Reactor 5 Programmer's Guide

Release 5.5.3

26500 Agoura Road, Suite 102-570, Calabasas, CA 91302 Phone: 818-440-1234 Fax: 818-475-5188 www.oakgrovesystems.com

SCOPE AND AUDIENCE

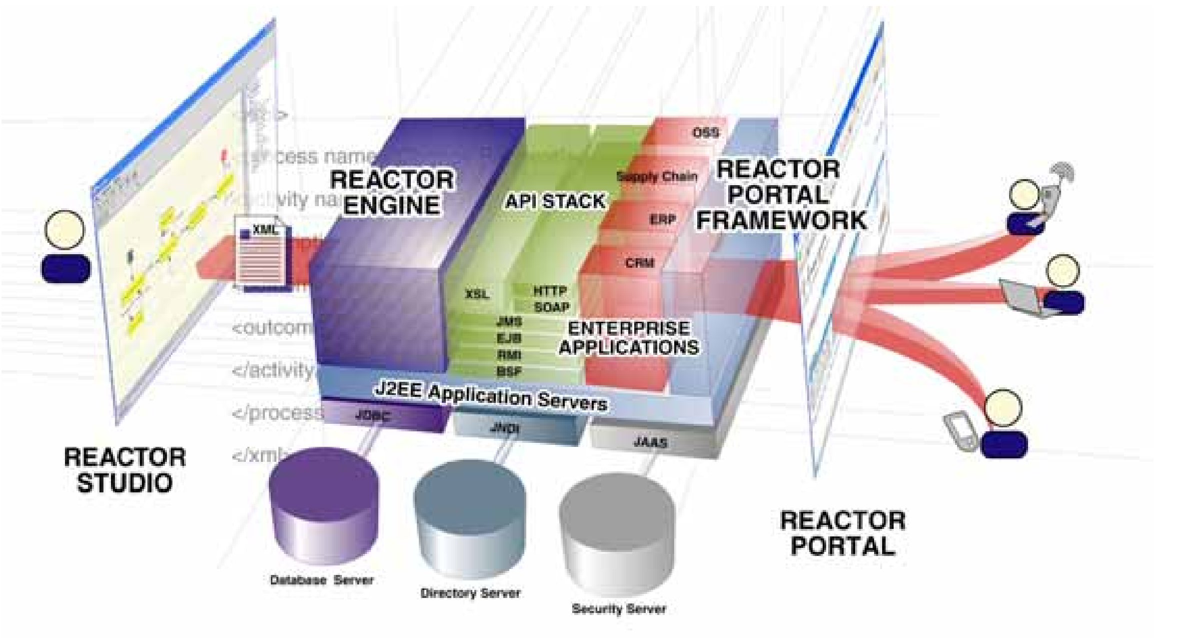
This document is intended to provide a technical view of the Reactor 5 system that will serve as an introduction and tutorial to enable a software developer to design and develop clients that work with the Reactor 5 Server.

http://www.oakgrovesystems.com
Email: info@oakgrovesystems.com

Last update: April 22, 2004

TABLE OF CONTENTS

Chapter 1 Introduction



What is Workflow?

Organizations of all sizes oscillate between conditions of chaos and order, with the forces of leadership and purpose acting to bring order, and the opposing forces of a changing marketplace, technological advances, and shifting customer priorities tending to create chaos. In other words, all successful organizations undergo ceaseless struggle to achieve and sustain the efficient conversion of their competencies and resources into received value for their customers. Success requires a delicate balance between establishing efficient, repeatable processes and maintaining the agility to adjust -- or completely replace -- these processes to fit current conditions. Software technologies sometimes referred to as Business Process Management, or simply "Workflow", can play a useful role in confronting this challenge in three ways.

1. People Acting in Concert

The actions of good managers, along with prior training and experience, largely determine how effectively members of a group can work together to bring about an aggregate result. Yet all of these factors take time to develop, and may never fully develop if the pace of change is high. A well designed process management system can help by orchestrating -- by way of notifications, reminders, delivery of resources, and tracking -- the work of many individuals involved in a business process. It can also automate much of the "startup" (e.g. finding the right forms, locating relevant policies and procedures, etc...) and "cleanup" (e.g. forwarding on to the next person in the process) in each individual activity.

2. Interleaving Automation

Not all processes can be usefully automated, but those that can will always be more efficient than corresponding human-in-the-loop processes. A well designed process management system can provide a roadmap for where automation can have the highest impact, along with an operational framework for deploying and managing automated processes.

3. Performance Analysis and Optimization

Bottom line results can indicate problems, but detailed analysis is required in order to find and fix bottlenecks and inefficiencies in operations. A properly implemented process management system can serve to collect detailed metrics on actual performance of key processes in real time, giving manager a solid basis for making decisions about how and when to make improvements.

In short, workflow systems can be thought of as a sort of "operating system" for the enterprise, whose function it is to orchestrate and track work, whether automated or carried out by humans. In the same way that databases capture what an organization consumes and produces, workflow systems encapsulate how the organization works.

Chapter 2 What is Reactor 5?

Reactor 5 Component Suite

Oak Grove Reactor 5 is a suite of products that work together to model, automate, integrate and streamline business processes, providing a platform for more efficient and productive business. The Reactor 5 suite consists of three major components:

> **Reactor 5 Server**: A completely J2EE-based process engine for the rapid deployment and flexible enactment of workflow or process integration applications. The Reactor 5 Server can be programmed by clients via its two Reactor 5 APIs. Or the Reactor 5 services can be leveraged by the EJB/Servlet developer as a process layer that is integrated into their application for rapid development of process intensive applications.
>
> **Reactor 5 Portal Framework**: A Servlet/JSP framework for rapid development of sophisticated web based user interfaces to workflow applications, or for use with existing web based systems for seamless integration with Reactor. Reactor 5 ships with an example "To-Do list" portal that is suitable out-of-the-box as an end-user To-Do list for workflow applications developed on Reactor 5.
>
> **Reactor 5 Studio Client**: A business process-modeling tool for the rapid development of process maps and activity diagrams. This tool gives application developers an easy way to build, organize, and reorganize the complex business processes that lay at the heart of their workflow or process integration application. Studio increases application developer productivity by giving them the power to quickly build and deploy process intensive systems.

The multi-platform, application server neutral J2EE-based process engine (Reactor 5 Server) is central to the Reactor 5 suite. Developers and system administrators can deploy the Reactor 5 Server to their application server (currently verified with BEA WebLogic, IBM WebSphere, Oracle 9iAS, JBoss, Orion, and others) for rapid deployment of tightly integrated workflow or process integration applications on virtually any os/hardware platform or environment. For customers who do not already have an application server, Reactor 5 Server comes bundled with the JBoss application server for one-click deployment of the entire platform.

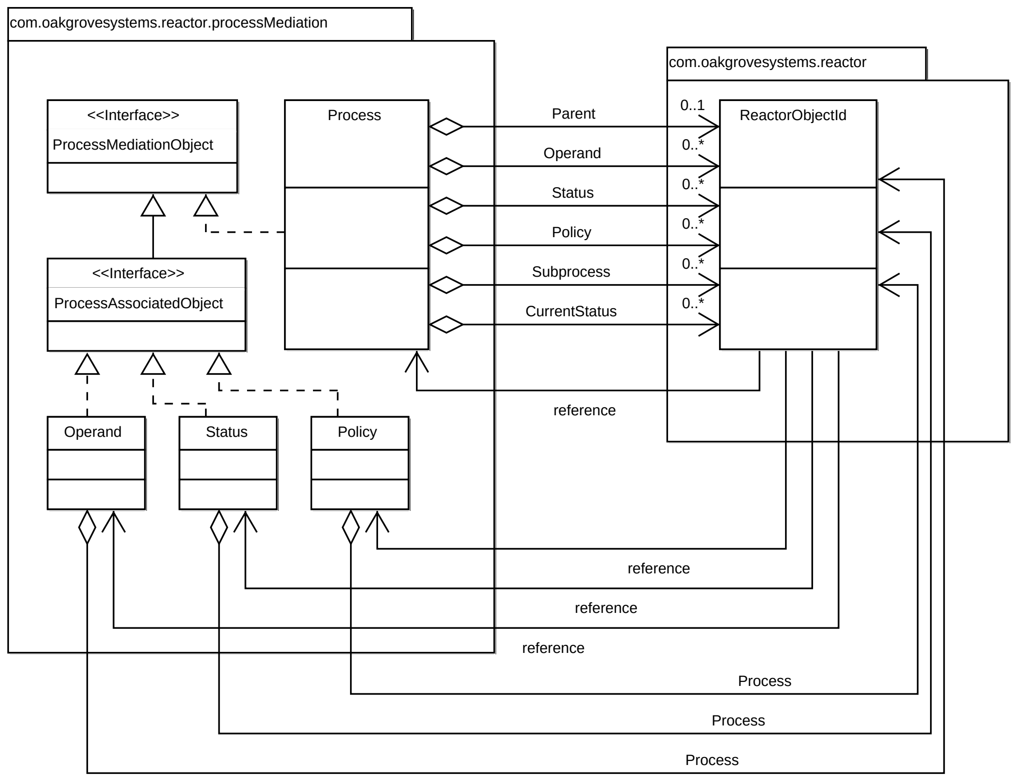
The Reactor 5 Server integrates seamlessly with existing J2EE applications. This makes Reactor 5 the best choice for J2EE application developers who want to leverage a process layer for more efficient business processes within their environment, as well as rapid reaction to change when demands of business change. This process layer can be integrated with any number of existing applications and solutions, including CRM, manufacturing, Web Services, workflow, and so on.

