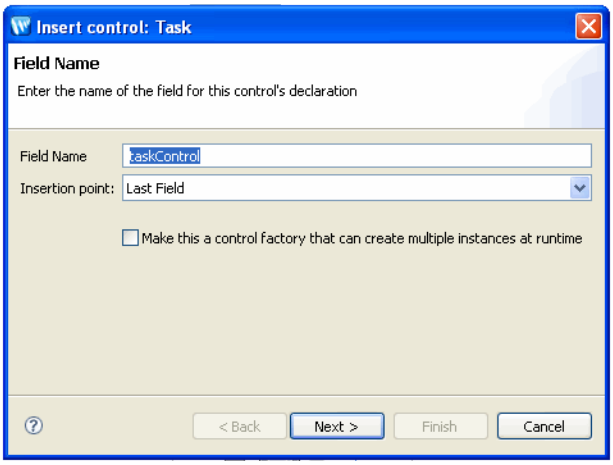
4. Select Integration Controls > Task.

Figure 4-4 Integration Controls



The Insert Control : Task dialog appears.

Figure 4-5 Insert Worklist Control


Insert control: Task
Field Name
Enter the name of the field for this control's declaration
Field Name
taskControl
Insertion point:
Last Field
□ Make this a control factory that can create multiple instances at runtime
< Back
Next >
Finish
Cancel

5. Enter the field name and specify the point at which the control should be inserted in the business process. If required, select the **Make this a control factory that can create multiple instances at runtime** check box and click Next. The Create Control dialog (Figure 4-1) appears.

6. Enter the name of the Task Control in the Name field.

7. Click Next**.** The Task Plan page of the Insert Control: Task dialog appears.

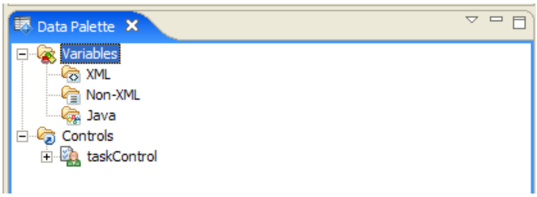
8. Click Browse and locate the task plan, and then click OK to continue.

9. Click Finish to add the new Task Control to the business process.

10. Select File → Save or use Ctrl+S to save the JPD file.

The task control is created based on the selected task plan. You can see the `<task control name>.java` file in the Package Explorer View. You can also see the new control listed in the Data Palette View.

Figure 4-6 Worklist Controls listed in Data Palette View



Creating a Task Batch Control

You can create a task batch control using any of the following methods.

- Creating from the Select a Wizard dialog and from the Data Palette in the Process perspective. For more information, see "Creating a Task Batch Control from the Select a Wizard Dialog" on page 4-22 and "Creating a Task Batch Control from the Data Palette in Process Perspective" on page 4-23
- Using the Task Plan Control Generator. For more information, see "Creating a Worklist Control Using Task Control Generator" on page 4-24.
- Manually creating the control. "Manually Creating Task Control and Task Batch Control" on page 4-28

Creating a Task Batch Control from the Select a Wizard Dialog

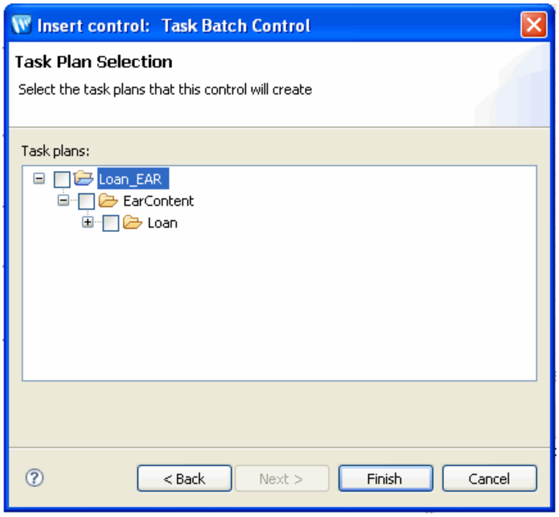
To create a Task Batch Control:

1. Select the `<web project name>/src` folder and click Ctrl+N**.** The Select a Wizard dialog appears.

   Note: The Web project corresponds to the Web application of Worklist that acts as the user interface for the system. For more information, see "Worklist Application" on page 2-9.

2. Click the ⊞ next to WebLogic Integration Controls and select Task Batch. Click Next**.** The Create Control page of the Insert Control: Task Batch dialog appears.
3. Enter the name of the Task Batch Control in the Name field.
4. Click **Next.** The Task Plan Selection page of the Insert Control: Task Batch dialog appears.

Figure 4-7 Select Task Plans


Insert control: Task Batch Control
Task Plan Selection
Select the task plans that this control will create
Task plans:
Loan_EAR
EarContent
Loan
< Back
Next >
Finish
Cancel

5. Select the required task plan and click Finish.

6. Select File > Save or use Ctrl+S to save the JPD file.

The task batch control is created. You can view the `<task batch control name>.java` file in the Package Explorer View.

Note: To use the methods in this control, drag and drop the `.java` file from the Package Explorer View into the Data palette View.

Creating a Task Batch Control from the Data Palette in Process Perspective

To create a task batch control:

1. Make sure that the Process perspective is available. For more information, see "Selecting a Perspective" on page 1-6.

2. Select the business process file in `<web project name> / src / <process folder name>` and open it with the Process Editor.

3. Click the down arrow in the Data Palette View (Figure 4-3).

4. Select Integration Controls > Task Batch (Figure 4-4). The Insert Control : Task Batch dialog appears.

5. Enter the field name and specify the point at which the control should be inserted in the business process. If required, select the **Make this a control factory that can create multiple instances at runtime** check box and click Next. The Create Control dialog appears.
6. Enter the name of the Task Batch Control in the Name field.
7. Click Next**.** The Task Plan Selection dialog appears.
8. Select the required task plans.
9. Click Finish to add the new Task Batch Control to the business process.
10. Select File > Save or use Ctrl+S to save the JPD file.

The task batch control is created based on the selected task plan. You can see the `<task control name>.java` file in the Package Explorer View. You can also see the new control listed in the Data Palette View.

Creating a Worklist Control Using Task Control Generator

You can use `TaskControlGenTask`, the Task Control Generator, to create an Apache Beehive control for a task plan. The generated control uses the TaskControl. A developer can change the annotations of the generated controls as required.

For generic information about custom controls, see Developing Custom Controls.

Generated controls take into account the constructors, steps, and actions defined in the task plan. They contain appropriate methods for creating tasks, taking actions on them, responding to events on them, and getting/setting system and user-defined property information for them.

Design the control interface to ensure that it contains:

- One method for every defined constructor in the task plan
- An inner interface for each non-terminal step in the task plan.

  A single inner interface should be generated within the generated control interface for each non-terminal step on the task plan.

  The inner interface for a step should contain a single `getTaskControl` method to allow the client to obtain a reference to the control that hosts the step interface. The inner interface should also contain a method for each action defined on the step. These methods allow the client to work the task through its life cycle from creation to completion. The methods (with the exception of `getTaskControl`) within the step's inner interface should represent the allowed actions on that step.

- A value in an inner `Step` enum representing each step in the task plan.

  This is an inner enum definition intended to enumerate the possible steps tasks of this type can be in. This enum is used to indicate the current step of the task instance from the method (generated on every generated task control extension).

- A method (on an inner interface) for each action on a step.

  For every inner interface generated for a non-terminal step, a single getter method should be generated to fetch that step's interface. This interface may be retrieved at any time (and with any step as the current step for the task). However, the action methods for this interface should be called only when the step represented by this interface is the current step for the task.

- A setter and getter for each property defined on the task plan.

  For every property on the task plan, a single getter and setter method should be generated. These methods should be strongly typed and should be named for the property they represent.

- A callback interface defining events representing:

  - The arrival of the task into any step (including terminal ones) on the task plan (the event is named for the step (example `onMyFirstStep()` and `onMySecondStep()`)
  - The standard event types defined by Worklist (these will be inherited from the `TaskControl.Callback` interface

  For every generated control extension, there should be exactly one inner interface that defines callback methods associated with events on the task. This callback interface should extend `TaskControl.Callback`, and thus inherit the callbacks for standard Worklist-defined events.

  In addition, a callback method is generated for each step in the task plan (including the terminal steps). These events should be fired and callbacks called when the task arrives at the step for which the event is named.

Running the Task Control Generator

You can run the Task Control Generator:

- From the command-line using its main method
- By calling it from another Java client
- From an Ant task that wraps the Java API.

Running the Task Control Generator from a Java Client or Command Line

The Java class name is `com.bea.wli.worklist.build.TaskControlGen` in the `WL_HOME/integration/lib/worklist-client.jar` file.

To run the generator from a Java client of the command line, type:

```
java com.bea.wli.worklist.build.TaskControlGen <options>
```

where options are listed in the following table:

Table 4-2 Command line options for running the task control generator

Option Name	Description
`-taskPlan`	The file location of the `.task` file that should be compiled. Required.
`-outputDir`	The directory where the generated control source should be output. Required.
`-hostAppRootDir`	The root directory of EAR holding the `.task` file. Optional. Can be the EARContent directory of a Java EE EAR project in Workshop. This root is used to calculate a relative path for the task plan relative to the root of the EAR. This relative path in turn is used in the default calculation of the package name for the generated control interface file. If you specify the parent directory of the `.task` file as the `hostAppRootDir`, the control interface will be placed in the `default` Java package. If not specified, the absolute path of the task type is used (potentially in the calculation of the Java package name for the generated control interface)
`-packageName`	The package name of the generated source. Optional. If the package name is not specified, the package name is derived from the task plan path. The package is calculated by replacing "/" in the path with ".", removing spaces, and replacing invalid package characters with "_".
`-interfaceName`	The name of the interface. Optional. If this value is not specified, the interface name is derived from the task plan path. The control interface name is taken from the last step in the task plan path, where spaces are removed, and invalid class name characters are replaced with '_'.

Running the Task Control Generator from Ant

The Ant task class name is `com.bea.wli.worklist.build.TaskControlGenTask` in the `WL_HOME/integration/lib/worklist-client.jar` file.

Note: We recommend that you import the task with a taskdef named `task-control-gen`.

The following Ant script describes how to use `TaskControlGenTask`:

```
<!-- Set some initial properties -->

<property environment="env"/>

<property name="wl.home" value="${env.WL_HOME}"/>

<property name="worklist-client.jar"

location="${wl.home}/integration/lib/worklist-client.jar"/>

<property name="build.dir" value="<some dir to hold build output>"/>

<!-- Define the task to Ant -->

<taskdef name="task-control-gen"

classname="com.bea.wli.worklist.build.TaskControlGenTask"

classpath="${worklist-client.jar}"/>


<!-- Give us a place to build control source files into -->

<mkdir dir="${build.dir}/controlsrc"/>


<task-control-gen

taskPlanFile="<see command-line section -taskPlan arg>"

outputDir="${build.dir}/controlsrc"

hostAppRootDir="<see command-line section -hostAppRootDir arg>"

packageName="<see command-line section -packageName arg">

interfaceName="<see command-line section -interfaceName arg">

<classpath>

<!-- Add this so TaskControlGen can see the needed classes. You can also add
your own classes (e.g. XMLBeans classes as needed) -->

<pathelement location="${worklist-client.jar}"/>

</classpath>

</task-control-gen>
```

Manually Creating Task Control and Task Batch Control

The Task Control Generator and the Worklist Control wizards should be viewed as shortcuts to creating Worklist controls in ways that users commonly need. However, you may create `TaskControl` and `TaskBatchControl` extensions manually if they like. The only requirement for these extensions is that they extend `TaskControl` or `TaskBatchControl` and are marked with `@ControlExtension` at the interface level.

You may add any number of methods to these control extensions and mark them up using any of the annotations supported for the given type of control. It may be appropriate to create your own custom control extensions in the following cases (and possibly others as well)

- You want a control interface that uses different names for methods than those generated for you by the Task Control Generator or Worklist Control wizards.
- The users of the control do not need to see all the methods generated by the Worklist tools, and you want to define a very focused and lightweight control.
- You do not want to use the inner step interface paradigm of the task controls generated by the Task Control Generator. You can create separate task controls for each step interface, and simply use the correct control at the correct step in the task plan.

In all these cases, you can start with a control as generated by the Worklist tools, and then manually update the control interface as desired.

Using Task and Task Batch Controls in Business Processes

This section provides information about using business process files (JPDs) and Worklist Controls to support the integration of human actors and automated actors. Automated actors interact with the JPD and human actors interact with Worklist User Portal.

As with other built-in controls in Workshop for WebLogic Platform, you use the controls by adding instances of the controls to your business process. Subsequently, you invoke operations on the controls at the point in the business process at which you want to integrate the business-user logic.

To design the interaction of a Task or Task Batch control with a business process, you must decide which methods on the control you want to call from the business process to support the business logic.

In the same way that you design the interactions between business processes and other controls in WebLogic Workshop, you can bind the Worklist control method to the appropriate control node in your business process (Control Send, Control Receive, and Control Send with Return). You do this in the Design View by simply dragging a control method from the Data Palette onto the business process at the point in your business process at which you want to design the logic. After you create an instance of a Task or Task Batch control, you can invoke its methods from within your business processes to perform operations on task instances. Your business processes can also wait to receive callbacks from task instances. Note that the Task and Task Batch controls can be extended to add customized methods and additional callbacks.

You can create a task instance using a Task Control or Task Batch Control in a JPD. During this process, you need to create a business process file and a Task Control or Task Batch Control, deploy the process. If required, you can create a sample task to validate the task instance creation.