Once the Reactor 5 Server is deployed, the Reactor 5 Studio tool can be used to quickly build a process map that accurately reflects the business logic of the application at hand. Reactor 5 can handle arbitrarily complex business or application process logic, as well as all known workflow and process modeling patterns. The process designer can use the Reactor 5 Studio features which enable both point and click process authoring and more powerful scripting language based programming capabilities that enable automation of the most sophisticated business processes. Process maps from Reactor 5 Studio are represented in XML, which can be uploaded to the Reactor 5 Server for immediate execution by end-users.

Process participants, process owners, managers, and process administrators interact with Reactor 5 capabilities either via the web based applications they’re already familiar with, or via a To-Do list “Portal” that has been modified from the supplied Reactor 5 Portal Framework, or developed in-house. End users are able to more effectively manage work assigned to them and coordinate automated or combined manual-automated tasks via this web based system.

Chapter 3 Object Model

Object Model Overview



Business processes are modeled in Reactor 5 with four different types of objects: Process, Operand, Status, and Policy.

Process objects are the primary building blocks of a model. They contain information about the current **state** and status of the process, the conditions under which it should start, stop, and change state, and the relationships the Process has with other objects. Most non-trivial business processes consist of multiple steps (subprocesses) that can often themselves be broken down into smaller, simpler steps. To model such processes, Process objects are associated with an arbitrary number of subprocesses, which are themselves Process objects, and can be broken down recursively into other subprocesses. Process objects represent both process definitions ("definitions") and enactments of a process definition ("instances"). The only difference between a Process object that is an instance and one that is a definition is that the attribute "definition" is true for definitions and false for instances. Also, some attributes are only relevant for instances (current statuses, start time, and end time.) Instances are cloned from definitions by issuing a CloneInstance command to the Reactor 5 system.

Operand objects (called Process Data in the Studio tool) encapsulate arbitrary data that is relevant to the business process modeled by a Process. Examples include a document URL or a purchase order number.

Status objects are very simple objects used to trigger conditions in Process objects or cause the execution of Policy objects, as well as to generally provide information about how a process is progressing.

Policy objects encapsulate arbitrary logic that is relevant to the process being modeled. They allow processes to be custom scripted and are extremely flexible. Policies associate a Java class or policy execution service script with an event that should trigger its execution.

Note the careful separation between the process data and logic and the business data and logic. Process objects represent only the abstract process data and logic. Operands represent the business process data and policies represent the business process logic. Statuses are the connection between the process logic and business logic.

These are some common tasks in the lifecycle of a process:

- author
  - creates a process definition
  - queries process definitions
  - updates a process definition
- invoker
  - clones a process definition, creating an instance
  - sets operands values in a process instance
  - starts a process instance
- participant
  - queries processes
  - queries a process tree
  - sets operand values in a process instance
  - adds current status to a process instance
  - stops a process instance

Referring to Specific Objects: label path

Each object stored by Reactor 5 has a globally unique identifier which is a string of characters that combines hexadecimal digits (0-9, a-f), colons (:), and dashes (-). These IDs are used by an object to identify other objects that are associated with it. For example, Process uses object IDs to list its associated Operands. Requests to Reactor 5 can use object IDs to specify objects.

In some cases, the exact ID of the object might not be known by something that needs to refer to the object. A Policy may need to set the value of an Operand, but the ID of that Operand is different for each instance of the associated Process. That Policy cannot identify the Operand using an ID. When an application creates a new object, Reactor 5 assigns it a unique ID. The application needs to use its own object IDs internally before sending the creation request to Reactor 5, but those internal Ids are not guaranteed to be globally unique. Applications creating new objects may need a different way to refer to specific objects.

The "label path" is an alternate way to refer to an object when using the object ID is not feasible. A label path contains a sequence of strings, where each string is the label of a parent or ancestor of the object, and the last label belongs to the object itself. A label path can include the ID of an object from which to start the path. If no ID is included, then the first label must refer to an object that has no parent Process or associated Process.

The syntax for label paths is described further in Appendix A: Label Paths.

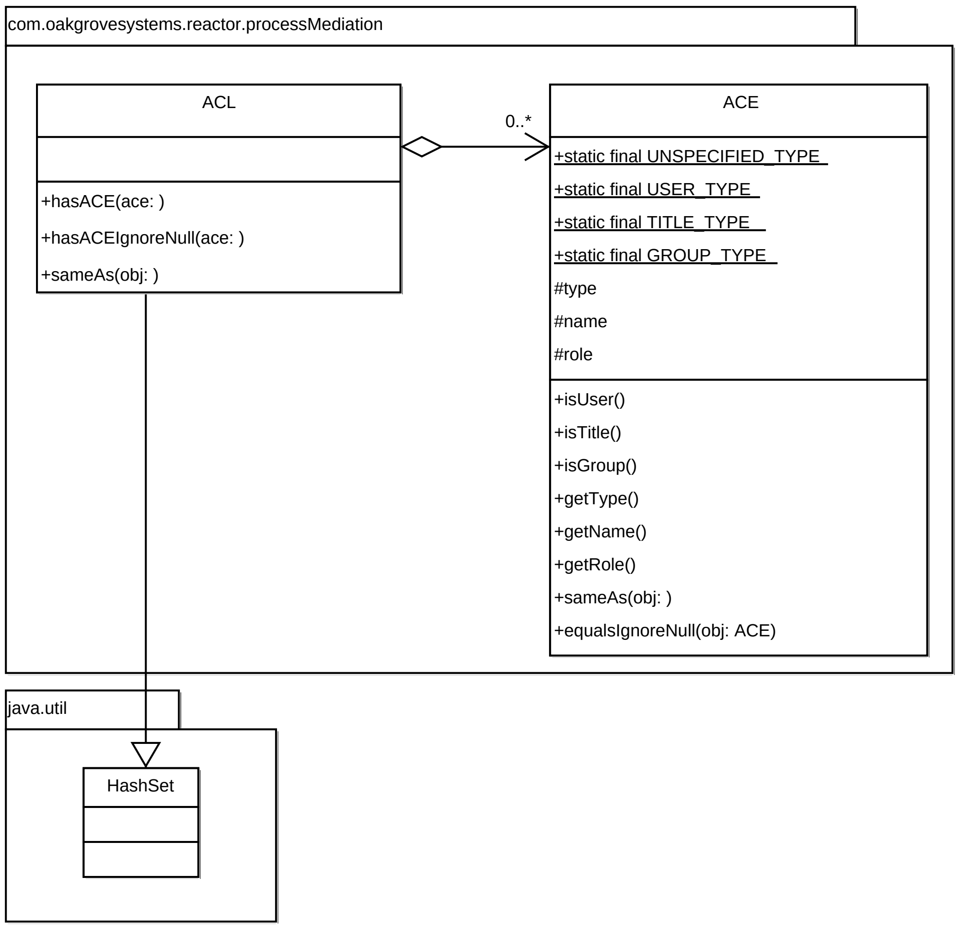
Assigning Roles to People and Groups: ACLS

Each Reactor 5 object of the four primary types has an Access Control List (ACL) containing Access Control Elements (ACEs). Each ACE indicates that a person or group has some role pertaining to the Reactor 5 object. The person or group can be defined explicitly by providing a name, or implicitly by specifying the title of a role to be resolved by the enterprise directory service.

Each ACE has the following attributes:

- Name (name of the user, organizational title, or group)
- Type (indicates whether the ACE refers to a user, title, or group)
- Role (relationship relative to the Reactor 5 object)

For example, suppose a Reactor 5 system is configured with two roles: "owner" and "assignee". The "assignee" role may only be associated with the permissions to add or remove a current status or stop Process objects. The "owner" role may be associated with all permissions. In this situation, one can edit the ACL of a particular object to specify which users, groups, and titles have "owner" permissions and which users, groups, and titles have "assignee" permissions with respect to that particular object.