You can use worklist controls in your business process by:

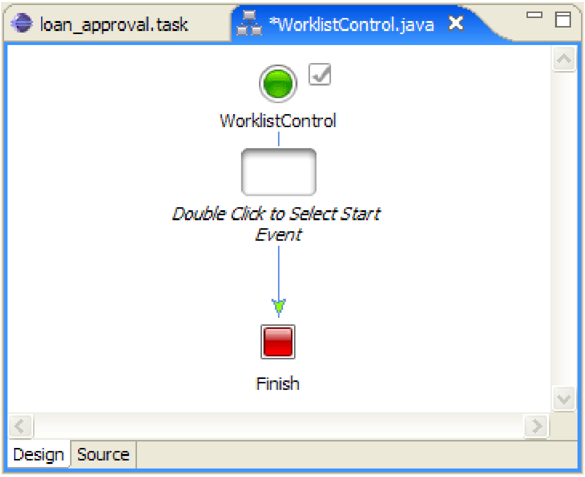
- Creating a Business Process
- Creating Worklist Controls
- Invoking the Task Plan Constructor from the Business Process
- Deploying and Validating the Business Process

Creating a Business Process

To create a business process that will use the worklist controls:

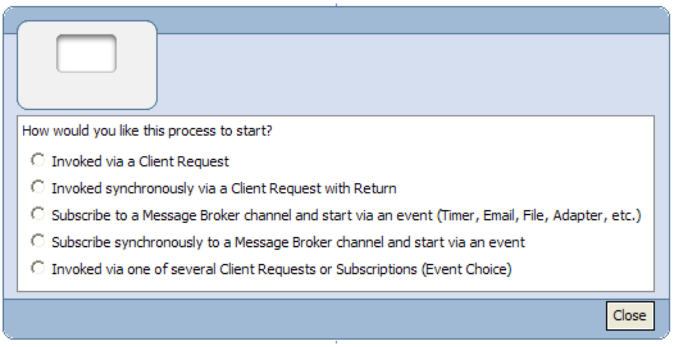
1. In the Package Explorer view, right-click the `<web project name>\src` folder, and select New > Folder. The New Folder dialog appears.
2. Enter the name of the folder in which the business process files are stored.
3. Click Finish.
4. Select the project folder and press Ctrl+N**.** The Select a Wizard dialog appears.
5. Select WebLogic Integration > Process and click Next. The New Process File dialog appears.
6. Enter a name for the business process file. This creates a JPD process file `<process name>.java` under the process folder.
7. Click Finish to complete the process. The new JPD in design view is displayed in the IDE browser.

Figure 4-8 New JPD Using Worklist Control



8. Double-click Select Start Event (Start node). A screen that allows you to select the manner in which the business process should be invoked appears.

Figure 4-9 Invoking a business process



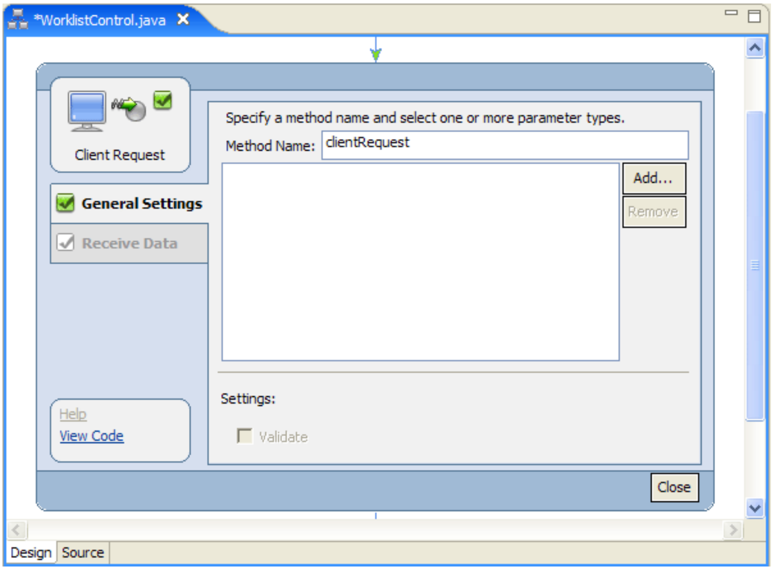
How would you like this process to start?

- ○ Invoked via a Client Request
- ○ Invoked synchronously via a Client Request with Return
- ○ Subscribe to a Message Broker channel and start via an event (Timer, Email, File, Adapter, etc.)
- ○ Subscribe synchronously to a Message Broker channel and start via an event
- ○ Invoked via one of several Client Requests or Subscriptions (Event Choice)

9. Select the manner in which the business process is invoked and click Close. For example, select Invoked via a Client Request. The JPD design view is refreshed and the Start node is named `Client Request`.

10. Double-click Client Request node to configure it.

Figure 4-10 Configuring Client Request



Specify a method name and select one or more parameter types.

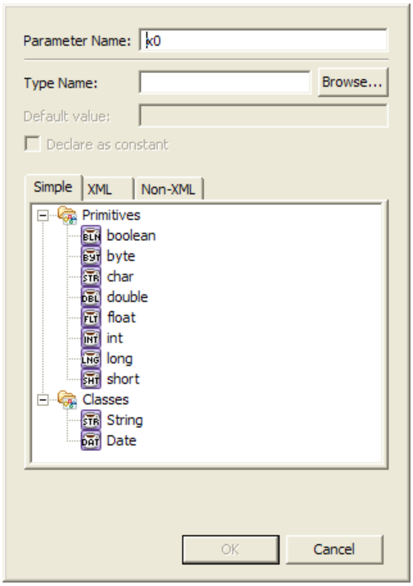
Method Name: clientRequest

Settings:

□ Validate

11. In the General Settings tab, click Add to display a dialog box for defining parameters. Create parameters and configure their types based on the user properties required for the task.

Figure 4-11 Add Parameters


Parameter Name: x0

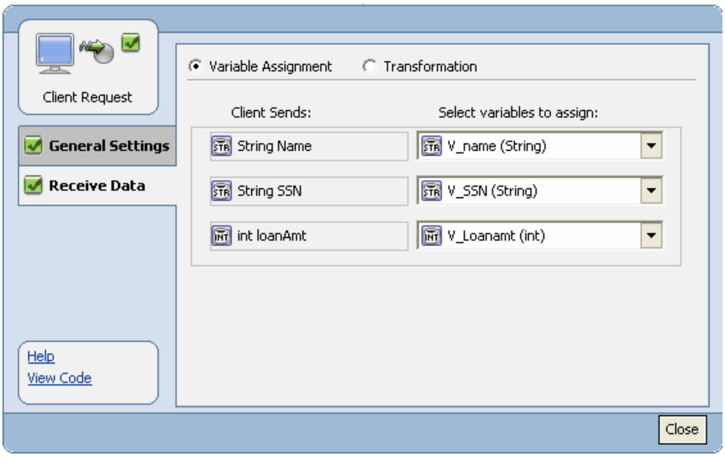
Type Name:

Default value:

□ Declare as constant

12. Click the Receive Data tab to create new variables and assign them the respective parameters created in the previous step.

Figure 4-12 Assigning Variables to Parameters


● Variable Assignment ○ Transformation

Client Sends:	Select variables to assign:
String Name	V_name (String)
String SSN	V_SSN (String)
int loanAmt	V_Loanamt (int)

13. Click Close to continue.

Invoking the Task Plan Constructor from the Business Process

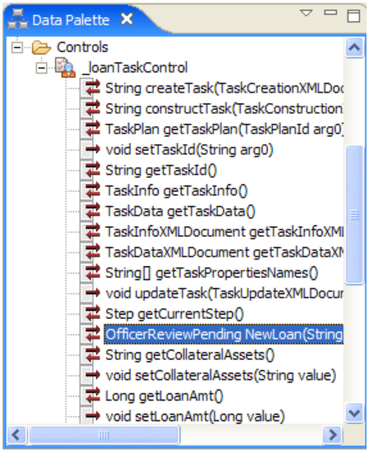
To invoke the task instance creation constructor to the business process.

1. Right click the `<worklist business process name>.java` file in the Package Explorer pane and select Open With → Process Editor option. Ensure the JPD is displayed in the Design tab and that you are using the Process Perspective.

2. From the Data Palette View on the bottom right corner of the IDE, navigate to <control name> folder in the Controls folder and select the method that you want to invoke in the business process.

   Constructor methods on the control will have the name of the constructor defined in the task plan, and will take a task name as the first argument in the method signature. Constructor methods also return the step interface for the start step designated by the constructor in the task plan.

Figure 4-13 Selecting the Control Method

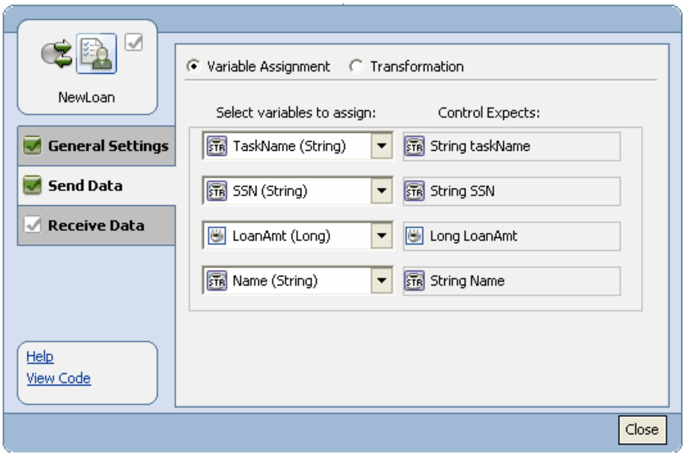


Controls
_loanTaskControl
String createTask(TaskCreationXMLDoc
String constructTask(TaskConstruction
TaskPlan getTaskPlan(TaskPlanId arg0)
void setTaskId(String arg0)
String getTaskId()
TaskInfo getTaskInfo()
TaskData getTaskData()
TaskInfoXMLDocument getTaskInfoXMl
TaskDataXMLDocument getTaskDataXM
String[] getTaskPropertiesNames()
void updateTask(TaskUpdateXMLDocur
Step getCurrentStep()
OfficerReviewPending NewLoan(String
String getCollateralAssets()
void setCollateralAssets(String value)
Long getLoanAmt()
void setLoanAmt(Long value)

3. Drag and drop the selected method into the JPD, between the Client Request and the Finish nodes.

4. Double click the method (displayed as a node in the Design View) and configure the Send Data properties.

Figure 4-14 Map the Send Data Variables with the Control Parameters



NewLoan

☑ General Settings

☑ Send Data

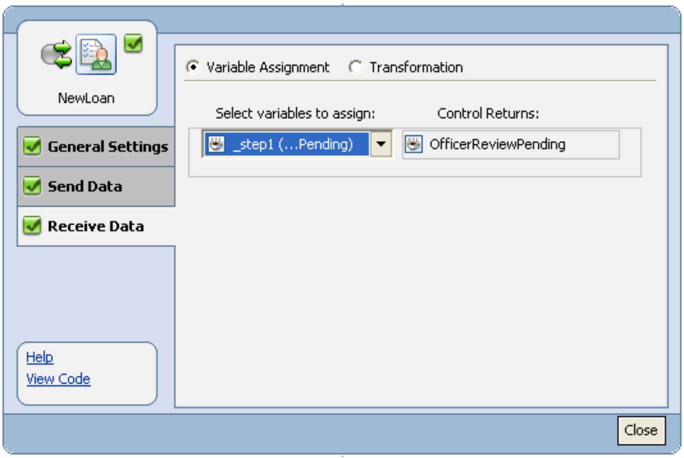
Receive Data

● Variable Assignment ○ Transformation

Select variables to assign:	Control Expects:
TaskName (String)	String taskName
SSN (String)	String SSN
LoanAmt (Long)	Long LoanAmt
Name (String)	String Name

5. Map the receive data variables for the method to variables in the business process. Click Receive Data, accept the given data type of the return value from the control method invocation, and then give a variable name, which will hold the returned step interface.

Figure 4-15 Map Receive Data Variable



NewLoan

☑ General Settings

☑ Send Data

☑ Receive Data

● Variable Assignment ○ Transformation

Select variables to assign:	Control Returns:
_step1 (...Pending)	OfficerReviewPending

6. Optionally, take actions on the task using the returned step interface from step 5. Note that the process designer does not treat the step interface as a separate control. Because of this, you will need to use perform nodes to invoke the methods of the step interface. You must write the code manually to invoke these methods. You may choose to have one perform node for

each method you call, or you can group many method calls into a single perform node. The benefit of the “one method per perform node” approach is that you will get a graphical representation of the methods being called on the task just by looking at the process flow graph.

7. Click Close and save the JPD.

Deploying and Validating the Business Process

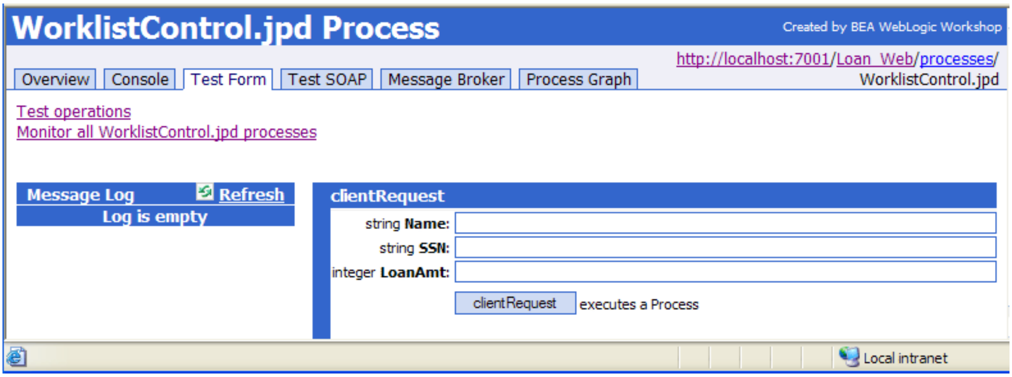
To deploy and validate the business process:

1. Select the JPD is selected in the Package Explorer View and select the Run > Run menu option.

   After successful deployment the JPD process page is launched in the IDE browser.

2. Click the Test Form tab of the process browser.

Figure 4-16 Business Process in Test Form Page



3. Enter the test values in their respective fields.

4. Click ClientRequest to execute the process with the test values. After a successful execution, the TestForm tab is refreshed

5. Start a Worklist User Portal session and log in using the user credentials.

The Assigned Tasks portlet box displays the task instance created by a JPD using a Control.

CHAPTER 5

Worklist Event-based Services

This section describes advanced topics for use by a Java programmer in BEA Workshop for WebLogic. It provides information about:

- Task Change Events
- EventHandler Modules
- Default Reporting Store
- Email Notification
- MessageBroker and JPD Integration
- Custom Task Event Listeners
- Custom Assignment Handlers

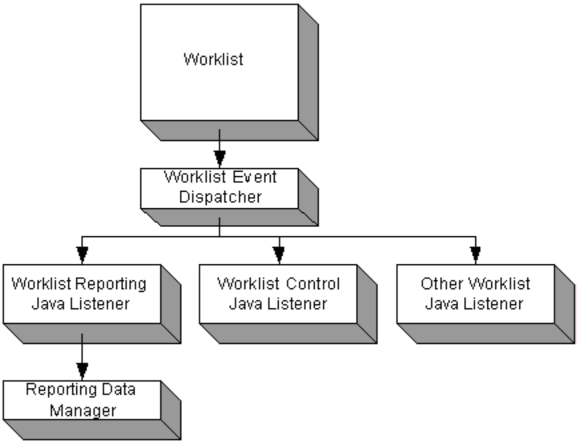
Task Change Events

Worklist defines task change events that can be triggered when there is a change in the state of a task. Worklist accepts subscriptions for events from interested parties (called listeners) and notifies them when events they have expressed interest in occur. Listeners can then take appropriate actions based on the events that occur to a task.

Event Delivery Paths

The possible paths taken by events generated by Worklist could be the Worklist reporting subsystem, web services listeners, and plain Java listeners.

Figure 5-1 Event Delivery Paths


Worklist
Worklist Event Dispatcher
Worklist Reporting Java Listener
Worklist Control Java Listener
Other Worklist Java Listener
Reporting Data Manager

Note: The listeners shown in the illustration are all Worklist event listeners and are internal system-level listeners. Custom task event listeners are plugged into this tree via `.listener` files and are registered as siblings of the Worklist default event listeners.

Events

Worklist defines a set of event types to support both runtime and reporting event scenarios. The assumption is that the only real difference between the two usages of events is that reporting more clearly requires both the old and new values of a change in a task. Runtime use of events could conceivably be satisfied to simply retrieve the current value from the task. We assume that in any case a task property is updated, the consumer of the event will query current property values using the Worklist API.

The following table defines the events that are produced by Worklist in response to various operations on a task. All events contain the following data items (accessible via Java methods on the `com.bea.wli.worklist.api.events.data.TaskEvent` interface):

- Event Type
- TaskId

- TaskName
- TaskPlanId
- Creator
- Owner
- Claimant
- AssigneeList (AssigneeDefinition[])
- Caller (user name of the user that caused the change)
- Timestamp (long value representing local system time)
- ServerName (name of the WebLogic Server from which this event originated)
- HostApplicationName (The name of the application on which this task change occurred).
- Summary (A summary string that gives relevant information about the event in a localized message format).

The following table lists any additional fields added to a specific event type:

Event Type	When	Extra Fields	Event Object Type
CREATE	Task is created. Will be followed by a step change event and an assigned or claimed event that will arrive separately after the create event.	TaskData, and Initial Property names for task. (sync listeners only - Initial Property Values)	TaskCreateEvent
ASSIGN	Task is assigned, either directly or as a result of moving to a new step, or taking an assign action on the step.	NewAssigneeList, OldAssigneeList	TaskAssignEvent
CLAIM	Task is claimed, either directly or as a result of moving to a new step, or taking an assign action on the step. If a task is auto-claimed, only the claimed event is sent. Not the assigned event.	NewClaimant, OldClaimant, (if auto-claimed - NewAssigneeList, OldAssigneeList)	TaskClaimEvent
RETURN	Task is returned	Reason, OldClaimant	TaskReturnEvent

Event Type	When	Extra Fields	Event Object Type
TAKE_ACTION	An action is taken. May be followed by • a STEP_CHANGE event after a work action that targets a different step. • assign/claim events if the task was assigned/claimed (after arriving at the new step after a work action, or after an assign action). • a return event after a return action. • complete, abort events after a work action if the new current step is a complete or abort step.	NewStep, OldStep, Action, Property Names for Action (sync listeners only - NewValue for properties)	TaskStepActionEvent
STEP_CHANGE	The current step of the task changes. This can happen administratively or via a work action.	NewStep, OldStep	TaskStepChangeEvent
SET_USER_PROPERTY	A task plan-defined property is updated directly via `WorklistTaskUser.setTaskProperties()`. If the property is updated as part of taking an action, the names of the modified props will be represented on the TAKE_ACTION event.	PropertyName (sync listeners only - OldValue, NewValue)	TaskUpdate PropertyEvent
SUSPEND	A task is suspended	Reason	TaskAdminEvent
RESUME	A task is resumed	Reason	TaskAdminEvent
SET_ERROR	A task is set to error state	Reason	TaskAdminEvent
CLEAR_ERROR	A task is reset from error set	Reason	TaskAdminEvent
COMPLETE	A task is completed	Reason	TaskAdminEvent
ABORT	A task is aborted	Reason	TaskAdminEvent
REACTIVATE	A task is reactivated	Reason, NewCurrentStep, OldCurrentStep	TaskReactivateEvent

Event Type	When	Extra Fields	Event Object Type
SET_CURRENT_STEP	A task has it's current step forcibly set. Will be followed by a STEP_CHANGE event indicating the arrival of the task into the new step. May be followed by assign, claim, complete, or abort events depending on the new current step.	Reason, OldCurrentStep, NewCurrentStep	TaskSetCurrentStep Event
SET_SYSTEM_PROPERTY	Any system property is set directly via `WorklistTaskUser.setTaskProperties()` or one of the `setTask<Name>()` methods. For information on which system properties generate this type of event, see "System Properties and TaskUpdateBuiltInProperty Events" on page 5-5.	PropertyName (sync listeners only - OldValue, NewValue)	TaskUpdateBuiltInPropertyEvent
STEP_EXPIRE	A step of a task was not completed before its scheduled completion date.	ScheduledExpireDateTime, StepName	TaskStepExpireEvent
TASK_EXPIRE	A task was not completed before its scheduled completion date.	ScheduledExpireDateTime	TaskExpireEvent

System Properties and TaskUpdateBuiltInProperty Events

The following system properties may (when updated) cause a `TaskUpdateBuiltInProperty` event to be delivered from Worklist. Other system properties are read-only and will not cause update events. These property names match the enumeration values from the Worklist `com.bea.wli.worklist.api.taskplan.SystemProperty` enumeration.

- TASK_NAME
- DESCRIPTION (Deprecated, and only from 8.1.x API)
- COMMENT
- PRIORITY
- OWNER
- IS_OWNER_GROUP

- CURRENT_STEP_COMPLETION_DUE_DATE
- CURRENT_STEP_CLAIM_DUE_DATE (Deprecated, and only from 8.1.x API)
- TASK_COMPLETION_DUE_DATE
- TASK_TIME_ESTIMATE
- STEP_TIME_ESTIMATES

Controlling Events Per-Task Plan

Various types of BEA-provided event listeners can perform *producer-side filtering* to limit the number of events sent to their external endpoints. This filtering is controlled by event configuration contained in the EventHandler configuration module. For more information, see "EventHandler Modules" on page 5-15.

Event Dispatch Modes (Quality of Service)

Worklist defines a Java listener interface (see *"Custom Task Event Listeners" on page 5-43* section) that Java clients can use to receive events when changes occur on a task instance. A listener can specify the quality of service it requires from Worklist. This quality of service is defined in terms of:

- Synchronous vs. Asynchronous Delivery
- Critical vs. Non-Critical Handling

Synchronous delivery means the listener is called on the stack from the same thread that caused the event to be created. Asynchronous delivery means that the event is placed on a queue and then delivered to the listener on a different thread than the thread that caused the event to be created.

Listeners may indicate their preferred dispatching mode when registering via their `.listener` file.

A critical listener is a listener that absolutely must receive and process all events for which it has subscribed and which may set an error on the task if the event cannot be consumed successfully. A non-critical listener is a listener that can tolerate lost events and may not successfully process some events without setting an error on the task for the event. Events destined for asynchronous critical listeners are put into a reliable and persistent queue and are delivered even in the face of a server crash. Events for non-critical listeners are delivered with no persistence and are not guaranteed to survive a server crash.

Listeners registered as a synchronous *critical* listener can exercise some amount of control over task changes by throwing an exception from its `onTaskEvent()` method. If a synchronous critical listener throws an exception from `onTaskEvent()`, the current transaction (the one that caused the event in the first place) is rolled back. In this case, the change is effectively undone. The task instance is not put into the error state, because to do so requires a live database transaction that can be committed.

All critical listeners are notified of events before non-critical listeners. If a critical listener throws an exception, no further dispatching of the current event is done (and the transaction holding the task change is rolled back).

Note: System administrators should carefully examine any Worklist application containing `.listener` files. They should scrutinize any `.listener` file that registers itself as a synchronous user, as this listener will run under the identity of the user making the updates to the tasks that caused the events. Thus, the listener code can do anything the original user could do. Only trusted code should be allowed to deploy as a synchronous listener.

EventListener Interface

Listeners implement a Java listener interface and are plugged into the event delivery mechanism using this listener implementation. Java listeners must implement the following listener interface:

```
public interface TaskEventListener {


    /**
     * Get the name assigned to this listener
     */
    public String getName();
    /**
     * Get the name for this listener
     */
    public void setName(String name);


    /**
    * Set any properties configured for this TaskEventListener in the custom
```

```
 * module that deployed it. This method is only called if properties are
  * available (example specified in the deployment descriptor for the
  * custom module).
  * @param properties
  */
public void setProperties(Map<String, Property> properties);

/**
  * Initialize any resources needed to start handling events in
  * onTaskEvent().
  * @throws com.bea.wli.worklist.api.ManagementException
  */
public void initialize()
    throws ManagementException;

/**
* Destroy this instance and release any resources obtained in initialize
  * or during calls to onTaskEvent().
  * @throws ManagementException
  */
public void destroy()
    throws ManagementException;

/**
  * Handle the given event.
 * @throws VetoException If any error occurs processing the event. How
  * this exception is handled by Worklist depends on how the listener
```

```
     * is registered (sync/async, critical/non-critical)
     */
    public void onTaskEvent(TaskEvent event)
        throws VetoException;
}
```

where `TaskEvent` is:

```
public interface TaskEvent
    extends Serializable {

    /**
     * The type of event this event instance represents. Listeners can use
     * this property for simple filtering. In order to obtain event
     * type-specific properties from the event, listeners should cast this
    * event to the appropriate event class based on the runtime type of this
     * event instance.
     */
    public TaskEvent.Type getEventType();

    /**
     * ID of the task that was changed.
     */
    public String getTaskId();

    /**
     * Name of the task that was changed
     */
    public String getTaskName();
```

```
        /**
         * ID of the task that was changed.
         */
        public TaskPlanId getTaskPlanId();

        /**
         * User that created this task.
         */
        public String getCreator();

        /**
         * Current owner at the time this event was created.
         */
        public String getOwner();

        /**
         * Current claimant at the time this event was created.
         */
        public String getClaimant();

        /**
         * Current assignee list at the time this event was created.
         */
        public AssigneeDefinition[] getAssigneeList();

        /**
```

```
 * Current admin state at the time this event was created.
 */
public AdminState.Type getAdminState();

/**
 * Current working state at the time this event was created.
 */
public WorkingState.Type getWorkingState();

/**
 * User name of the user that caused the change.
 */
public String getCaller();

/**
 * Timestamp (long value representing local system time).
 */
public long getTimestamp();

/**
 * The name of the server upon which this task change occurred.
 */
public String getServerName();

/**
 * A summary of the most relevant information in this event, limited to
 * 100 characters.
 */
public String getSummary();

/**
* Enumeration for different types of events. Note that the CLAIM_EXPIRE,
* START and STOP event types are deprecated and should not be used in new
 * development.
 */
public enum Type {
    CREATE,
    ASSIGN,
    CLAIM,
    RETURN,
    TAKE_ACTION,
    STEP_CHANGE,
    SET_USER_PROPERTY,
    //PROPERTY_SET_LOCKSTATE,
    SUSPEND,
    RESUME,
    SET_ERROR,
    CLEAR_ERROR,
    COMPLETE,
    ABORT,
    REACTIVATE,
    SET_CURRENT_STEP,
    SET_SYSTEM_PROPERTY,
    CLAIM_EXPIRE_81X, // deprecated
```

```
        STEP_EXPIRE,
        TASK_EXPIRE,
        START_81X, // deprecated
        STOP_81X;  // deprecated

        public String getValue() {
            return this.name();
        }

        public static Type getFromOrdinal(int ordinal) {
            Type[] types = Type.class.getEnumConstants();
            for (int i=0; i < types.length; i++) {
                if (types[i].ordinal() == ordinal) {
                    return types[i];
                }
            }
            return types[0];
        }

    };

}
```

Notes: A listener must be coded with the following points in mind:

- It must have a public no-arg constructor. Use the `setProperties()` method to obtain external configuration, the `initialize()` method to obtain or initialize any external resources, and the `destroy()` method to release those resources.

- If a synchronous critical listener throws an exception from the `onTaskEvent()` method, the processing of the action that caused the event is aborted and the current transaction is rolled back (thus undoing the action that caused the event).
- If a synchronous non-critical listener throws an exception from the `onTaskEvent()` method, the exception is logged and execution will continue normally.
- If an asynchronous critical listener throws an exception from the `onTaskEvent()` method
  - the task is placed into an error state with the message from the exception as the error message
  - the event is placed back onto the queue and delivery is retried (up to a configurable maximum number of retries and with a configurable retry delay. The configuration of this is done via JMS)
- If an asynchronous non-critical listener throws an exception from the `onTaskEvent()` method, the exception is logged, the message is removed from the queue (not retried) and execution will continue normally.
- Synchronous listeners have access to property values delivered in CREATE, TAKE_ACTION, SET_USER_PROPERTY, and SET_SYSTEM_PROPERTY event types. Asynchronous listeners receive these events with the property values cleared out. Asynchronous listeners must use the Worklist API to obtain property values for the properties indicated (by name) on the event.

So, developers should only register listeners as critical when it is really necessary to the business that the listener receives all events it intends to consume. Also, developers should take precautions not to throw exceptions unless it is absolutely necessary to do so.

Installing EventListeners

See the "Deploying Custom Event Listeners" on page 6-9 section for details.

Event Listener and Run-As Identity

Synchronous listeners (critical or non-critical) run under the identity on the thread that created the event being handled. Asynchronous listeners run under the identity configured for the TaskEventDispatcher MDB (by default this is anonymous, but can be overridden via the WebLogic Server Administration Console).

Note: System administrators should carefully examine any Worklist application containing `.listener` files. They should scrutinize any `.listener` file that registers itself as a synchronous user, as this listener will run under the identity of the user making the updates to the tasks that caused the events. Thus, the listener code can do anything the original user could do. Only trusted code should be allowed to deploy as a synchronous listener.