Processes

Each Process object is associated with an arbitrary number of other Process objects, which are subprocesses. Thus, the objects form a tree where each Process object is associated with a node in the tree and no two Process objects are associated with the same node. The nodes associated with the subprocesses are child nodes of the node associated with the parent process. Each Operand, Status, and Policy object is also associated with a node in the tree, but multiple Operands, Statuses or Policies can be associated with the same node. There is no Process associated with the root node of the Reactor 5 data, but the root node may be associated with "global" Operands, Statuses, and Policies.

Each Process has the following attributes and associations:

- Attributes:

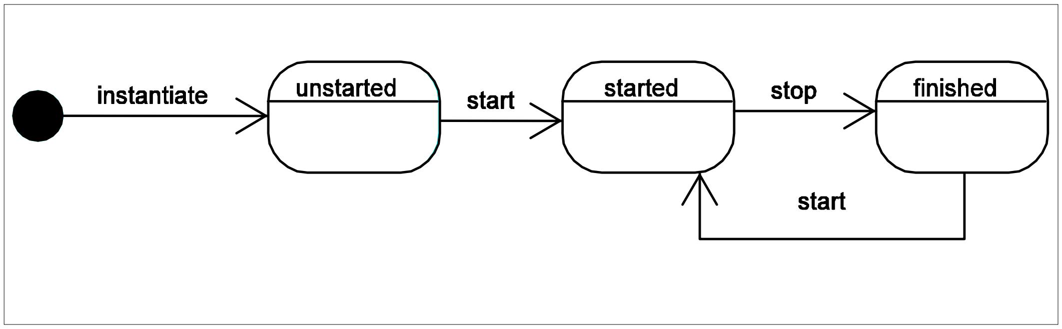
ID	A String that uniquely identifies an object within the Reactor system.
label	Labels are short, human-readable strings that can be used as an alternative to ids to identify a Process. All the objects associated with a Process should have unique labels.
description	Descriptions are human readable text describing a Process. They are not used by internal Reactor functionality, but are included for use by Reactor clients.
ACL (set of ACEs)	The relationships between the Reactor objects and the users of the system are captured using ACLs, which are composed of ACEs (Access Control Elements). User- A single person known to the directory and authentication services Group- A collection of users defined in the directory service Title- A string associated with one or more users in the directory service (functionally similar to a group). E.g. CEO, CTO, VP Sales, Director of Marketing Role- A string associated with a collection of R5 permissions. This string describes the relationship between a user and an object. There is one set of roles for the Reactor system. This set is completely customizable. ACL (Access Control List)- A set of mappings (ACEs- Access Control Elements) that determines which sets of users, groups, and titles have which roles with respect to a given object.

is definition (true, false)	A boolean indicating whether the object represents a Process definition (true) or a Process instance (false)
state (unstarted, started, finished)	See the next section 'states and current statuses' for details.
start date	If a process restarts (iterates), then the start date holds the most recent one
end date	If a process restarts (iterates), then the end date hold the most recent one
preconditions	The preconditions of a Process dictate the circumstances under which it should be started automatically. Specifically, one of the conditions specified by the preconditions attribute must be false and become true in order for the Process to be started automatically.
change conditions	The change conditions of a Process dictate the circumstances under which the Process's state or current statuses should change. A change condition for a Process is composed of a condition that must be met and a change that should be applied to the Process when the condition is met. The condition specified by a change condition must be false and become true for the change to be applied automatically. changeConditions is a Set of instances of ProcessChangeCondition.
timers	An ordered list of timers (TimerSpec objects) that will be started in the Reactor Timer Service when the Process starts and deleted when the Process stops. The ordering of the List is used to match up TimerSpecs with the timer IDs of the started timers. The ids for the timers are stored in the timerIds List such that the first id in the TimerIds list corresponds to the first Timer in the timers list, and so on. The timerIds list is populated at the time the Process is started. Thereafter, the lengths of the timers and timerIds Lists should always be equal.

- Associations:

parent process	The ReactorObjectId of the Superprocess of this Process
subprocesses	The set of ReactorObjectIds of the Subprocesses of this Process
operands	The set of ReactorObjectIds of the Operands associated with this Process
statuses	The set of ReactorObjectIds of the Statuses associated with this Process
current statuses	The current statuses attribute of a Process object indicates the Statuses that currently apply to the Process. It is not used by Process definitions- only instances. A Process can have any number of current statuses at a given time. Reactor adds or removes Statuses to the Process's set of current statuses in response to an 'add status' or 'remove status' command or by application of the Process's change conditions.
policies	The set of ReactorObjectIds of the Policies associated with this Process

States and Current Statuses



The **state** attribute of a Process object holds the information on the current state of the process. It is not used by Process definitions, only by instances. Process instances can be in one of three states: **unstarted**, **started**, or **finished**. When an instance is cloned from a definition, it is placed in the **unstarted** state. It remains in the **unstarted** state until either a **Start** command is issued to start it, or until its precondition becomes true and it is started automatically. It then transitions to the **started** state. It remains in the **started** state until either a **Stop** command is issued to stop it or one of its **change conditions** becomes true and the change dictated by that **change condition** indicates that the process should be stopped. It then transitions to the finished state. From the finished state, an instance can be restarted (transition back to the started state) in the same way that it transitions from the unstarted state to the started state, by a Start command or by its precondition becoming true. A Process will also transition to the finished state from either of the other states if its parent process transitions to the finished state. State transitions can also occur as a result of application of the Process's change conditions.

The "current statuses" attribute of a Process object indicates the statuses that currently apply to the Process. It is not used by Process definitions, only by instances. A Process can have any number of current statuses at a given time. Statuses can be added or removed to the Process's set of current statuses by issuing an AddStatus or RemoveStatus command to the Reactor 5 system or by application of the Process's change conditions.

Pre-Conditions and Change Conditions

The basic relationships between Processes are captured in their precondition and change conditions attributes. As implied by the name, the precondition of a Process dictates the circumstances under which it should be started automatically. Specifically, the condition specified by the precondition attribute must be false and become true in order for the Process to be started automatically. The change conditions of a Process dictate the circumstances under which the Process's state or current statuses should change. A change condition for a Process is composed of a condition that must be met and a change that should be applied to the Process when the condition is met. The condition specified by a change condition must be false and become true for the change to be applied automatically.

There are two types of simple conditions that can be used in preconditions and change conditions: ProcessStateEquals and ProcessHasStatus. A ProcessStateEquals condition specifies a Process object and one of the three possible states of that Process (unstarted, started, or finished). It evaluates to true if the state of a Process specified is equal to the state specified. A ProcessHasStatus condition specifies a Process object and a Status object. It evaluates to true if the Status specified appears in the set of current Statuses of the Process specified.

The conditions used in preconditions and change conditions may also be compound conditions. There are three types of compound conditions: conjunctions, disjunctions, and inversions. A conjunction specifies a set of other conditions and evaluates to true if and only if all of the conditions in the specified set evaluate to true. A disjunction specifies a set of other conditions and evaluates to true if and only if one or more of the conditions in the specified set evaluate to true. The conditions in the set of a conjunction or disjunction may either be simple conditions or compound conditions. An inversion specifies another condition, and evaluates to true if and only if the condition specified evaluates to false. The condition specified by an inversion may be either a simple condition or a compound condition.

There are three types of changes that can be used in change conditions: ProcessStateChange, ProcessStatusAddition, and ProcessStatusRemoval. A ProcessStateChange specifies the new state that a Process object will have after the change has been applied. ProcessStatusAddition and ProcessStatusRemoval changes specify the status that should be added or removed respectively from the Process's set of current statuses when the change is applied.

When the state or set of current statuses of a Process instance changes, the preconditions or change conditions of the changed Process or other Processes may become true, which triggers further changes, which may make other preconditions or change conditions true. This cascade of changes continues recursively. This is how Processes are automated by the Reactor 5 system. Changing one Process automatically causes other Processes to change and fire events that can trigger Policy execution or external actions.

Specifically, when a Process's state or current status changes, the following conditions are evaluated (in order) and if they have been made true by the change, the appropriate resulting change is applied.

1. The preconditions of the subprocesses of the changee.
2. The preconditions of the peer processes of the changee (processes with the same superprocess as the changee).

3. The change conditions of the changee.
4. The preconditions of the changee.
5. The change conditions of the superprocess of the changee.

Timers

Reactor 5 provides the ability to schedule actions for some point in the future. This is accomplished by setting timers that expire at some point in the future. When a timer expires, it fires an event, which can trigger policies.