EventHandler Modules

The EventHandler module is a configuration module that is used to define configuration for the delivery of reporting, email notification, and other types of events.

An EventHandler module is deployed as part of a Worklist host application. It is a WebLogic Server configuration module that can be added to the host application to configure the delivery of events for tasks of the types defined in that host application.

The configuration of an event handler includes some general information as well as listener-type-specific elements. An event handler defines:

- A unique name for the handler within the host application
- The task plans this handler is designed to work with. Handlers can be global (all task plans) or list specific task plans by their Task Plan ID
- A disabled flag, which if true, forces the runtime to ignore the handler while the flag is set. Other handlers remain in force.
- One or more event subscriptions

An event subscription also contains general and listener-type-specific elements. An event subscription defines:

- A unique name for the subscription within this handler
- A list of one or more event types that are applicable to this subscription. Subscriptions can be applied to all event types by setting a `ForAllEventTypes` flag on the subscription.
- A disabled flag, which if true, forces the runtime to ignore the subscription while the flag is set. Other subscriptions on this handler and other handlers remain in force.
- Listener type-specific sections to configure the delivery of the given event types to the listener's endpoint.

The listener type-specific sections are defined for the needs of these types of listeners:

- Email Notification
- Reporting

- MessageBroker

At runtime, event subscriptions in a handler are collected together with the event subscriptions from all other handlers in the application. These subscriptions are additive with each other. The event configuration in a given event handler is merged additively with the configuration of all other event handlers (including the default Worklist event handlers).

Event Handler File

An EventHandler is defined in a `.handler` file as an XML document. The XML content must conform to a schema for EventHandlers. EventHandler (`.handler`) files can be edited in the Workshop IDE using the EventHandler editor plug-in.

Note: Once a `.handler` file exists in an application, you may edit the configuration it contains using Worklist Console. This allows you to make your best judgement for the event types that must be handled, and then fine tune the handling at runtime.

Deployment

The EventHandler module is deployed when the host application (and task plans it contains) are deployed. At runtime, the various listener types consult the deployed EventHandlers to determine if an event is of interest to their type of listener, and if so deliver it to their endpoint. This delivery is controlled and configured by the listener-type-specific configuration in the handler.

For more information, see the "Deploying Event Handlers" on page 6-3 section.

Event Handler Editors

Worklist Console provides a facility to edit event handlers, and the listener-type-specific configuration within them. For more information, see "Changing the Event Handler Details" in Worklist Administration.

The BEA Workshop for WebLogic IDE has no direct support for editing `.handler` or `.listener`, etc. files. However, the IDE registers these types of files as custom modules. This allows them to be deployed automatically (with no other action on the part o the user). However, editing these files is manual, and should be done according to the examples given in the respective sections of this document.

Default Event Configuration

Worklist deploys a single default event handler, called WorklistEventHandler that defines event delivery for the following types of listeners:

- Email Notification
- Reporting
- MessageBroker

This file is included as part of the Worklist J2EE application library and is a global event handler (applies to all task plans). You can use Worklist Console to change the default event configuration in this handler. For more information, see “Changing the Event Handler Details” in Worklist Administration.

Default Reporting Store

An event-based mechanism that allows a customer to detect changes to the Worklist runtime repository and make corresponding changes to their own reporting repository is available. Worklist does not dictate or be aware of, in any way, the design of the customer reporting repository. In addition, an interface for customer-provided reporting repositories is defined to make task history information available to Worklist clients.

For out-of-the-box reporting use, Worklist provides a default reporting repository and a default reporting facility that allows you to glean some amount of long-term reporting information without having to implement your own reporting repository.

Note: If your application requires information that is not easily gleaned from the default reporting repository, consider defining your own reporting repository. You can attach this repository to Worklist events and even disable the default reporting store completely, if desired. For more information, see “Task Change Events” on page 5-1.

Task History

For some types of tasks, where a task has been is equally as important as where the task was defined to go. In most of these cases, the task can be loosely defined to move through some broadly scoped steps, but in general, the actual path of the task, the human actors it will be associated with, and so on are not known in detail ahead of time.

An example of such a task is the handling of customer service trouble tickets or the resolution and tracking of defects in a software or hardware product. In these cases, there is a general procedure put in place to track all types of trouble and defects, but each case has its own significantly unique considerations. In such a case, the history of a task may be critical to helping human actors that subsequently encounter the task or are associated with it. For example, a software defect may at first appear to be in one software component, but upon investigation by a software engineer attached to the first component, it is determined the defect is in a second software component.

Here, the analysis and findings of the first engineer may be critical to the second engineer in finding and fixing the problem.

Task History Provider

A Task History interface is defined as:

```
package com.bea.wli.worklist.api.config;


/**
 * Defines a provider of task history information. This provider
 * may be registered for all task plans or selected task plans.
 */
public interface TaskHistoryProvider {


    public void setProperties(Property[] props);


    public void initialize()
        throws ManagementException;


    public TaskHistory getTaskHistory(String taskId)
        throws RemoteException, ManagementException;


    public void destroy()
        throws ManagementException;
}


public interface TaskHistory {
    /**
     * Get an iterator over all events that have occurred for this task
     * from the startDate to the endDate (inclusive).
     */
    public TaskEventIterator
    getTaskEvents(Date startDate, Date endDate)
        throws RemoteException, ManagementException;
    /**
     * Reconstruct the state of a task on a given date. The provider
     * should use its history information up to the given date in
     * constructing the returned TaskData. If providers do not support this
     * they should throw NotSupportedException.
     */
    public TaskData getTaskAtDate(Date date)
       throws NotSupportedException, ManagementException, RemoteException;
}

public interface TaskEventIterator {
    public boolean hasNext()
        throws RemoteException;
    public TaskChangeEvent next()
        throws RemoteException, ManagementException;
}
```

When you implement the custom reporting repository, you may optionally define your own TaskHistoryProvider implementation and then register that implementation globally or for selected task plans. The TaskHistoryProvider is registered by deploying a TaskHistoryProvider custom module. For more information see the “Deploying Custom Task History Providers” on page 6-18 section for details.

Worklist provides the `WorklistTaskUser.getTaskHistory()` method as a front-end to the server-side TaskHistoryProvider.

```
public interface WorklistTaskUser {
    …

    /**
     * Is the task plan associated with a task history
     * provider. If this returns true, you can reasonably expect to get
     * non-empty results from getTaskEvents().
     */
    public boolean isTaskHistoryProvided(TaskPlan taskType);

    /**
     * Get all events that have occurred, sorted by date, on this task
     * from the start date to the end date (inclusive). If no task history
     * is available for this task, an empty array is returned. This may
     * happen if the date range falls in during a period of inactivity
     * for the task, or it may indicate that no task history information
     * is being captured for tasks of this type.
     */
    public TaskEvent[] getTaskEvents(String taskId,
                                     Date startDate, Date endDate)
        throws ManagementException, RemoteException;

    …
}
```

Worklist provides a TaskHistoryProvider implementation for the Worklist default reporting store. This default TaskHistoryProvider is registered automatically as a *global* provider, thus applying to all task plans.

At most one TaskHistoryProvider can be associated with any given task plan. If a custom TaskHistoryProvider is registered for a given task plan, that provider becomes the sole provider of task history information for that task plan (replacing the default/global TaskHistoryProvider). Registering a global TaskHistoryProvider replaces the default TaskHistoryProvider. This should be done with caution, and only when a global TaskEventListener has been registered that will take responsibility for capturing task events need to generate the task history for all task plans.

Reporting Store Design

The reporting store is intended to provide at least the level of reporting functionality available in Worklist 8.1 with the added benefit of being able to do queries over live and historical data (in 8.1, records that had been archived had to be queried separately from the information in the Worklist runtime store).

The technical design goal of the reporting store is to store as much data as possible with the smallest processing overhead as possible. It is assumed that the reporting process or user interface can absorb the processing hit for focused tasks in the reporting store more easily than the runtime can absorb the processing hit across all events it generates.

Business Goals

The business goal of the reporting store is to support the following general types of queries:

- Audit/update trail for a particular task instance
  - When was this task created?
  - What steps did it progress through?
  - Who took the action to move from step A to step B and when was it taken?
- Gathering statistics
  - How long are tasks of a given type taking to complete?
  - What percentage of tasks of a given type are completing normally and what percentage are being aborted or encountering errors?
  - How many tasks of a given type is a given user completing?
  - How long is it taking for a given user to complete tasks of a given type?
  - How many tasks of a given type have we processed this week?
- Task Plan-specific queries. For example,
  - Based on the purchase orders for this month, what companies are we paying the most money to? (This query might be based on the *PurchaseOrder* task plan, and look into

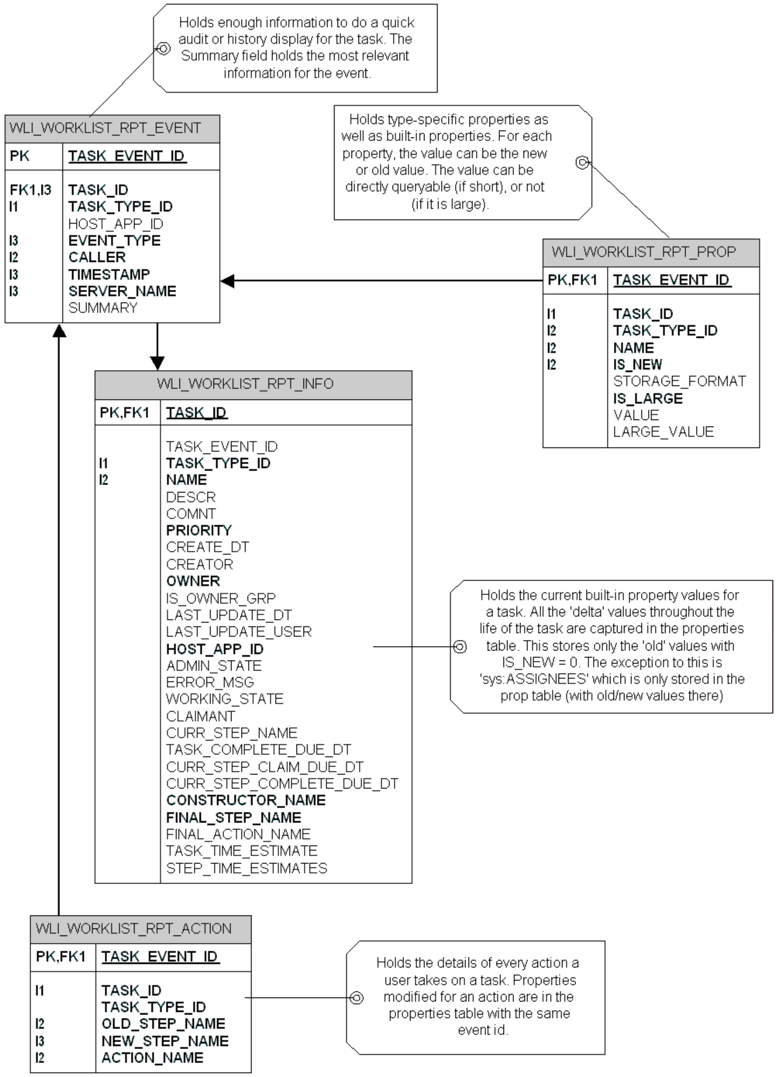
the purchase order property of PurchaseOrder tasks, calculating the total amount of the purchase order, and the payee for the purchase order).

- Based on the expense reports for this quarter, how much money have we spent on hotel expenses? (This query might use the expense report property of *ExpenseReport* tasks, and look for all the items marked as *Hotel*, between the first and last day of the month).

Database Schema

The following diagram shows an example of the reporting store schema. For a few examples of how the reporting store may be used to retrieve data to satisfy the business goals set out in "Business Goals" on page 5-21, see "Sample Queries" on page 5-23.

Figure 5-2 Reporting Store Schema



Holds enough information to do a quick audit or history display for the task. The Summary field holds the most relevant information for the event.

Holds type-specific properties as well as built-in properties. For each property, the value can be the new or old value. The value can be directly queryable (if short), or not (if it is large).

Holds the current built-in property values for a task. All the 'delta' values throughout the life of the task are captured in the properties table. This stores only the 'old' values with IS_NEW = 0. The exception to this is 'sys:ASSIGNEES' which is only stored in the prop table (with old/new values there)

Holds the details of every action a user takes on a task. Properties modified for an action are in the properties table with the same event id.

Sample Queries

Example: Show me everything that has happened with task XYZ to date

```
select * from WLI_WORKLIST_RPT_TASKEVENT
```

```
  where TASK_ID='XYZ'
  order by TIMESTAMP
```

Example: Who took the action to move task XYZ from step A to step B, and when was it taken?

```
select e.CALLER, e.TIMESTAMP
  from WLI_WORKLIST_RPT_TASKEVENT e, WLI_WORKLIST_RPT_TASKACTION a
  where e.TASK_ID = 'XYZ' and
        e.TASK_EVENT_ID = a.TASK_EVENT_ID and
        a.OLD_STEP_NAME = 'A' and a.NEW_STEP_ID = 'B'
```

Example: How long is it taking for Bob to complete CustomerRefund tasks?

```
select avg(complete.TIMESTAMP - create.TIMESTAMP)
  from WLI_WORKLIST_RPT_TASKEVENT create, WLI_WORKLIST_RPT_TASKEVENT
complete,
       WLI_WORKLIST_RPT_TASKINFO info
  where create.EVENT_TYPE = 'TaskCreate' and
        complete.EVENT_TYPE = 'TaskComplete' and
        create.TASK_ID = complete.TASK_ID and
        info.TASK_ID = create.TASK_ID and
        info.TASK_TYPE_ID = 'CustomerTasks:1.0/CustomerRefund:1.0'
        info.OWNER = 'Bob'
  group by info.TASK_ID
```

Example: How many CustomerRefund tasks have been processed this week?

```
select count(TASK_ID)
  from WLI_WORKLIST_RPT_TASKEVENT
  where EVENT_TYPE = 'TaskComplete' and
        TASK_TYPE_ID = 'CustomerTasks:1.0/CustomerRefund:1.0' and
        TIMESTAMP > [long millis for start of week] and
        TIMESTAMP < [long millis for end of week]
  group by EVENT_TYPE
```

Example: Based on the expense reports for this quarter, how much money has been spent on hotel expenses?

Note: This query might use the *Hotel Charges* property of *ExpenseReport* tasks. The task plan designer would have to foresee the need to query on hotel expenses and create a field to hold those expenses apart from other information in the expense report document. Let us assume you want to know your hotel expenses were between the first and last day of the month.

```
select event.TASK_ID, prop.VALUE, event.TIMESTAMP
  from WLI_WORKLIST_RPT_TASKEVENT event, WLI_WORKLIST_RPT_TASKPROPERTY prop
  where event.EVENT_TYPE = 'TaskComplete' and
        event.TASK_TYPE_ID = 'EmployeeTasks:1.0/ExpenseReport:1.0' and
        prop.TASK_ID = event.TASK_ID and
        event.TIMESTAMP > [long millis for the start of month] and
        event.TIMESTAMP < [long millis for the end of month] and
        prop.NAME = 'Hotel Charges'
  order by event.TASK_ID, event.TIMESTAMP
```

Note: It is assumed here that you cannot actually calculate the amount of the hotel expense in each ExpenseReport task. Some database types allow you to convert field values from one data type to another. In databases that support such conversions, you could convert Hotel Charges to a float or currency value and then group and sum the resulting values.

Reporting Event Filtering

Producer-side filtering allows you to exclude certain event types from being reported into the default reporting store. For more information, see "Controlling Events Per-Task Plan" on page 5-6.

A reporting event configuration section is defined within the EventHandler event subscription to indicate if the event types for the subscription should be delivered to the default reporting store. If no such configuration is present for a subscription, no reporting events are delivered as a result of handling that subscription at runtime. If no subscription in any handler for the host application defines that a given event types should be delivered to the default reporting store, then the event is ignored by the default reporting store.

The default Worklist event handler defines *all* event types to be of interest to the default reporting store. If required, this default configuration may be changed using Worklist Console.

Email Notification

Worklist provides a measure of control over business tasks involving human interaction. Tasks are associated with human actors in several ways:

- Task Creator
- Task Owner
- Assignee
- Claimant

These human actors often need to be notified of changes to the tasks they are associated with. The most common example of this is when a task is assigned to a group of human actors. In this case, those actors may not be immediately aware that they have had a task assigned to them. If they regularly have to open Worklist User Portal (or any custom task UI) refresh it regularly, they will see their tasks appear in their Assigned Tasks portlet in the Worklist User Portal. However, if they do not regularly use and refresh their Worklist User Portal, then an alternate method of notification is needed. Most people today have an email client open at all times. Thus, email becomes a powerful tool for notification.

This section provides information about how Worklist support notifications via email.

Worklist allows configuration of email notifications from the task plan definition. This configuration involves choosing the event types that cause an email notification, the actors to whom those notifications should go, and what the body of the email will look like.

Specifying Actors for Email Notification

You can specify actors in terms of their affinity to a task (Creator, Owner, Claimant, Assignee), or by user name (no direct association with the task). In addition, you can use custom email recipient calculators to allow the email address of an actor or actors to be calculated dynamically at runtime. For more information, see “Deploying EMail Recipient Calculators” on page 6-13.

By specifying a task affinity or user name as an email recipient, you implicitly take responsibility for ensuring that these actors have email addresses configured for them in Worklist Console. For more information, see “Setting the E-mail Address of a User” in Worklist Administration in *Using Worklist Console*.

Note: The email notification subsystem logs warnings whenever it cannot find the email address for a configured actor.

Email Recipient Calculators

For cases where the proper recipient for a given type of email notification cannot be known in advance, the email address can be calculated to be used dynamically at runtime.

These calculators allow a user to augment the `To:` list of the e-mail messages that are sent to recipients in response to a Worklist Task Event. For example, there is a Task Plan that defines how to handle the approval of purchase orders. In cases of very expensive purchase orders, perhaps the controller for a business unit should be notified via e-mail when the purchase order is created. The default email notification configuration cannot express this requirement. In this case, the user could implement the `EmailRecipientCalculator` interface, place the implementation in the application that hosts the Task Plan and then have the implementation return the controller's e-mail address any time a CREATE Task Event is received where the `PurchaseOrderAmount` property is over a given amount.

Note: The calculator implementation can be parameterized at deployment time (by a custom module descriptor, `.email` file, in the host application) to allow the amount to be specified, or to pass in some address for an external service that can be used to look up the amount.

```
/**
 * Defines a pluggable implementation of a calculator that can produce the
 * email addresses of actors that are to receive email notifications for
 * a given TaskEvent. This interface is really only intended to provide a
 * simple facility for augmenting the To: list for emails that are already
 * getting sent from the default email subsystem in Worklist. If you need
 * to control the subject, body, or other attributes of emails being sent,
 * you should define your own custom event listener from which you can send
* custom emails.
 * <p>
 * Implementations of this interface MUST provide a public
 * default (no arg) constructor. Implementations must make no assumptions
 * about the lifecycle of their instances nor their relationship to other
 * instances. Lifecycle is controlled entirely via the initialize() and
* destroy() methods.
```

```
 */
public interface EmailRecipientCalculator {

    /**
    * Set any properties configured for this EmailRecipientCalculator in the
     * custom module that deployed it. This method is only called if
properties
     * are available (e.g. specified in the deployment descriptor for the
custom
     * module). Note this method (if called at all) is called prior to the
     * call to the initialize lifecycle method.
     * @param properties
     */
    public void setProperties(Property[] properties);

    /**
    * Initialize any resources needed to start calculating email addresses
in
     * getEmailRecipientsForEvent().
     * @throws com.bea.wli.worklist.api.ManagementException
     */
    public void initialize()
        throws ManagementException;

    /**
    * Destroy this instance and release any resources obtained in initialize
     * or during calls to getEmailRecipientsForEvent().
     * @throws ManagementException
     */
    public void destroy()
        throws ManagementException;

    /**
    * Calculate the email addresses for any actors that should receive email
     * notification for the given event. If no email notifications are to be
     * sent, this method may return null or an empty array.
     * @param event The event for which email notifications are to be sent.
     * @param currentRecipients This is a set of email addresses to which
     *        the email for this event will already be sent. This set is a
    *      copy of the set that will be used to populate the To: list on the
     *        email, so adding/removing items in this set has no effect.
     * @return An array of email addresses for actors that should get email
     *          notifications in response to the event. These actors will
     *          receive an email with a pre-determined subject and body. The
    *        body will contain a URL to the task that caused this event. The
     *          URL may optionally be calculated by installing a
     *          TaskURLCalculator.
     * @throws ManagementException If any error occurs calculating
recipients.
     *          This error will be logged to the server log, but otherwise
     *          will not alter the handling of this event.
     * @see TaskURLCalculator
     */
    public String[] getEmailRecipientsForEvent(TaskEvent event,
                                            Set<String> currentRecipients)
        throws ManagementException;
```

```
}
```

To install an email recipient calculator implementation that implements the above interface, define a custom module in your host application that defines the class name of the implementation and any properties needed to initialize that instance before it can calculate recipients. For more information, see "Deploying EMail Recipient Calculators" on page 6-13.

EMail Delivery Mechanism

E-mail is sent using the JavaMail API. The mail session that is to be used for sending the e-mails is indicated in the Worklist System Instance configuration in Worklist console. This configuration is the name of a mail session configured in WebLogic Server (via the WebLogic Server Administration Console). For more information, see "Changing the Worklist System Instance Configuration" in Worklist Administration in *Using Worklist Console*, Create a Mail Session in *The WebLogic Server Administration Console help* and Programming JavaMail with WebLogic Server in *Developing Applications with WebLogic Server*.

Make sure that the JavaMail session must define the name of the SMTP host to which Worklist will deliver emails. This is done by specifying a session property of the name:

```
mail.smtp.host=<SMTP Host name or IP Address>
```

For example, a Microsoft Exchange server or ISP SMTP server can be specified here.

Note: The email notification subsystem logs errors when a configured JavaMail session cannot be found at runtime.

Event Types

Worklist can deliver a number of different event types in response to changes on a task. Worklist defines the most common events of interest to human actors to be:

- CREATE
- ASSIGN
- CLAIM
- RETURN
- SUSPEND
- RESUME

- SET_ERROR
- CLEAR_ERROR
- COMPLETE
- ABORT
- REACTIVATE
- STEP_EXPIRE
- TASK_EXPIRE

If no specific event type configuration is given for a task plan, the above list is assumed. Task plan designers and Worklist administrators are free to modify this list as required by their specific business cases.

By default, each of the above event types are configured to be sent to a specific set of human actors associated with the task that was changed. The following table describes the default e-mail recipients for each event type and the email template that is used to create the subject and body of the e-mail. Again, task plan designers and Worklist administrators are free to modify this configuration if needed.

By default, email templates are localized to the default locale of the server (not the e-mail client). Event notifications can be configured to be delivered in other locales, if required.

Note: This default configuration specifies a JavaMail session with a JNDI name of `WorklistMailSession`. You must define a JavaMail session of this name in order to utilize this default configuration. For more information, see Create a Mail Session in *The WebLogic Server Administration Console help*

Table 5-1 Text templates of Subject and Body of E-mail Messages

Event Type	Default Recipients	Subject Template	Body Template

CREATE	Creator, Owner	Task '{1}' has been created	Task '{1}' has been created by {2} and is owned by {3}. Task comment is:
			{4}
			To see details for this task, go to this URL:
			{0}

ASSIGN	Assignees	Task '{1}' has been assigned to you	Task '{1}' has been assigned to you. To see details for this task, go to this URL: {0}
CLAIM	Owner, Claimant	Task '{1}' has been claimed	Task '{1}' has been claimed by {2}. To see details for this task, go to this URL: {0}

RETURN	Owner	Task '{1}' has been returned	Task '{1}' has been returned by {2}. The reason given was:
			{3}
			To see details for this task, go to this URL:
			{0}

TAKE_ACTION		Task '{1}' has had action {2} taken on it	Task '{1}' has had action {2} taken on it from step {3} and is now in step {4}. To see details for this task, go to this URL: {0}

Table 5-1 Text templates of Subject and Body of E-mail Messages

Event Type	Default Recipients	Subject Template	Body Template
STEP_CHANGE		Task '{1}' has moved from step {2} to step {3}	Task '{1}' has moved from step {2} to step {3}. To see details for this task, go to this URL: {0}
SET_USER_PROPERTY		Task '{1}' has had property {2} set on it	Task '{1}' has had property {2} set on it. To see details for this task, go to this URL: {0}
PROPERTY_SET_LOCKSET		Task '{1}' has had property set {2}'s lock state changed	Task '{1}' has had property set {2}'s lock state changed from {3} to {4}. To see details for this task, go to this URL: {0}

SUSPEND	Creator, Owner, Claimant	Task '{1}' has been suspended	Task '{1}' has been suspended. The reason given was:
			{2}
			To see details for this task, go to this URL:
			{0}

RESUME	Creator, Owner, Claimant	Task '{1}' has been resumed	Task '{1}' has been resumed. The reason given was:
			{2}
			To see details for this task, go to this URL:
			{0}

Table 5-1 Text templates of Subject and Body of E-mail Messages

Event Type	Default Recipients	Subject Template	Body Template

SET_ERROR	Creator, Owner, Claimant	Task '{1}' has been set into an error state	Task '{1}' has been set into an error state. The reason given was:
			{2}
			To see details for this task, go to this URL:
			{0}

CLEAR_ERROR	Creator, Owner, Claimant	Task '{1}' has had its error state cleared	Task '{1}' has had its error state cleared. The reason given was:
			{2}
			To see details for this task, go to this URL:
			{0}

COMPLETE	Creator, Owner, Claimant	Task '{1}' has been completed	Task '{1}' has completed. The reason given was:
			{2}
			To see details for this task, go to this URL:
			{0}

ABORT	Creator, Owner, Claimant	Task '{1}' has been aborted	Task '{1}' has been aborted. The reason given was:
			{2}
			To see details for this task, go to this URL:
			{0}

Table 5-1 Text templates of Subject and Body of E-mail Messages

Event Type	Default Recipients	Subject Template	Body Template

REACTIVATE	Creator, Owner, Claimant	Task '{1}' has been reactivated	Task '{1}' has been reactivated. The reason given was:
			{2}
			To see details for this task, go to this URL:
			{0}

SET_CURRENT_STEP	Creator, Owner, Claimant	Task '{1}' has had its current step set	Task '{1}' has had its current step set. The reason given was:
			{2}
			The old step was {3} and the new step is {4}. To see details for this task, go to this URL:
			{0}

SET_BUILTIN_PROPERTY		Task '{1}' has had system property {2} set on it	Task '{1}' has had system property {2} set on it. To see details for this task, go to this URL: {0}
STEP_EXPIRE	Creator, Owner, Claimant	Task '{1}' at step {2} has past its completion due date	Task '{1}' at step {2} has past its completion due date of {3,time,full} on {3,date,full}. To see details for this task, go to this URL: {0}
TASK_EXPIRE	Creator, Owner, Claimant	Task '{1}' has past its completion due date	Task '{1}' has past its completion due date of {2,time,full} on {2,date,full}. To see details for this task, go to this URL: {0}

Calculating Task URLs

Each email template (body template) shown in the previous table has a marker for a URL that may be used to navigate directly to the task UI of the task. The `TaskURLCalculator` interface is available for this purpose.

```
public interface TaskURLCalculator {

    public URL getURLForTask(String taskId)

        throws ManagementException;

}
```

A custom module that allows individual Worklist host applications to specify their own custom URL calculator. This is necessary if the Worklist host application also provides a custom Worklist or Task UI that requires a specific or custom URL for navigation to the task. Only one TaskURLCalculator can be registered at any given time.

Worklist provides a default `TaskURLCalculator` implementation that calculates URLs appropriate for use with the default Worklist User Portal.

A new method is available in WorklistTaskUser in the Worklist API to calculate the URL for a task (using the registered or default calculator)

```
public interface WorklistTaskUser {

    …

    public URL getURLForTask(String taskId)

        throws ManagementException, RemoteException;

    …

}
```

Email Templates

Table 5-1 lists the text templates that are used for as the subject and body of the e-mail message. Information from the underlying Worklist task event is substituted into the templates to generate the full text that goes into the e-mail message.

These templates are defined in the WorklistEMail text formatter file and are localized into any language supported by WebLogic Integration. If required, you can customize these template by directly editing the `WorklistEmailTextFormatter.properties` file located in the `worklist.jar` file.

Email Notification Configuration and EventHandler Modules

The configuration for email notification is defined in a listener-type-specific section of an EventHandler module. This module is also used to define configuration for reporting events and other types of events. For more information, see "EventHandler Modules" on page 5-15.