A common use for timers is to manage due dates for tasks. For example, suppose that if a Process object has not finished after two days, some action should be taken. A good way to accomplish this is to associate a policy with the "ProcessStarted" event for the Process that sets the timer with a specific event type. Then associate a policy with the timer expiration event that takes the necessary action.

Each Timer has the following attributes:

schedule date	protected java.util.Date scheduleDate
schedule expression	protected java.lang.String scheduleExpression
repeat expression	protected java.lang.String repeatExpression
repeat count	protected int repeatCount
end date	protected java.util.Date endDate
end expression	protected java.lang.String endExpression
calendar	protected java.lang.String calendar
description	protected java.lang.String description
event	ReactorEvents are used by the Reactor System to trigger Reactor Policy execution. A single Event can trigger the execution of multiple Reactor Policy objects. And a single Reactor Policy can be triggered by different kinds of events. ReactorEvents are characterized by their type (a.k.a. name) (e.g. "ProcessStateChange"), their source (e.g. "ProcessCommandService", "TopLevelProcessX.SubProc1", etc.),

and their attributes, which are stored in a `Map`. ReactorEvents are sent from various parts of the Reactor system to the Policy Execution Service, which decides which Policy objects should be executed. Events can also optionally be scoped to the ancestor nodes of a given node, called the scopeNodeId. In this case, only policies associated with the ancestor nodes of the scopeNodeId are eligible to be started as a result of this event. ReactorEvent objects are immutable.

The event can contain arbitrary key/value pairs.

Operands

Each Operand has the following attributes and associations:

- Attributes:

ID	A String that uniquely identifies an object within the Reactor system.
label	Labels are short, human-readable strings that can be used as an alternative to ids to identify an object. All the objects associated with a Process should have unique labels.
description	Descriptions are human readable text describing an object. They are not used by internal Reactor functionality, but are included for use by Reactor clients.
ACL (set of ACEs)	An ACL (Access Control List) defines the relationships between a Reactor object and the users of the system. An ACE is composed of ACEs (Access Control Elements).
visible in subtree (true, false)	A boolean that defines the scope of the Operand.
value	The data stored in the Operand.

- Associations:

process	The id of the Process with which this Operand is associated

If an Operand has its "visible in subtree" attribute set to "true", then the Operand's value can be accessed by subprocesses of the process associated with the Operand. Otherwise, only the Process associated with the Operand can access the Operand's value.

Operands can store any value that can be expressed as a string. Objects or complicated data structures can be stored by converting to an XML representation or using Java serialization. However, this is not recommended. Instead, the recommended approach is to store the data in another system, such as a database, and store a handle or reference ID in the operand value itself.

Statuses

Each Status has the following attributes and associations:

- Attributes:

ID	A String that uniquely identifies an object within the Reactor system.
label	Labels are short, human-readable strings that can be used as an alternative to ids to identify an object. All the objects associated with a Process should have unique labels.
description	Descriptions are human readable text describing an object. They are not used by internal Reactor functionality, but are included for use by Reactor clients.
ACL	An ACL (Access Control List) defines the relationships between a Reactor object and the users of the system. An ACE is composed of ACEs (Access Control Elements).
available in subtree (true, false)	A boolean that defines the scope of the Status.

- Associations:

process	The id of the Process with which this Status is associated

If a Status has its "available in subtree" attribute set to "true", then the Status can be added to subprocesses of the process associated with the Status. Otherwise, the Status can only be added to its associated Process.

Statuses can be used for a variety of purposes. The most common usage is to use statuses in preconditions, for specifying transitions between activities. Statuses can also be used to trigger policy execution. This can be independent of any process transition. Finally, statuses can be used as markers that provide meaning when they are queried or displayed by client applications.

Policies

Each Policy has the following attributes and associations:

- Attributes:

ID	A String that uniquely identifies an object within the Reactor system.
label	Labels are short, human-readable strings that can be used as an alternative to ids to identify an object. All the objects associated with a Process should have unique labels.
description	Descriptions are human readable text describing an object. They are not used by internal Reactor functionality, but are included for use by Reactor clients.
ACL (set of ACEs)	An ACL (Access Control List) defines the relationships between a Reactor object and the users of the system. An ACE is composed of ACEs (Access Control Elements).
event source	the event source a ReactorEvent must have to trigger the execution of this Policy (null indicates that a ReactorEvent with any event source may trigger the execution of this Policy)
event name	the event name (a.k.a. event type) a ReactorEvent must have to trigger the execution of this Policy (null indicates that a ReactorEvent of any type may trigger the execution of this Policy)
event attributes	the event name (a.k.a. event type) a ReactorEvent must have to trigger the execution of this Policy

language	the language of this Policy (only used with BSF Policies)
source code type (CODE, CLASSNAME, URL)	whether the String sourceCode attribute contains the classname, url or raw source for the executable code of this Policy
source code	the classname, URL, or raw source for the executable code of this Policy, as specified by the sourceCodeType attribute
security policy	the name of the security policy to enforce on the execution of this Policy

- Associations:

process	the id of the Process with which this Policy is associated

The event source, event name, and event attributes combine to describe the event that triggers the policy execution. The language, source code type, and source code specify the action to be taken by the policy.

Each Policy consists of one of two types of executable instructions that are executed when the Policy is triggered: Java classes and BSF scripts. The source code type attribute of a Policy attribute indicates the type of the Policy and how to access the executable instructions so that they can be executed. The source code type must be one of the following three values:

- SourceCode (indicates that the executable instruction is a Policy execution script)
- Classname (indicates that the executable instruction is a Java class)
- URL (indicates that the executable instruction is a Policy execution script)

If the source code type is Classname, the contents of the source code attribute of the Policy are assumed to be a fully qualified Java class name. It attempts to load the class from the class path of the application server and execute an instance of it. The class must implement the `java.lang.Runnable` interface to be executed correctly.

If the source code type is SourceCode, the contents of the source code attribute of the Policy are interpreted as raw source code of the language specified by the language attribute of the Policy. This source code is sent to the Policy execution service.

If the source code type is URL, the contents of the source code attribute of the Policy is assumed to be a URL that can be used to retrieve the raw source code of the language specified by the language attribute of the policy. If no language is specified by the language attribute of the policy, the Reactor 5 system attempts to determine the language from the file extension of the URL. The source code is sent to the Policy execution service.

Chapter 4 Events and Policy Execution

Events

The execution of a Policy is triggered in response to the firing of an event. There are three basic sources of events. Some events are triggered by changes to a process, either a change in state or a change in current status. Other events are triggered by a Process's timers. External systems can also send events to Reactor 5.

Each event consists of an event type, an event source, a table of event attributes, and optionally a Process ID that determines the scope of the event. If the scope Process ID is included, only Policies associated with the Process specified or its ancestor Processes or Policies in the root node may be triggered by the event. The attributes in the table are specific to the type of event. For example, the attributes of a "ProcessFinished" event contains values for the following keys: "ProcessID" and "ProcessLabelPath"; the attributes of a "StatusAddition" event contains values for the following keys: "ProcessID", "ProcessLabelPath", "StatusID", and "StatusLabelPath".

Future, external systems will be able to register with Reactor 5 to be notified of events.

Events only trigger the execution of Policies that match the event. The attributes of a Policy determine which events the Policy matches. A Policy can specify the type, source, and/or attributes of the events that should trigger it. For example, if a Policy specifies a type of "ProcessStarted" and an attribute value of "foo.bar" for the key "ProcessLabelPath", any event of type "ProcessStarted" with that attribute in its list of attributes will trigger the execution of that Policy, regardless of the source of the event or the other attributes of the event.

Process Change Events

The source of process change events is "ProcessCommandService", an internal Reactor 5 service. These are the different event names sent by the Process Command Service:

- ProcessStarted
- ProcessFinished
- StatusAddition
- StatusRemoval

The ProcessStarted event has these properties:

- ProcessID (ID of the started process)
- ProcessLabelPath (label path of the started process)

The ProcessFinished event has these properties:

- ProcessID (ID of the stopped process)
- ProcessLabelPath (label path of the stopped process)

The StatusAddition event has these properties:

- ProcessID (ID of the process)
- ProcessLabelPath (label path of the process)
- StatusID (ID of the status)
- StatusLabelPath (label path of the status)

The StatusRemoval event has these properties:

- ProcessID (ID of the process)
- ProcessLabelPath (label path of the process)
- StatusID (ID of the status)
- StatusLabelPath (label path of the status)

Timer Events

The source of timer events is "TimerService", an internal Reactor 5 service. Every timer event has the name "TimerExpired". Properties are specified by the process author who creates the timers, but every timer has a property named "ProcessID" which identifies the process containing the timer.

Writing Policies

Reactor 5 allows policy code to be written in a variety of languages, including but not limited to Java, JavaScript, Tcl, and Python.