The email configuration of an event handler maps task event types to the actors who should receive e-mail notification when events of a given type occur. Mapping event types to actors for any task plan augments (not replaces) the default event notification configuration for that task plan. In other words, the email configuration given in the configuration module is merged with the default configuration for a given task plan.

MessageBroker and JPD Integration

Worklist may be used directly from a Worklist Portal, or via Worklist API. One of the primary interfaces to Worklist in this API is the Worklist controls. Worklist controls are primarily hosted within WebLogic Integration Process Definitions (JPD). Controls are sufficient to ensure a robust integration in cases where the JPD instantiates or knows the task IDs of the tasks it will work with. However, sometime the JPD does not know the task IDs it will work with, and in some cases, a user may want the creation or modification of a task to cause the creation of a JPD instance.

In such scenarios, JPD MessageBroker is used to deliver messages to a JPD from event channels. These channels can have a semantic attached to them and allow message filters to control delivery of these messages to consumers. This is an ideal mechanism to allow existing JPD instances to become aware of task instances as needed (by responding to events of a given type from a given task plan for instance). This mechanism, via static subscriptions, also allows a JPD instance to be created in response to a task event.

Worklist Channel

Worklist delivers events to a MessageBroker channel when changes occur to tasks of any type using the following Worklist channel:

```
/WorklistEvent
```

So, a JPD may define a MessageBroker static or dynamic subscription using the `/WorklistEvent` channel name (and any relevant header fields in a filter) to receive Worklist events it is interested in. This channel is a `rawdata` channel and has messages with a body that is a serialized TaskEvent instance. The actual runtime type of the TaskEvent instance depends on the event type for the event. For more information, see "Task Change Events" on page 5-1.

You need to deserialize the bytes of the MessageBroker message using `ObjectInputStream.readObject()`, and cast the resulting object to a TaskEvent. An example JPD is given below:

```
@com.bea.wli.jpd.Process(
      process="<process name=\"WorklistControlStatefulControlReceive\">\n"
+
                "  <clientRequest name=\"Client Request\"
method=\"clientRequest\"/>\n" +
                "</process>"
)
public class HandleTaskEvent implements com.bea.jpd.ProcessDefinition
{
    public String _x0;


    static final long serialVersionUID = 1L;


    @MessageBroker.StaticSubscription(channelName = "/WorklistEvent",
                                      messageMetaData="{meta}",
                                      messageBody = "{body}")
    public void clientRequest(TaskEventMetadataDocument meta,
                              Object body)
        throws IOException, ClassNotFoundException {
        RawData rawData = (RawData)body;
        ByteArrayInputStream bais = new
ByteArrayInputStream(rawData.byteValue());
        ObjectInputStream ois = new ObjectInputStream(bais);
        TaskEvent event = (TaskEvent)ois.readObject();
       System.out.println("## Got TaskEvent in JPD: " + event.getSummary());
        this._x0 = event.getSummary();
    }
}
```

Event Types

Events are delivered to the Worklist MessageBroker channel as they occur in the Worklist engine. The MessageBroker listener is capable of sending any defined task event to MessageBroker. By default, only the following events are delivered via MessageBroker

- CREATE
- STEP_CHANGE
- SET_CURRENT_STEP
- COMPLETE
- ABORT
- SET_ERROR/CLEAR_ERROR
- REACTIVATE
- STEP_EXPIRE
- TASK_EXPIRE

Event delivery may be customized on a per-task plan or global basis. For more information, see "MessageBroker Event Configuration and EventHandler Modules" on page 5-42.

Message Header Fields

Each task event posted to MessageBroker is posted to a single Worklist channel. This message has a number of header fields to allow filtering of messages at the point of consumption in the JPD. The various message header fields are:

- TaskPlanId - The ID of the TaskPlan for the task upon which the event occurred
- EventType - The type of event that occurred
- TaskId - The ID of the task upon which the event occurred
- Timestamp - The date and time the event occurred
- Claimant - The name of the claimant for the task

This message header fields are delivered as an XMLBean event metadata object of the type

`com.bea.wli.worklist.xml.TaskEventMetadataDocument`

as defined in the `Worklist.xsd` schema

Note: This schema is available from BEA Workshop for WebLogic IDE by creating a Worklist application with a utility project and adding system schemas to that project. The XMLBean is available in the `worklist-client.jar` public jar file).

```
Example Process:


package process;


import java.util.Date;

import com.bea.jpd.ProcessDefinition;

import com.bea.jpd.JpdContext;

import org.apache.beehive.controls.api.bean.Control;

import java.io.ByteArrayInputStream;

import java.io.ObjectInputStream;


@com.bea.wli.common.XQuery(prolog="declare namespace ns0 = 
\"http://www.bea.com/wli/worklist/xml\";")

@com.bea.wli.jpd.Process(process = 

"<process name=\"StaticMBWithNoXQuery\">" + 

"  <clientRequest name=\"Subscription\" method=\"subscription\"/>" + 

"  <perform name=\"Print Event Info\" method=\"perform\"/>" + 

"</process>")

public class StaticMBWithAndFilterValue implements ProcessDefinition {


    public com.bea.wli.worklist.xml.TaskEventMetadataDocument _e1;

    public com.bea.data.RawData _rd;
    @com.bea.wli.jpd.Context
    JpdContext context;

    static final long serialVersionUID = 1;

    @com.bea.wli.control.broker.MessageBroker.StaticSubscription(xquery =
"fn:concat($metadata/ns0:taskPlanId,  $metadata/ns0:eventType)",
filterValueMatch = "/Loans/Loan Approval:1.0:LoanAppCREATE", channelName =
"/WorklistEvent", messageBody = "{x0}", messageMetaData = "{x1}")
    public void subscription(com.bea.data.RawData x0,
                             com.bea.wli.worklist.xml.TaskEventMetadataDocument
x1) {
        // #START: CODE GENERATED - PROTECTED SECTION - you can safely add
code above this comment in this method. #//
        // input transform
        // parameter assignment
        this._rd = x0;
        this._e1 = x1;
        // #END  : CODE GENERATED - PROTECTED SECTION - you can safely add
code below this comment in this method. #//
    }

    public void perform() throws Exception {

        Date dt = new Date();
        StringBuffer sb = new StringBuffer();
        sb.append("\n\n[" + dt.getTime() + "]\t ++++++++
StaticMBWithAndFilterValue : GOT MB Event in JPD with Filter Value Match
+++++ " + new Date());
```

```
        sb.append("[" + dt.getTime() + "]\t Event Info: " + this._e1);
       sb.append("[" + dt.getTime() + "]\t Is Raw Data NULL? " + (this._rd
== null ? true : false));
        sb.append("[" + dt.getTime() + "]\t
---------------------------------------------------------------------------
--");

        try {
            ByteArrayInputStream baos =
                new ByteArrayInputStream(_rd.byteValue());
            ObjectInputStream ois = new ObjectInputStream(baos);
            TaskEvent event = (TaskEvent)ois.readObject();
            sb.append("[" + dt.getTime() + "]\t Event object summary: " +
event.getSummary());
        } catch (Exception e) {
            sb.append("[" + dt.getTime() + "]\t Event object (error): " +
e.toString());
        }

        System.out.println(sb);
    }
}
```

MessageBroker Event Configuration and EventHandler Modules

Task events are delivered to MessageBroker via an EventHandler. By default, Worklist defines a single EventHandler with a default event configuration. For more information, see "Event Types" on page 5-30. However, Worklist administrators can modify this configuration globally or for specific task plans using Worklist Console. For more information, see "Changing the Event Handler Details" in Worklist Administration.

The configuration for MessageBroker event delivery is defined as part of an EventHandler module. This module is also used to define configuration for email notification, reporting events and other types of events. For more information, see "EventHandler Modules" on page 5-15.

The MessageBroker event configuration of an event handler indicates task event types that are to be sent to MessageBroker. Indicating an event type within the MessageBroker event configuration *enables* that event for delivery via MessageBroker. Enabling event types for any task plan augments (not replaces) the default event notification configuration for that task plan. In other words, the configuration given in the configuration module is merged with the default configuration for a given task plan.

Custom Task Event Listeners

Worklist allows you to listen to runtime events (see "Task Change Events" on page 5-1) and take any action in response to those events. Your listener can request synchronous or asynchronous event delivery and can request to be registered as a critical or non-critical listener. For more information, see "Event Dispatch Modes (Quality of Service)" on page 5-6.

Your custom task event listener implements the `TaskEventListener` interface (see "Task Change Events" on page 5-1). You can then register your listener by including a `.listener` file in your application. For more information, see "Deploying Custom Event Listeners" on page 6-9.

Custom Assignment Handlers

You can assign users to specific tasks using Worklist. These default assignment facilities should be sufficient for a large number of assignment scenarios. However, in some cases, the default assignment facilities in Worklist may not be enough. In these cases, you can provide a custom assignment handler that allows you to control the assignment process for your tasks.

Custom assignment handlers take full control of the assignment process. They may be appropriate in cases where special load balancing or availability checking logic is required. A custom assignment handler can be applied to specific task plans or globally to all task plans.

Note: Only one global assignment handler can be present at any given time, so this option should be used carefully. Applying a global assignment handler overrides any previously applied handler (including the default Worklist assignment handler).

A custom assignment handler must implement the following interface:

```
com.bea.wli.worklist.api.config.AssignmentHandler
```

and *must* provide a public default (no arg) constructor. Custom assignment handlers must not expect to control how many instances of the handler that are instantiated nor the lifecycle of any given instance. The lifecycle of a given handler instance is controlled via the `initialize()` and `destroy()` lifecycle methods.

A summary of this interface is as follows. For more information about the complete interface, see the Worklist API JavaDoc available at
`http://edocs/wli/docs92/worklist.javadoc/index.html`

```
/**
 * Describes an assignment algorithm that can be used to assign a task
 * of a given type to users as candidates for claiming the task. This
algorithm
 * may optionally choose to claim the task in the name of a designated
 * candidate user. Implementations of this interface MUST provide a public
 * default (no arg) constructor. Implementations must make no assumptions
about
 * the lifecycle of their instances nor their relationship to other
instances.
 * Lifecycle is controlled entirely via the initialize() and destroy()
methods.
 */
public interface AssignmentHandler {

    /**
     * Request object passed to an assignment handler when Worklist is
     * requesting the assignment of a task.
     */
    public class Request {

        private String _taskId;
        private String[] _candidateUserIds;
        private CandidateListHandling _handling;
        private boolean _availabilityCheckEnabled;
        private WorkloadRequest _workloadRequest;

        …
    }

    /**
     * Response object an assignment handler returns describing the 
assignment
     * decision that has been made for the task.
     */
    public interface Response {

        /**
         * Assignee list as calculated by the assignment handler, or
         * null to accept the assignee list that was used to generate the
         * candidate user id list.
         */
        AssigneeDefinition[] getNewAssigneeList();

        /**
       * The user id of the user who should be made the claimant of the task,
         * or null if no claimant was chosen.
         */
        String getClaimant();
    }
```

```
/**
 * Set any properties that were configured with this assignment
 * handler. Note that the value should be obtained from the default value
 * of the provided properties.
 * @param props
 */
public void setProperties(Property[] props);

/**
 * Initialize this instance in preparation for calls on assignTask()
 * @throws ManagementException if any error occurs that would prevent
 *         this instance from properly handling calls to assignTask()
 */
public void initialize()
    throws ManagementException;

/**
 * Make assignment decisions for the given task. The given list of
 * candidate claimant user names may be used by this method, or ignored.
 * The candidate user name list is derived from the assignee list for the
 * current step of the task, or the assignee list passed to
 * WorklistTaskAdmin.assignTask(). This method MUST NOT actually modify
 * the task to reflect the claimant it chooses (if any) or cause any
 * other side effects to the task. Assignment decisions are communicated
 * back to the caller via the returned Response instance.
 * @param request A Request instance containing the information needed to
 *        assign a task.
```

```
     * @return A Response instance indicating any newly calculated assignee
     *        list to be used for this task, and any claimant chosen for the
     *         task.
     * @throws AssignmentException if any authorization or logical error
happens
     *         during assignment.
     * @throws ManagementException If any other error occurs during
assignment.
     */
    public Response
    assignTask(Request request)
        throws AssignmentException, ManagementException;


    …


    /**
     * Release any resources obtained in the call to initialize or calls to
     * assignTask().
     * @throws ManagementException If any error occurs releasing resources.
     */
    public void destroy()
        throws ManagementException;
}
```

The assignment handler implementation is registered by deploying an assignment handler configuration module within the Worklist host application. For more information, see "Deploying Custom Assignment Handlers" on page 6-6.

At runtime, a registered assignment handler is invoked via its `assignTask` method. It receives a candidate list, the ID of the task that is being assigned and parameters that are intended as hints to guide the execution of the custom assignment handler. The `handling` and enableLoadBalancingAvailabilityCheck parameters may be used by the assignment handler, but this is not required.

The assignment handler is free to make use of any information in the task (including its type, properties, and so on.) in making assignment and claim decisions. Any decision made by the handler is communicated back to the caller of the assignTask() method via the returned Response instance.

The Response object can define a newly calculated assignee list (null indicates the old list should still be used) and any claimant for the task (null means no claimant was chosen). The assignment handler *must not* attempt to force the claiming of the task (for example, by calling WorklistTaskUser.claimTask()) or call any method in the Worklist API that causes side effects on the task.

The assignment handler's assignTask() method is used by Worklist to handle applying any updated assignee list or claimant information to the task after the return.

Note: An assignment handler configuration module can specify configuration properties that may be used by the assignment handler at runtime. Such properties may parameterize and control the behavior of the handler at runtime. Any configuration properties specified for the assignment handler are provided to the assignment handler implementation at the time the handler instance is created via a call to the setProperties() method. For more information, see "Deploying Custom Assignment Handlers" on page 6-6.

CHAPTER 6

Extending the Capabilities of your Application with Custom Modules

Custom modules in your application can be used to add extra custom metadata and services into your application. You may add any of the following services to your application:

- Task Plans (`.task` file) to register a new task plan. These task plans may refer to specific schema file. In such a scenario, those schema files must also be deployed and present in the application.
- Event Handler (`.handler` file) to configure default Worklist listeners (reporting, email, MessageBroker, control).
- Assignment Handler (`.assign` file) to configure and register custom assignment handlers.
- Task Event Listener (`.listener` file) to configure and register custom task event listeners.
- EmailRecipientCalculator (`.email` file) to calculate the e-mail address of the recipients of an e-mail notification and send it dynamically at runtime.
- TaskURLCalculator (`.taskurl` file) to calculate the URL required to navigate directly to the task UI for the task.
- TaskHistoryProvider (`.history` file) to retrieve task history of the task associated with this provider.

Deploying Task Plans

Any application can have task plans installed into it to become a task plan host application. Task plans are defined as `.task` files within a task plan host application. However, in order to get these task plans registered into a task plan registry so that they can be used, you must first install them into a task plan host application, and deploy that task plan host application to a WebLogic Server instance.

The Workshop for WebLogic plugin for Worklist handles installing `.task` files in your application for you. So, when you create a task plan in Workshop for WebLogic IDE, it is automatically installed in your host application. However, if you are not using Workshop for WebLogic, Worklist provides a command-line and Ant task for installing task plans into a host application. Both these utilities require the following arguments:

- **Application Root Directory** - The path to the root of the application that will receive the installed task plans.
- **Task Plan Source Directory** - The path to a directory containing the `.task` files to be deployed.

  This directory may exist either inside or outside the application directory. If it is outside, the `.task` files are copied to a location within the application directory that matches the location of the source `.task` file relative to the specified task plan source directory. If the source directory is inside the application directory, no file copying is done.

The command-line and Ant utilities scan the source directory for `.task` files and then register all the `.task` files found for deployment with the task plan host application.

Using the command-line utility to install task plans into a host application:

```
java com.bea.wli.worklist.build.InstallTaskPlan <options>
```

The options and arguments to this command are defined as follows:

`-appRootDir` - Holds the `META-INF` directory for the target application.

`-taskTypeSrcDir` - Holds the `.task` files to be installed into the target application. If this directory is a child of appRootDir, no file copying is done. If not, the `.task` files found under this directory (recursively) are copied into the same relative path within the target application at the root of the application.

Using Ant utility to install task plans into a host application:

```
<project name="MyProject">

  ...

  <taskdef name="install-tasktype-in-app"
    classname="com.bea.wli.worklist.build.InstallTaskPlanTask"

      classpath="${<Classpath that includes WebLogic Integration's
public.jar}"/>
```

```
  <install-tasktype-in-app
    appRootDir="${<Host application's root directory>}"
    taskTypeSrcDir="${<Source directory of your TaskPlan>}"/>
  ...
</project>
```

Deploying Schemas in your Application

You need to deploy schemas in your application based on requirement or to support Task Plans that are associated with specific schemas.

Task plans installed within your application can refer to XMLBean data types. These data type references can define custom variants that are specific XMLBean types compiled from schemas you provide. Those schemas must be provided in the same application (or at least be visible from the application) that hosts the task plans. The simplest and recommended approach for this is to create a Schemas project at the root of your application, and include the schema XSD files under a `src` directory in that project. The build facility within the Workshop for WebLogic IDE can be used to compile those schemas to XMLBeans.

Deploying Event Handlers

You can define your own event handlers to control the operation of the Worklist-defined listeners of these types:

- Email Notification
- Reporting
- MessageBroker

Your event handler can be associated with one or more task plans, and/or event types, or it can be applied globally. If the handler is associated with specific task plans, the configuration it contains will only be used for the associated task plans or event types. If an event handler is applied globally, it may be used by any task and any type of event.

To use an event handler, you must install the event handler into the Worklist host application. To install your event handler:

- Create an EventHandler module descriptor (`.handler`) file in your Worklist host application

Note: The schema of this descriptor is available in the `worklist-config-binding.jar` file as `schema-0.xsd`. For a sample of this descriptor file, see "Sample of an Event Handler Descriptor File" on page 6-5.

- Run the `InstallEventHandler` utility to install your event handler into the Worklist host application.

The Workshop for WebLogic plugin for Worklist automatically installs the event handler when you create the `.handler` file, so you need not do this manually when you use Workshop for WebLogic. However, if you are not using Workshop for WebLogic, Worklist provides a command-line and Ant task for installing event handlers into a Worklist host application. Both these utilities require the following arguments:

- **Application Root Directory** - The path to the root of the application that will receive the installed event listener.
- **Handler Source Directory** - The path to a directory containing the `.handler` files to be installed.

  This directory may exist either inside or outside the application directory. If it is outside, the `.handler` files is copied to a location within the application directory that matches the location of the source `.handler` file relative to the specified source directory. If the source directory is inside the application directory, no file copying is done.

The command-line and Ant utilities scan the source directory for `.handler` files, and then register all the files found for deployment with the host application.

Using the command-line utility to install event handlers into a host application:

```
java com.bea.wli.worklist.build.InstallEventHandler <options>
```

The options to this command are defined as follows:

`-appRootDir` - Holds the `META-INF` directory for the target application.

`-handlerSrcDir` - Holds the `.handler` files to be installed into the target application. If this directory is a child of appRootDir, no file copying is done. If not, the `.handler` files found under this directory (recursively) are copied into the same relative path within the target application at the root of the application.

Using the Ant utility to install event handlers into a host application:

```
<project name="MyProject">
  ...
  <taskdef name="install-eventhandler-in-app"
    classname="com.bea.wli.worklist.build.InstallEventHandlerTask"

     classpath="${<Classpath that includes WebLogic Integration's
public.jar}"/>

  <install-eventhandler-in-app

    appRootDir="${<Host application's root directory>}"

    handlerSrcDir="${<Source directory of your event handler>}"/>

  ...

</project>
```

Sample of an Event Handler Descriptor File

Create an Event Handler descriptor file in your application that looks something like this:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<java:event-handler xmlns:java="java:com.bea.wli.worklist.config"
                    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <java:global-handler>false</java:global-handler>

  <java:handled-task-type-id>/MyFolder/MyTaskPlan:1.0</java:handled-task-t
ype-id>

  <java:name>My Handler</java:name>


  <java:event-subscription>

    <java:name>CREATE/ASSIGN Subs</java:name>

    <java:event-type><java:type>CREATE</java:type></java:event-type>

    <java:event-type><java:type>ASSIGN</java:type></java:event-type>


    <java:email-event-subscription>

    <java:java-mail-session-jndi-name>MyMailSession</java:java-mail-sessio
n-jndi-name>

    <java:from-name>Me@bea.com</java:from-name>
```

```
          <java:actor>
             <java:name>me</java:name>
             <java:type>User</java:type>
          </java:actor>
          </java:email-event-subscription>
        </java:event-subscription>

      </java:event-handler>
```

Deploying Custom Assignment Handlers

You can define your own custom assignment handlers by implementing the following Java interface:

`com.bea.wli.worklist.api.config.AssignmentHandler`

Your custom assignment handler can be associated with one or more task plans or can be applied globally. If the handler is associated with specific task plans, it is called whenever a task of that task plan is assigned to users. If an assignment handler is applied globally, it *replaces* any previously applied handler (including the Worklist default handler) within the Worklist system instance that receives the deployment of the handler (usually the local system instance in a stand-alone Worklist application scenario). The new global handler is called when any task is assigned to users.

Note: Applying an assignment handler globally should be done with caution as you may overwrite a previously installed custom handler.

To use of a custom assignment handler, you need to install it into the Worklist host application. To install your custom assignment handler:

- Create a CustomAssignmentHandler module descriptor (`.assign` file) in your Worklist host application

  Note: The schema of this descriptor is available in the `worklist-config-binding.jar` file as `schema-0.xsd`. For a sample of this descriptor file, see “Sample of an Assignment Handler Descriptor File” on page 6-8.

- Make sure that the implementation class of your handler and the classes it depends on are in the `APP-INF\lib` directory of the Worklist host application as `.jar` files or in the `APP-INF\classes` directory as individual class files.
- Run the `InstallAssignmentHandler` utility to install your custom assignment handler into the Worklist host application.

The Workshop for WebLogic plugin for Worklist automatically installs the assignment handler when you create the `.assign` file, so you need not do this manually when you use Workshop for WebLogic. However, if you are not using Workshop for WebLogic, Worklist provides a command-line and Ant task for installing assignment handlers into a Worklist host application. Both these utilities require the following arguments:

- **Application Root Directory** - The path to the root of the application that will receive the installed assignment handler.
- **Handler Source Directory** - The path to a directory that contains the `.assign` files to be installed.

  This directory may exist either inside or outside the application directory. If it is outside, the `.assign` files are copied to a location within the application directory that matches the location of the source `.assign` file relative to the specified source directory. If the source directory is inside the application directory, no file copying is done.

These utilities scan the source directory for `.assign` files and then register all the files found for deployment with the host application.

Using the command-line utility to install assignment handlers into a host application:

```
java com.bea.wli.worklist.build.InstallAssignmentHandler <options>
```

The options to this command are defined as follows:

`-appRootDir` - Holds the `META-INF` directory for the target application.

`-handlerSrcDir` - Holds the `.assign` files to be installed into the target application. If this directory is a child of appRootDir, no file copying is done. If not, the `.assign` files found under this directory (recursively) are copied into the same relative path within the target application at the root of the application.

Using the Ant utility to install assignment handlers into a host application:

```
<project name="MyProject">

  ...
  <taskdef name="install-assignmenthandler-in-app"
   classname="com.bea.wli.worklist.build.InstallAssignmentHandlerTask"

    classpath="${<Some Classpath that includes WebLogic Integration's
public.jar}"/>

   <install-assignmenthandler-in-app

    appRootDir="${<Host Application's root dir>}"

    handlerSrcDir="${<Source directory of your handler>}"/>

  ...

</project>
```

Sample of an Assignment Handler Descriptor File

Create an assignment handler descriptor file in your application that looks something like this:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<custom-assignment-handler-set

                   xmlns ="java:com.bea.wli.worklist.config"

                   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <custom-assignment-handler>

    <global-handler>false</global-handler>

    <!-- One or more handled-task-type-id elements if this is not a global
handler -->

    <handled-task-type-id>/MyFolder/MyTaskPlan:1.0</handled-task-type-id>

    <handled-task-type-id>/MyFolder/MyOtherTaskPlan:2.0</handled-task-type
-id>

    <implementation-class-name>com.acme.mystuff.MyAssignmentHandler

    </implementation-class-name>

    <!-- Any properties you might need to properly construct your handler.
Only string values are supported. These will be passed to the setProperties
method of your AssignmentHandler impl -->

    <property>
```

```
        <name>ConstructorProp1</name>
        <value>Value for prop 1</value>
      </property>
    </custom-assignment-handler>
  </custom-assignment-handler-set>
```

Deploying Custom Event Listeners

You can define your own custom EventListeners by implementing the following Java interface:

`com.bea.wli.worklist.api.events.TaskEventListener`

Your custom listener may be associated with one or more task plans, event types, or can be applied globally. If the listener is associated with specific task plans, or event types, it is called when a task of that type is modified in a way that creates an event of the desired type. If the listener is applied globally, it is called when any event occurs on any type of task.

To use a task event listener, you must install it into the Worklist host application as follows:

- Create a TaskEventListener module descriptor (`.listener`) file in your Worklist host application.

  Note: The schema for this descriptor is available in the `worklist-config-binding.jar` file as `schema-0.xsd`. For a sample of this descriptor file, see "Sample of an Event Listener Descriptor File" on page 6-10.

- Make sure that the implementation class of your listener and the classes it depends on are in the `APP-INF\lib` directory of the Worklist host application as `.jar` files or in the `APP-INF\classes` directory as individual class files.
- Run the `InstallTaskEventListener` utility to install your custom event listener into the Worklist host application.

The Workshop for WebLogic plugin for Worklist automatically installs the custom event listener when you create the `.listener` file, so you need not do this manually when you use Workshop for WebLogic. However, if you are not using Workshop for WebLogic, Worklist provides a command-line and Ant task for installing custom event listeners into a Worklist host application. Both these utilities require the following arguments:

- **Application Root Directory** - The path to the root of the application that will receive the installed event listener.

- **Listener Source Directory** - The path to a directory containing the `.listener` files to be installed. This folder may exist either inside or outside the application directory. If it is outside, the `.listener` files will be copied to a location within the application directory that matches the location of the source `.listener` file relative to the specified source directory. If the source directory is inside the application directory, no file copying is done.

These utilities scan the source directory for `.listener` files and then register all the files found for deployment with the host application.

Using the command-line utility to install event listeners into a host application:

```
java com.bea.wli.worklist.build.InstallTaskEventListener <options>
```

The options to this command are defined as follows:

`-appRootDir` - Holds the `META-INF` directory for the target application.

`-listenerSrcDir` - Holds the `.listener` files to be installed into the target application. If this directory is a child of appRootDir, no file copying is done. If not, the `.listener` files found under this directory (recursively) are copied into the same relative path within the target application at the root of the application.