Using Precompiled Java Classes

Any Java class that implements the `java.lang.Runnable` interface or the `com.oakgrovesystems.reactor.PolicyScript` interface can be invoked by a Policy. Specify the class name in the Policy's "source code" attribute. The class must be accessible to the classloader that loads the PolicyExecutionMessageBean in the application server. A good way to make it accessible to this classloader is to place it in the ReactorPolicies.jar file within the Reactor EAR file. Alternatively, some application servers have alternate mechanisms to “hot deploy” classes into their runtime classpaths, which may be preferable to modifying the EAR file.

The Navigator class is useful for accessing Reactor 5 objects from within a Policy. It can also be helpful to have access to the Policy object and the ReactorEvent that triggered the Policy execution. These objects are made available to classes that implement the PolicyScript interface rather than just the Runnable interface. Prior to calling the run() method of a PolicyScript, Reactor calls its setTriggeringEvent(), setPolicy(), and setNavigator() methods, which allows the class to store references to those three objects in instance variables for subsequent use by the run() method. For more details, see the Javadoc API for the `com.oakgrovesystems.reactor.PolicyScript` class.

Using Uncompiled Source Code via BSF

Reactor 5 can use IBM's Bean Scripting Framework (BSF) to execute policies. This allows policy code to be written in a variety of languages, including but not limited to Java, JavaScript, Tcl, and Python.

Using Java Source Code

Set the Policy's source type to "Source Code", and its source language to "Java". Put valid Java source code into the Policy's source code attribute. This code will be inserted into a Java method. The return value from this method will be ignored, so just return null. The method type is not void, so always include the return value.

Instead of making the source code an attribute of the Policy object, one could instead place the source code elsewhere (on the network or file system, for example) and simply provide the URL of the source code to the Policy. In this case, set the Policy’s source type to "URL" and set the Policy source code attribute to be the URL. The code will be retrieved from the URL and placed into a Java method. The code should return null; as described above.

The Navigator class is useful for accessing Reactor 5 objects from within a Policy. Use this code to get a Navigator object from a Java Policy:

```
Navigator navigator = (Navigator) bsf.lookupBean("navigator");
```

For more details, see the Javadoc API for the Navigator class in the `com.oakgrovesystems.util` package.

Using Other Languages

BSF allows other languages to access the Navigator object, even though it is a Java object. See the BSF documentation for more information. BSF was originally developed at IBM:

http://www-124.ibm.com/developerworks/projects/bsf/

Future versions of BSF will be released through the Apache Jakarta project:

http://jakarta.apache.org/bsf/

Writing Reusable Policies

The first step to making a Policy reusable is to avoid hard-coding any data into the script. Put data in Operands. The Reactor 5 Studio provides a mechanism to add new Policy templates. A Policy template (called a "parameterized Policy") is made available to Studio by creating an XML file in the `..\studio\actions` directory of the Studio distribution. See that directory for examples of XML files with the correct format.

Chapter 5 Writing Reactor 5 Clients

There are currently three interfaces to communicate with the Reactor 5 Server: calling EJB Session Bean methods, sending SOAP messages via HTTP, and sending Reactor XML via HTTP. There will soon be facilities for sending JMS messages to the Reactor 5 Server. Until then, application developers can create custom message-driven EJBs to receive JMS messages and send them to the Reactor 5 Server through existing interfaces.

Regardless of the underlying mechanism, the concepts used by all the interfaces are the same. The application creates a request with a certain type, adds an authentication token, possibly adds some parameters, and then sends the request to the Reactor 5 Server. Reactor 5 then sends back a response containing a result code and possibly some return values. The result code contains a numeric code indicating success or failure, a corresponding text message meant to be read by users, and possibly some information that could help with debugging if something went wrong. The authentication token is obtained at the beginning of a session by providing the proper credentials in a Login request (which does not require a valid authentication token.) The result codes are listed in Appendix E: Result Codes.

Java developers can also use the client API which provides shortcut methods for sending requests and managing objects.

Authentication

Except for the Login request, each request requires a valid authentication token. This authentication token is acquired by sending a Login request with a valid username and password. The response, if successful, contains an authentication token must be used by subsequent requests. The token serves to identify the person or entity making the request. Requests with an invalid token are denied.

Creating a Reactor Client

Writing Client Code

The com.oakgrovesystems.reactor.client.ReactorProxy interface provides methods for interacting with Reactor 5 Server. The com.oakgrovesystems.reactor.client package also includes two concrete implementations of the ReactorProxy interface, XMLReactorProxy and EJBReactorProxy. The XMLReactorProxy class uses XML/HTTP to communicate with Reactor 5 Server. The EJBReactorProxy class communicates with an EJB stateless session bean to provide the same functionality. The EJBReactorProxy in most cases has better performance, but the XMLReactorProxy class makes it easier to communicate through a firewall.

The ReactorObjectId and LabelPath classes are used to identify objects. They are found in the com.oakgroveystems.reactor package. The ReactorObjects class, also found in com.oakgroveystems.reactor, provides methods for storing and accessing objects that have been queried from Reactor 5 Server. The QueryCriteria class, found in the com.oakgrovesystems.reactor.client package, is used to create queries for getting objects from Reactor 5 Server. The objects themselves are described in the com.oakgrovesystems.reactor.processMediation package. The four main classes are Process, Operand, Status, and Policy. The com.oakgrovesystems.processMediation.xml package contains builder and outputter classes which convert objects to XML representations and back to objects. For details, see section Chapter 124578440 "Reactor Java API" in this document, or the JavaDoc HTML in the Reactor 5 Server distribution in this location:

```
docs/api/index.html
```

Building a Client Application

The file Reactor-5.0.jar contains classes necessary to compile an application that connects to the Reactor 5 Server. It is located in the `share` directory in the Reactor 5 Server distribution.

Running a Client Application

In addition to Reactor-5.0.jar, other jar files may be required to run a client application.

If the application connects to Reactor 5 Server using XMLReactorProxy, then the following jar, distributed with Reactor 5 Server, must be included in the classpath:

```
- share/jdom.jar
```

If the application connects to Reactor 5 Server using EJBReactorProxy, then the EJB and JNDI jar files must be present. Each application server may have its own EJB and JNDI client jar files. The following jar files will work with the distribution for the JBoss application server:

```
- share/ejb.jar
- share/jndi.jar
- jboss321/client/jnp-client.jar
- jboss321/client/jboss-client.jar
- jboss321/client/jbosssx-client.jar
```

In addition to having the EJB and JNDI jar files, clients using the EJB interface will need to configure JNDI properly. In general, this will be done by having a `jndi.properties` file in the runtime classpath of the application. The file `jboss321/bin/jndi.properties`, in the distribution for the JBoss application server, can be copied for use in client applications.

Sample Client Applications

The Reactor 5 Server distribution contains examples of client applications. See the samples/client directory for code, build scripts, and launch scripts.

For Windows, use `build.bat` and `run.bat`.

For UNIX, type

```
/bin/sh build.sh
```

and

```
/bin/sh run.sh
```

to use the build and launch Bourne shell scripts.

The com.oakgrovesystems.portal.action package in the Reactor 5 Portal Framework is code that is intended to be executed in a servlet, but it uses the ReactorProxy interface like a client application. The source code is available in the Reactor 5 Server distribution under the `portal` directory.

Creating a Reactor Web application

Reactor 5 Portal Framework

Overview

The Reactor 5 Portal Framework is a collection of patterns, conventions, and Java classes for building web applications that communicate with Reactor 5 Server. A web application built with the Reactor 5 Portal Framework consists of JSP pages, Java servlets, and a web.xml configuration file. Some web applications may include additional Java classes, HTML files, and images.

The JSP pages use HTML, JSP tags, and Apache Struts tags to display data passed to the JSP pages through Java Beans. The JSP tags provide access to data that is passed through the session or through the request, forwarded from the servlet. The Apache Struts tags provide the capability to iterate over lists of beans. The Apache Struts tags also provide the capability to conditionally display data, based on the value of bean properties. This makes it possible to have JSP pages with no Java code, suitable for graphic artists to edit the displays with third-party HTML editing software. More information about Apache Struts is available here:

> http://jakarta.apache.org/struts/

Since the JSP pages do not contain any Java code, all the behavior takes place in servlets. Each servlet delegates most of the work to one or more "action" objects. The servlet passes a ReactorProxy object to the action object, which allows the action object to communicate with the Reactor 5 Server.

When an action object executes, it creates content beans with text properties that are ready to be displayed. The servlet then gets the content beans from the action object and uses the servlet request context to pass them to the JSP page for display.

The source code to the Reactor 5 Portal Framework is bundled with the Reactor 5 Server distribution, in the ...\samples\portal\src\framework directory.