Using the Ant utility to install event listeners into a host application:

```
<project name="MyProject">

  ...

    <taskdef name="install-taskeventlistener-in-app"
       classname="com.bea.wli.worklist.build.InstallTaskEventListenerTask"

       classpath="${<Some Classpath that includes WebLogic Integration's
public.jar}"/>

    <install-taskeventlistener-in-app

    appRootDir="${<Host Applications's root dir>}"

    listenerSrcDir="${<Source directory of your listener>}"/>

  ...

</project>
```

Sample of an Event Listener Descriptor File

Create a task event listener descriptor (`.listener`) file in your application that looks something like this:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<task-event-listener
    xmlns ="java:com.bea.wli.worklist.config"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <!-- Unique name in the app -->
  <name>MyTaskEventListener</name>

  <!-- If global, task-plan-id elements are ignored -->
  <global-listener>false</global-listener>

  <!-- One or more task-plan-id elements if this is not a global
       listener -->
  <task-plan-id>/MyFolder/MyTaskPlan:1.0</task-plan-id>
  <task-plan-id>/MyFolder/MyOtherTaskPlan:1.0</task-plan-id>

  <!-- If true, event-type elements are ignored -->
  <for-all-event-types>false</for-all-event-types>

  <!-- One or more event types if this is not applied for all event types
       -->
  <event-type>CREATE</event-type>
  <event-type>COMPLETE</event-type>
  <event-type>ABORT</event-type>

  <!-- Dispatch Mode (careful, sync listeners are dispatched in the same
       thread in which the task update occurs, and thus runs as the user
       that caused the update. Async listeners run as the run-as principal
      for the TaskEventDispatcherMDB (anonymous by default, but configurable
       in the WLS console) -->
  <sync>true</sync>

  <!-- Criticality -->
  <critical>true</critical>

  <!-- TaskEventListener Impl Class Name (must have public no-arg
constructor) -->
  <implementation-class-name>
    demo.MyTaskEventListener
  </implementation-class-name>

  <!-- Any properties you might need to properly construct your listener.
       Only string values are supported. These will be passed to the
       setProperties method of your TaskEventListener impl -->
  <property>
    <name>Prop1</name>
    <value>Hello</value>
  </property>

</task-event-listener>
```

The above listener descriptor would install an instance of `com.acme.mystuff.MyTaskEventListener` and make it applicable to tasks of type:

- `MyFolder/MyTaskPlan:1.0`
- `MyFolder/MyOtherTaskPlan:2.0`

and events of types:

- `CREATE`
- `COMPLETE`
- `ABORT`

The listener instance is constructed using a no-arg constructor, and then it's `setProperties` method is called with the properties specified in the property elements, followed by it's `initialize` method.

Deploying EMail Recipient Calculators

You can define your own custom email recipient calculator by implementing the following Java interface:

```
com.bea.wli.worklist.api.config.EmailRecipientCalculator
```

The custom e-mail recipient calculator is applied globally and is invoked any time an e-mail notification is sent. When you install an `EmailRecipientCalculator,` any previously applied calculator within the Worklist system instance that receives the deployment of the calculator (usually the local system instance in a stand-alone Worklist application scenario) is replaced. The new global calculator is called when any e-mail notification is sent from the Worklist email subsystem.

To use an email recipient calculator, you must install it into the Worklist host application as follows:

- Create an EmailRecipientCalculator module descriptor (`.email` file) in your Worklist host application

  Note: The schema for this descriptor is available from the `worklist-config-binding.jar` file as `schema-0.xsd`. For a sample of this descriptor file, see “Sample of an Email Recipient Calculator Descriptor File” on page 6-15.

- Make sure that the implementation class of your calculator and the classes it depends on are in the `APP-INF\lib` directory of the Worklist host application as `.jar` files or in the `APP-INF\classes` directory as individual class files.

- Run the `InstallEmailRecipientCalculator` utility to install your calculator into the Worklist host application.

The Workshop for WebLogic plugin for Worklist automatically installs the calculator when you create the `.email` file, so you need not do this manually when you use Workshop for WebLogic. However, if you are not using Workshop for WebLogic, Worklist provides a command-line and

Ant task for installing the email recipient calculators into a Worklist host application. Both these utilities require the following arguments:

- **Application Root Directory** - The path to the root of the application that will receive the installed email recipient calculator.
- **Calculator Source Directory** - The path to a directory containing the `.email` files to be installed. This directory may exist either inside or outside the application directory. If it is outside, the `.email` files are copied to a location within the application directory that matches the location of the source `.email` file relative to the specified source directory. If the source directory is inside the application directory, no file copying is done.

These utilities scan the source directory for `.email` files and then register all the files found for deployment with the host application.

Using the command-line utility to install email recipient calculator into a host application:

```
java com.bea.wli.worklist.build.InstallEmailRecipientCalculator <options>
```

The options to this command are defined as follows:

`-appRootDir` - Holds the `META-INF` directory for the target application.

`-calculatorSrcDir` - Holds the `.email` files to be installed into the target application. If this dirrectory is a child of appRootDir, no file copying is done. If not, the `.email` files found under this directory (recursively) are copied into the same relative path within the target application at the root of the application.

Using the Ant utility to install email recipient calculator into a host application:

```
<project name="MyProject">

  ...

  <taskdef name="install-emailrecipientcalculator-in-app"
classname="com.bea.wli.worklist.build.InstallEmailRecipientCalculatorTask"

         classpath="${<Some Classpath that includes WebLogic Integration's
public.jar}"/>

  <install- emailrecipientcalculator -in-app

      appRootDir="${<Host Application's root directory>}"

      calculatorSrcDir="${<Source directory of your calculator>}"/>

  ...

</project>
```

Sample of an Email Recipient Calculator Descriptor File

Create an email recipient calculator module descriptor (`.email`) file in your application that looks something like this:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<java:email-recipient-calculator

  xmlns:java="java:com.bea.wli.worklist.config"

  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">


<java:implementation-class-name>com.bea.wli.worklist.test.testEmailNotific
ation.EmailRecipientCalculatorImpl</java:implementation-class-name>

  <java:property>

    <java:name>EmailDomainAddr</java:name>

    <java:value>custom.worklist.drt.com</java:value>

  </java:property>

</java:email-recipient-calculator>
```

Deploying Task URL Calculators

You can define your own custom task URL calculator by implementing the following Java interface:

```
com.bea.wli.worklist.api.config.TaskURLCalculator
```

Your calculator is applied globally, and is invoked any time an email notification is sent. When you install a TaskURLCalculator, any previously applied calculator within the Worklist system instance that receives the deployment of the calculator (usually the local system instance in a stand-alone Worklist application scenario) is replaced. The new global calculator is called when any email notification is sent from the Worklist email subsystem.

To use a Task URL Calculator, you must install it into the Worklist host application as follows:

- Create a Task URL Calculator module descriptor (`.taskurl` file) in your Worklist host application.

Note: The schema for this descriptor is available from the `worklist-config-binding.jar` file as `schema-0.xsd`. For a sample of this descriptor file, see "Sample of a Task URL Calculator Descriptor File" on page 6-17.

- Make sure that the implementation class of your calculator and the classes it depends on are in the `APP-INF\lib` directory of the Worklist host application as `.jar` files or in the `APP-INF\classes` directory as individual class files.
- Run the `InstallTaskURLCalculator` utility to install your calculator into the Worklist host application.

The Workshop for WebLogic plugin for Worklist automatically installs the calculator when you create the `.taskurl` file, so you need not do this manually when you use Workshop for WebLogic. However, if you are not using Workshop for WebLogic, Worklist provides a command-line and Ant task for installing the task URL calculators into a Worklist host application. Both these utilities require the following arguments:

- **Application Root Directory** - The path to the root of the application that will receive the installed calculator.
- **Calculator Source Directory** - The path to a directory containing the `.taskurl` files to be installed. This directory may exist either inside or outside the application directory. If it is outside, the `.taskurl` files is copied to a location within the application directory that matches the location of the source `.taskurl` file relative to the specified source directory. If the source directory is inside the application directory, no file copying is done.

These utilities scan the source directory for `.taskurl` files and then register all the files found for deployment with the host application.

Using the command-line utility to install task URL calculators into a host application:

```
java com.bea.wli.worklist.build.InstallTaskURLCalculator <options>
```

The options to this command are defined as follows:

`-appRootDir` - Holds the `META-INF` directory for the target application.

`-calculatorSrcDir` - Holds the `.taskurl` files to be installed into the target application. If this directory is a child of appRootDir, no file copying is done. If not, the `.taskurl` files found under this directory (recursively) are copied into the same relative path within the target application at the root of the application.

Using the Ant utility to install task URL calculators into a host application:

```
<project name="MyProject">
```

```
  ...
  <taskdef name="install-taskurlcalculator-in-app"
classname="com.bea.wli.worklist.build.InstallTaskURLCalculatorTask"
           classpath="${<Classpath that includes WebLogic Integration's
public.jar}"/>
  <install-taskurlcalculator-in-app
      appRootDir="${<Host Application's root directory>}"
      calculatorSrcDir="${<Source directory of your calculator>}"/>
  ...
</project>
```

Sample of a Task URL Calculator Descriptor File

Create a Task URL Calculator module descriptor (`.taskurl`) file in your application that looks something like this:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<java:task-url-calculator
  xmlns:java="java:com.bea.wli.worklist.config"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

<java:implementation-class-name>com.bea.wli.worklist.test.testURLCalculato
rImpl</java:implementation-class-name>
  <java:property>
    <java:name>MyProperty</java:name>
    <java:value>My Custom Value</java:value>
  </java:property>
</java:task-url-calculator>
```

Deploying Custom Task History Providers

You can define your own custom task history providers by implementing the following Java interface:

`com.bea.wli.worklist.api.config.TaskHistoryProvider`

Your custom task history provider can be associated with one or more task plans or can be applied globally. If the provider is associated with specific task plans, it is called whenever task history is requested for a task of that type. If it is applied globally, it replaces any previously applied provider (including the Worklist default provider) within the Worklist system instance that receives the deployment of the provider (usually the local system instance in a stand-alone Worklist application scenario). The new global provider is called when task history is requested for a task of any type.

Note: Applying a task history provider globally should be done with caution as you may overwrite a previously installed custom provider.

To use a custom task history provider, you must install it into the Worklist host application as follows:

- Create a Task History Provider module descriptor (`.history` file) in your Worklist host application

  Note: The schema for this descriptor is available from the `worklist-config-binding.jar` file as `schema-0.xsd.` For a sample of this descriptor file, see "Sample of a Custom Task History Provider Descriptor File" on page 6-19.

- Make sure that the implementation class of your provider and the classes it depends on are in the `APP-INF\lib` directory of the Worklist host application as `.jar` files or in the `APP-INF\classes` directory as individual class files.
- Run the `InstallTaskHistoryProvider` utility to install your provider into the Worklist host application.

Note: The Workshop for WebLogic plugin for Worklist does not automatically install the provider as with other modules. You *must* do this manually to deploy your provider deployed at runtime.

Worklist provides a command-line and Ant task for installing custom providers into a Worklist host application. Both these utilities require the following arguments:

- **Application Root Directory** - The path to the root of the application that will receive the installed task history provider.

- **Provider Source Directory** - The path to a directory containing the `.history` files to be installed. This directory may exist either inside or outside the application directory. If it is outside, the `.history` files are copied to a location within the application directory that matches the location of the source `.history` file relative to the specified source directory. If the source directory is inside the application directory, no file copying is done.

These utilities scan the source directory for `.history` files and then register all the files found for deployment with the host application.

Using the command-line utility to install task history providers into a host application:

```
java com.bea.wli.worklist.build.InstallTaskHistoryProvider <options>
```

The options to this command are defined as follows:

`-appRootDir` - Holds the `META-INF` directory for the target application

`-providerSrcDir` - Holds the `.history` files to be installed into the target application. If this directory is a child of appRootDir, no file copying is done. If not, the `.history` files found under this directory (recursively) are copied into the same relative path within the target application at the root of the application.

Using the Ant utility to install task history providers into a host application:

```
<project name="MyProject">

  ...

  <taskdef name="install-taskhistoryprovider-in-app"
classname="com.bea.wli.worklist.build.InstallTaskHistoryProviderTask"

         classpath="${<Some Classpath that includes WebLogic Integration's
public.jar}"/>

  <install-taskhistoryprovider-in-app

      appRootDir="${<Host Application's root dir>}"

      providerSrcDir="${<Source directory of your provider >}"/>

  ...

</project>
```

Sample of a Custom Task History Provider Descriptor File

Create a task history provider descriptor (`.history`) file in your application that looks something like this:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<task-history-provider
                      xmlns ="java:com.bea.wli.worklist.config"
                    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <global-handler>false</global-handler>
  <!-- One or more handled-task-type-id elements if this is not a global
       provider -->
  <handled-task-type-id>/MyFolder/MyTaskPlan:1.0</handled-task-type-id>

<handled-task-type-id>/MyFolder/MyOtherTaskPlan:2.0</handled-task-type-id>
  <implementation-class-name>
    com.acme.mystuff.MyTaskHistoryProvider
  </implementation-class-name>
  <!-- Any properties you might need to properly construct your provider.
       Only string values are supported. These will be passed to the
       setProperties method of your TaskHistoryProvider impl -->
  <property>
    <name>ConstructorProp1</name>
    <value>Value for prop 1</value>
  </property>
</task-history-provider>
```

CHAPTER 7

Using Worklist Console

Worklist Console allows you to manage and monitor Worklist tasks, users, and the business calendar.

Worklist Console provides a navigation menu containing several modules for performing focused operations. Table 7-1lists the modules available in Worklist Console and summarizes the tasks associated with each of those modules.

Table 7-1 Worklist Module

Module	Associated Tasks
Worklist	For information about each of the following tasks, see Worklist Administration. Overview of the Worklist Module Security Policies Listing and Locating Worklist System Instances Managing Sessions Changing the Worklist System Instance Configuration Setting the Global Worklist System Instance Policies Setting the Worklist System Instance Policies Setting the Global Task Plan Policies Setting the Task Plan Policies Purging Tasks Listing and Locating Event Handlers Changing the Event Handler Details Listing and Locating Event Subscriptions Adding Event Subscriptions Changing Event Subscriptions Deleting Event Subscriptions Viewing Task Plan Details Changing Task Plan Details Listing and Locating Worklist Tasks Constructing a Custom Query for Task Instances Viewing and Changing Task Details Updating Task State Claiming a Task for a User Assigning a Task to User or Group Deleting Tasks Managing User Profiles Setting the E-mail Address of a User Reassigning Work to a User Listing and Locating Work Substitute Rules Adding a Work Substitute Rule Changing a Work Substitute Rule Deleting a Work Substitute Rule

Module	Associated Tasks
Worklist User	For information about each of the following tasks, see User Management. Overview of the User Management Module WebLogic Integration Users, Groups, and Roles Security Provider Requirements for User Management Listing and Locating Users Adding a User Viewing and Changing User Properties Listing and Locating Groups Adding a Group Viewing and Changing Group Properties Listing and Locating Roles Adding a Role Constructing a Role Statement Viewing and Changing Role Conditions Deleting Users, Groups, or Roles
Business Calendar	For information about each of the following tasks, see Business Calendar Configuration. About Business Calendars and Business Time Calculations Overview of the Business Calendar Module Listing and Locating Business Calendars Adding a Business Calendar Viewing and Changing Business Calendars Defining a Time Period Rule Associating Business Calendars with Users Exporting and Importing Business Calendars Deleting Business Calendars

Starting Worklist Console

Access to Worklist Console is password protected.

To start Worklist Console:

1. Open the following URL in your Web browser:

   ```
   http://host name:port/worklistconsole
   ```

   Here, *`host name`* is the host name or IP address of the WebLogic Server administrative server, and *`port`* is the server listening port.

2. Enter the username and password when prompted.

   Note: To log into Worklist Console, you must be a valid WebLogic Server user. However, the actions that you can do and the information that you can see depends upon your security policy. For more information, see “Security Policies” in Worklist Administration. For the sample integration domain, the default login details are:
   username: weblogic
   password: weblogic

   The Worklist Console home page is displayed.