Content Beans

The com.oakgrovesystems.portal.content package defines the following content beans:

- OperandContentBean
- OperandListBean
- ProcessContentBean
- ProcessListBean
- StatusContentBean
- StatusListBean
- ValueBean

The OperandContentBean has these properties:

- label
- value
- dataType

The OperandListBean has these properties:

- operands (returns a List)
- size

The ProcessContentBean has these properties:

- id
- idURLEncoded (useful for including in URLs)
- labelPath
- labelPathURLEncoded
- label
- description

- startDate
- state ("unstarted", "started", "finished")
- parentLabel
- title
- lockState ("", "eligible", "acquired", "standby")

The ProcessListBean has these properties:

- processList (returns a List)
- size

The StatusContentBean has these properties:

- id
- idURLEncoded (useful for including in URLs)
- label
- description

The StatusListBean has these properties:

- statusList (returns a List)
- size

The ValueBean is used for storing simple strings. It has one property:

- value

The com.oakgrovesystems.portal.content package also includes the ProcessTemplate class. This has a static method toString(String, Process, ReactorObjects). This takes a template string and fills in values for the process and associated objects.

Actions

The com.oakgrovesystems.portal.action package contains the following action objects:

- AcquireActivity
- ApplyLogin
- ApplyStartForm
- ReleaseActivity

- ShowActivity
- ShowActivityList
- ShowAttachment
- ShowCompletionForm
- ShowProcessList
- ShowStartForm

Each action must be initialized before it can be executed. First, the setServerProxy() method passes a ReactorProxy object to the action, so it can communicate with the Reactor 5 Server. The setAuthentication() method passes an Authentication object to the action, which is used to get the identity of the user who initiated the action. The setParameters() method passes a Map object with key/value pairs obtained by calling the Servlet Facade's parseParameters() method.

After initialization, one calls the execute() method which does all the work. This can throw a PortalException object if anything goes wrong. If the action succeeds, it might set properties that can be accessed with "get" methods particular to a particular action class. An appropriate use of these properties would be to indicate which type of display should be used to present the contents generated by the action. The contents themselves are accessed through the getContent() method of the action object. The PortalServlet base class provides a convenience method, setContent(), for copying beans from the action's content to the servlet's request context.

Once the content beans are placed in the servlet's request context, the servlet should forward the display to the appropriate JSP page. In cases where the servlet is processing a submitted form, often it is appropriate to redirect to some other page rather than forward to a JSP for display. One such example is a servlet that processes the form to complete an activity. Once the activity is complete, the servlet redirects to a servlet that shows the list of available activities that remain.

AcquireActivity Action

Use the AcquireActivity action when a user wants to lock an activity, so that other users can no longer participate. The activity's ACL should have an entry where the user holds the "Eligible Participant" role. After executing the AcquireActivity action, all entries with the "Eligible Participant" role will be modified to have the "Standby Participant" role, and the user will be modified to have the "Activity Participant" role. Use the ReleaseActivity action to reverse the effects of an AcquireActivity action. This action takes the ID of a process instance, along with the ID and label of an optional status, and makes an addStatusToProcess() request to Reactor 5 Server. If no status is specified, this action makes a stopProcess() request instead of an addStatusToProcess() request.

Parameters:

Id	String with the ID of the process instance

ReactorProxy Methods Called:

Lock()	attempts to add the Initiative Participant role for the user, and also move Eligible Participant ACEs to have the Standby Participant role, for the selected process instance

Content:

(none)

Side Effects:

The user gains the Initiative Participant role in the process instance.

ApplyCompletionForm Action

Use the ApplyCompletionForm to stop an activity. The status and operands of an activity can be updated as part of completing the activity.

This action takes the ID of a process instance, along with the ID and label of an optional status, and makes an addStatusToProcess() request to Reactor 5 Server. If no status is specified, this action makes a stopProcess() request instead of an addStatusToProcess() request.

Parameters:

id	String with the ID of the process instance
statusLabel	String with the label of the selected status
statusIdFor${LABEL}	String with the ID of the status with the label ${LABEL}

ReactorProxy Methods Called:

addStatusToProcess()	adds a status to stop the process instance
stopProcess()	stops the process instance, adding no statuses

Content:

(none)

Side Effects:

The process instance is stopped. If a status parameter is present, the specified status is added to the process instance.

ApplyLogin Action

Use the ApplyLogin action to verify a user's credentials and get an authentication token from the Reactor 5 Server.

This action takes a username and password, makes an authentication request to Reactor 5 Server, then saves the authentication token in the servlet session. This is the only action that does not require a valid authentication token to already be in the servlet session.

Parameters:

username	String with the user's login name
password	String with the attempted password

ReactorProxy Methods Called:

login()	gets an authentication token

Content:

message	short text describing success or failure

State:

getAuthentication()	returns Authentication object if successful, or null if authentication failed

ApplyStartForm Action

Use the ApplyStartForm action when the user has submitted a form to initiate a process. The start form is typically created by the ShowStartForm action. This form can contain fields with values for operands to be initialized before starting the process. Each field with an operand value must be named "operand:" followed by the label of the operand.

This action takes the "id" parameter and "operand:${LABEL}" parameters and makes cloneInstance(), query(), setObjects(), and startProcess() requests to Reactor 5 Server.

Parameters:

id	String with the ID of the process definition
operand:${LABEL}	String with the value of an operand, for each operand with the label ${LABEL} (one for each operand value in the start form)

ReactorProxy Methods Called:

cloneInstance()	makes an instance of the process
query()	gets the instance and associated objects
setObjects()	sets the operands, modifies the process
startProcess()	starts the process instance

Content:

(none)

Side Effects:

process is created, configured and started

ReleaseActivity Action

The ReleaseActivity action is the opposite of the AcquireRelease action. It modifies an activity's ACL to remove the entry where the user has the "Initiative Participant" role. It also modifies all entries with the "Standby Participant" role to have the "Eligible Participant" role.

This action takes the ID of a process instance and makes an unlock() request to Reactor 5 Server, using the username from the Authentication object.

Parameters:

id	String with the ID of the process instance

ReactorProxy Methods Called:

unlock()	attempts to remove the Initiative Participant role for the user, and also move Standby Participant ACEs to have the Eligible Participant role, for the selected process instance

Content:

(none)

Side Effects:

the process instance no longer has the user as an active participant

ShowActivity Action

Use the ShowActivity action to display the details of an activity that has been assigned to the user.

This action takes the ID of a process instance, makes a query() request to Reactor 5 Server, then stores a ProcessContentBean with the activity details, an OperandListBean with the operand values, and a StatusListBean with the possible statuses that could be used to complete the activity. If the process metadata has a custom activity template, then the template is filled in and stored in a ValueBean.

Parameters:

id	String with the ID of the process instance

ReactorProxy Methods Called:

query()	to get the process instance and associated objects

Content:

processBean	ProcessContentBean
operandListBean	OperandListBean, sorted by label
statusListBean	StatusListBean, sorted by label
activityContents	ValueBean with the contents of a custom activity template (if defined), with values filled in from the process instance and operands

<u>State:</u>

isCustomTemplate()	true if activity has a custom template
getCustomJSP()	String with URL for servlet to forward to, if custom JSP is found in process metadata

ShowActivityList Action

Use the ShowActivityList action to display all activities either that are assigned to the user, or can be acquired by the user. To show the details of a particular activity, use the ShowActivity action.

This action uses the username from the Authentication object to create a ProcessListBean with all the activities in which the user has "Eligible Participant" or "Initiative Participant" roles.

<u>Parameters:</u>

(none)

<u>ReactorProxy Methods Called:</u>

query()	gets process instances and associated operands

<u>Content:</u>

processListBean	ProcessListBean object, sorted by label

ShowAttachment Action

Use the ShowAttachment action to retrieve the contents of a "File" operand which has been stored in a document repository using the AttachmentModule.

This action takes the ID of a document which has been stored as an attachment, uses the configured AttachmentModule to get metadata and contents of the document, then sends the contents of the document to the servlet's output stream. Since this class uses the servlet's output stream directly, setServlet() must be called before execute().

<u>Parameters:</u>

id	String with the attachment ID

<u>ReactorProxy Methods Called:</u>

(none)

Content:

data is sent directly to the output stream

ShowCompletionForm Action

Use the ShowCompletionForm action to create a form to complete an activity. This provides the content of a form, but does not modify the activity. If the form is empty, discovered by calling the action's isEmptyForm() method after execution, then the activity can be completed immediately using the ApplyCompletionForm action, or by using the ReactorProxy API directly.

This action takes the ID of a process instance, the name of the selected completion status, and the IDs of possible completion statuses. It makes a query() request to Reactor 5 Server, gets the process instance and its associated operands, and stores a ProcessContentBean and the selected completion status ID and label. If the process instance has a custom completion form defined in its metadata, then a ValueBean is set with the completion form. Otherwise, an OperandListBean is added with some of the operands. Operands will not be included unless their metadata specifies that they appear in completion forms.

Parameters:

id	String with the ID of the process instance
statusLabel	String with the label of the selected status
statusIdFor${LABEL}	String with the ID of a status with the label ${LABEL}, for each possible status

ReactorProxy Methods Called:

query()	gets process instance and associated objects

Content:

processBean	ProcessContentBean with the process instance data
statusId	ValueBean with the ID of the selected status
statusLabel	ValueBean with the label of the selected status

operandListBean	OperandListBean with operands to be included in the default completion form, sorted by label
formContents	ValueBean with contents of the custom form, if a custom form is defined

State:

isCustomForm()	true if a custom completion form is defined
isEmptyForm()	true if no operands are marked to be included in the completion form, and no custom form is defined

ShowProcessList Action

Use the ShowProcessList action to display a list of process definitions. To create and start an instance of a process definition, use the ShowStartForm action.

This action takes no parameters, makes a query() request to the Reactor 5 Server, then stores a ProcessListBean named "processListBean".

Parameters:

(none)

ReactorProxy Methods Called:

query()	with "must be definition" and "must be root" criteria

Content:

processListBean	ProcessListBean object, sorted by label

ShowStartForm Action

Use the ShowStartForm to display a form to configure a process to be started. The ApplyStartForm action will start the process after the form is submitted.

This action takes the "id" parameter, makes a query() request to Reactor 5 Server, then stores a ProcessContentBean named "processBean", a OperandListBean named "operandListBean". If the process has a custom start form in its "Portal: Start Form" metadata, then the a ValueBean named "formContents" is also stored in the content.

Parameters:

id	String with the ID of the process definition

ReactorProxy Methods Called:

query()	gets process with associated operands and statuses

Content:

processBean	ProcessContentBean with process info
operandListBean	OperandListBean, sorted by label
formContents	OperandListBean, sorted by label

State:

isCustomForm()	true if custom start form was found

Attachments

The ServletFacade class provides the parseParameters() method, which creates a Map object with key/value pairs for each parameter. This method also handles forms with an encoding type of "multipart/form-data". Files that are uploaded as part of multipart forms are converted into Attachment objects, which can then be stored in a document repository using classes in the com.oakgrovesystems.attachments package. Attachments can be retrieved by users through servlets that use the ShowAttachment action. The ShowAttachment action takes the ID of an Attachment stored in a document repository and sends the file contents through the servlet output stream to the web browser.

Writing Servlet Code

Portal servlets should be very simple. Action objects are responsible for business logic, and content beans are responsible for formatting data for display.

Each portal servlet extends the PortalServlet class and implements the service(HttpServletRequest, HttpServletResponse) method. The following import statements are required:

```
import com.oakgrovesystems.portal.Authentication;
import com.oakgrovesystems.portal.PortalException;
import com.oakgrovesystems.portal.PortalServlet;
import com.oakgrovesystems.portal.ServletFacade;
import com.oakgrovesystems.portal.action.*; // select appropriate actions
import javax.servlet.ServletException;
```

```
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
```

The service() method should create a ServletFacade object for accessing the Java Servlet API. This makes it easier to unit test the portal servlet in isolation, and can also be more convenient.

```
ServletFacade servlet;
servlet = new ServletFacade(request, response, getServletContext());
```

Use the ServletFacade object to get an authentication object from the session. This assumes that there is a login servlet that puts the authentication in a standard location.

```
String authKey = "reactor.authentication";
Authentication auth;
auth = (Authentication) servlet.getSessionAttribute(authKey);
```

Check that the authentication object exists and is valid. If no valid authentication object is found, then display the login page or an appropriate error. Data in the session may expire if not used for some time, so a missing authentication object is not necessarily an unusual event.

```
String loginFormJSP = "/Login";
if ((auth == null) || (!auth.isValid())) {
      servlet.forward(loginFormJSP);
}
```

Once the authentication object is found, get the authentication token and pass it to the ReactorProxy object named "reactorProxy", defined in the PortalServlet class. This ReactorProxy object will be used by the action object to communicate with the Reactor 5 Server.

```
reactorProxy.setAuthToken(auth.getToken());
```

Create an action object to process parameters and generate content. See Section Chapter 65536 “Actions” for descriptions of action classes that are included with Reactor 5 Portal Framework, or create your own action classes.

```
ApplyCompletionForm action;
action = new ApplyCompletionForm();
```

Initialize the action object. This includes passing the ReactorProxy object and the authentication object. If there are parameters, the parameters should be parsed and passed to the action object. It is important that a servlet that parses parameters should not forward to another servlet that also parses parameters! This could result in the second servlet waiting forever to read data from an input stream that has already been completely read.

```
action.setServerProxy(reactorProxy);
action.setAuthentication(auth);
action.setParameters(servlet.parseParameters());
```

Execute the action, and handle any PortalException object which might be thrown.

```
String errorKey = "reactor.error";
String errorJSP = "/Error";
try {
      action.execute();
} catch (PortalException e) {
```

```
        servlet.setRequestAttribute(errorKey, e);
        servlet.forward(errorJSP);
        return;
}
```

Choose a JSP page to forward or redirect to for display. Forward to display pages which depend on content beans being passed in the request context. Redirect to servlets or displays that do not depend on any parameters, especially after processing a submitted form. Be careful not to forward to another servlet that processes the input stream from a servlet that also processes the input stream.

```
String activityListRedirect = "ActivityList";
servlet.sendRedirect(activityListRedirect);
```

Writing JSP Pages

Each display should have its own JSP page. The JSP page might use the JSP tag "<%@ include file="..." %>" to include shared elements, such as headers or footers. The beginning of the JSP page should import the appropriate Java classes and set the page's "session" value to "true". The JSP page must also define any tag libraries and declare any beans used in the page.

```
<%@ page language="java"
        session="true"
        import="com.oakgrovesystems.portal.content.*,java.util.*"
        errorPage="Error.jsp" %>
<%@ taglib uri="/WEB-INF/struts-bean.tld"  prefix="bean" %>
<%@ taglib uri="/WEB-INF/struts-logic.tld" prefix="logic" %>
<jsp:useBean id="processBean" scope="request"
        class="com.oakgrovesystems.portal.content.ProcessContentBean"/>
<jsp:useBean id="operandListBean" scope="request"
        class="com.oakgrovesystems.portal.content.OperandListBean"/>
```

To display a value stored in a simple content bean, use the <jsp:getProperty> tag.

```
<jsp:getProperty name="processBean" property="label"/>
```

To iterate through a list of beans, use the Apache Struts tags.

```
        <logic:iterate name="operandListBean"
property="operands"
id="operandBean"
scope="request"
type="com.oakgrovesystems.portal.content.OperandContentBean">
                <li> <bean:write name="operandBean" property="label"/>
        </logic:iterate>
```

To conditionally include HTML depending on the value of a particular bean's property, use the Apache Struts <logic:equal> and <logic:notEqual> tags.

```
<logic:notEqual name="operandBean" property="dataType" value="File">
        <bean:write name="operandBean" property="value"/>
</logic:notEqual>

<logic:equal name="operandBean" property="dataType" value="File">
        <a href="GetAttachment?id=<bean:write name='operandBean'
                property='value'/>">Attachment</a>
</logic:equal>
```

When the contents of a form might include uploaded files, remember to use the multipart form encoding type.

```
<form action="ApplyForm" method="POST" enctype="multipart/form-data">
```

Configuring a Web Application

Each servlet and JSP page needs to be configured in the servlet's deployment descriptor. The deployment descriptor is an XML document named "web.xml" located in the WEB-INF directory of the web application. This section describes the minimal subset of the deployment descriptor required to deploy a portal web application.

The format of the deployment descriptor is described by a DTD published by Sun Microsystems. This how a valid servlet deployment descriptor begins:

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

<!DOCTYPE web-app PUBLIC
      "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
      "http://java.sun.com/j2ee/dtds/web-app_2.2.dtd">
```

The root element of the document is "web-app".

```
<web-app>
```

The next group of elements is "servlet" elements. Each one describes either a servlet or a JSP page. Both use the "servlet-name" element to assign a unique name to the servlet. The elements for JSP pages contain a "jsp-file" element, but "servlet-class" elements are used for servlets.

```
<servlet>
      <servlet-name>ProcessList<servlet-name>
      <servlet-class>activechecklist.ProcessList<servlet-class>
</servlet>

<servlet>
      <servlet-name>ProcessListJSP<servlet-name>
      <jsp-file>/ProcessList.jsp<jsp-file>
</servlet>
```

After adding all of the "servlet" elements, add a "servlet-mapping" element for each "servlet" element. While it might be more convenient to place each "servlet-mapping" element immediately after its corresponding "servlet" element in the file, that would violate the file format described in the DTD. Each "servlet-mapping" element contains a "servlet-name" element using the name assigned previously in the file. Then it contains a "url-pattern" element with a relative path. This assigns a URL to the servlet or JSP page, relative to the root URL of the web application.

```
<servlet-mapping>
```

```
        <servlet-name>ProcessList</servlet-name>
        <url-pattern>/ProcessList</url-pattern>
</servlet-mapping>

<servlet-mapping>
        <servlet-name>ProcessListJSP</servlet-name>
        <url-pattern>/ProcessListJSP</url-pattern>
</servlet-mapping>
```

After all the servlet mapping elements, declare any custom tag libraries used by the JSP pages.

```
<taglib>
        <taglib-uri>/WEB-INF/struts.tld</taglib-uri>
        <taglib-location>/WEB-INF/struts.tld</taglib-location>
</taglib>
```

Close the tag of the root element, and the document is finished.

```
</web-app>
```

Building a Web Application

A Portal web application consists of HTML pages, images, JSP pages, Servlet classes, JAR files, and a deployment descriptor. To build a web application, create a directory to store all these things.

Copy the HTML pages, images, and JSP pages into the directory. Create a subdirectory named "WEB-INF". Create a "classes" subdirectory of "WEB-INF". Compile the servlet classes, possibly with ancillary supporting classes, and copy them into the "WEB-INF/classes" directory. Make sure that the directory structure under "classes" mirrors the package structure of the compiled classes. For example, the com.oakgrovesystems.example.SampleServlet class file would go into the "WEB-INF/classes/com/oakgrovesystems/example/SampleServlet.class" file. If there are JAR files required by the web application that are not in the application server's default classpath, then create a "lib" subdirectory in "WEB-INF". Copy the required JAR files into the "WEB-INF/lib" directory. Copy the "web.xml" deployment descriptor file into the "WEB-INF" directory. Create a WAR file using the same tool that you use to create JAR files, like the "jar" tool that is part of Sun's JDK. Use the ".war" filename extension instead of ".jar". The parent of the "WEB-INF" directory should not be included in the WAR file. The JSP pages and "WEB-INF" directories should be at the top level of the WAR file's contents.

See your application server's documentation for instructions to deploy a WAR file. To deploy a WAR file in the JBoss application server that is bundled with some evaluation copies of the Reactor 5 Server, just copy the WAR file into the "jboss321/server/default/deploy" directory.

Sample Web Application

The Reactor 5 Server distribution includes a sample web application in the "..\samples\portal\src\portal\activechecklist" directory. The servlet source code is located under the "src" directory. The JSP pages, HTML, images, and deployment descriptor are under the "webapp" directory. The build scripts use Ant, from the Apache Jakarta Project. The "build.xml" file contains the build configuration, and the "build.bat" and "build.sh" scripts execute the build on Microsoft Windows and UNIX respectively.

More information about Ant is available here:

http://jakarta.apache.org/ant/

The build scripts create an "ActiveChecklist.war" file in the "dist" directory.

Reactor Java API

This section describes the highlights of the Java API for Reactor. Refer to the Javadoc HTML pages for specific details about arguments, return values, and exceptions. The Javadoc HTML pages are distributed with the Reactor 5 Server, in this directory:

```
docs/javadocs/index.html
```

The ReactorProxy, ReactorQuery, and ReactorObjects classes are the most important classes for accessing process mediation objects and communicating with Reactor 5 Server.

ReactorProxy

The ReactorProxy interface is found in the com.oakgrovesystems.reactor.client package. It is implemented by two classes in the same package, EJBReactorProxy and XMLReactorProxy. They provide the following methods:

```
login()
logout()

setAuthToken()
getAuthToken()

createObjects()
setObjects()
delete()

cloneInstance()
startProcess()
stopProcess()

get()
query()

lock()
unlock()

addStatusToProcess()
removeStatusFromProcess()
```

ReactorProxy login()

The login() method takes a username and password as arguments, and returns an authentication token. This authentication token is also stored by the ReactorProxy object for use in subsequent method calls. Either login() or setAuthToken() must be called successfully before other methods will be able to succeed. Reactor 5 Server requires a valid authentication token with every request, except login requests.

ReactorProxy logout()

The logout() method invalidates the authentication token set by a previous call to login() or setAuthToken().

ReactorProxy setAuthToken()

The setAuthToken() method provides a way to pass an authentication token directly to the ReactorProxy object. This is useful when the token has been acquired previously and stored as session data.

ReactorProxy getAuthToken()

The getAuthToken() method returns the authentication token set by a previous call to login() or setAuthToken(). This is useful for getting the token to be stored as session data.

ReactorProxy createObjects()

The createObjects() method sends a set of objects to Reactor 5 Server to be added to the workflow engine. It will fail if an object in the set has the same ID as an object already known to the workflow engine. To modify existing objects, use the setObjects() method.

ReactorProxy setObjects()

The setObjects() method sends a set of objects to Reactor 5 Server to be updated in the workflow engine. The objects can be either new or modified; they do not have to be previously known to the workflow engine.

ReactorProxy delete()

The delete() method removes objects from the workflow engine. When a process is deleted, all the associated objects are also deleted.

ReactorProxy cloneInstance()

Before a process can be started, it must be cloned. This means that a copy is made of a process definition, along with all its subprocesses and associated objects. The only difference between the process definition and the process instance is the value of the isDefinition property in the process.

The cloned instance begins in an unstarted state. This provides the opportunity to call setObjects() to configure the process instance. This might include adding operands, setting operand values, and setting ACLs.

ReactorProxy startProcess()

Once a process instance has been created, the startProcess() method starts the process instance. This event may trigger the execution of policies. The new state of the process may also trigger changes by causing some conditions to become true.

ReactorProxy stopProcess()

The stopProcess() method causes a running process instance to stop. This event may trigger the execution of policies. The new state of the process may also trigger changes by causing some conditions to become true.

ReactorProxy get()

The get() method retrieves one object from Reactor 5 Server. Either the label path or object ID must be known. Use the query() method to retrieve multiple objects, or when an object's exact identity is not known.

ReactorProxy query()

The query() method retrieves objects from the Reactor 5 Server based on some criteria. A QueryCriteria object defines the specific criteria for which objects to return, as well as what associated objects to return. The results are stored in a ReactorObjects object.

ReactorProxy lock()

The lock() method attempts to acquire a role in an Access Control Element (ACE) that is not held by any other user or group. The method fails if the role is already held. The lock() method optionally takes arguments for eligible and standby roles. All elements with the eligible role are modified to have the standby role. Another optional argument is a status that to be added to the process if the lock succeeds. If the specified object is not a process, then the status will be added to the object's associated process.

The lock() method is useful in situations where only one person should be able to edit a process definition or work on an activity in a running process instance.

ReactorProxy unlock()

The unlock() method reverses the effects of a lock() method. It attempts to remove a role in an Access Control Element. It can also modify entries with a specified standby role to change to an eligible role. If a status is specified, it will remove the status from the process. If the object is not a process, it will attempt to remove the status from the object's associated process.

ReactorProxy addStatusToProcess()

The addStatusToProcess() method adds a status to the set of current statuses for a process. This event may trigger the execution of policies. The change in status of the process may also trigger changes by causing some conditions to become true.

ReactorProxy removeStatusFromProcess()

The removeStatusFromProcess() method removes a status from the set of current statuses for a process. This event may trigger the execution of policies. The change in status of the process may also trigger changes by causing some conditions to become true.

QueryCriteria

The QueryCriteria class is found in the com.oakgrovesystems.reactor.client package. It is used to retrieve processes and their associated objects. Some properties of the criteria identify characteristics that must be matched by retrieved processes. Other properties specify which associated objects to retrieve, and how many levels of subprocesses to retrieve. Each property has accessor methods to set and get the property. The methods to set properties are listed below.

```
setProcessId()
setProcessLabelPath()

setACE()
setStateEquals()
setMustBeDefinition()
setMustBeInstance()
setMustBeRoot()
setHasStatusId()
setHasStatusLabel()

setHasParentId()
setHasParentLabel()
setHasAssociatedSubprocessId()
setHasAssociatedSubprocessLabel()
setHasAssociatedOperandId()
setHasAssociatedOperandLabel()
setHasAssociatedStatusId()
setHasAssociatedStatusLabel()
setHasAssociatedPolicyId()
setHasAssociatedPolicyLabel()

setDepth()
setIncludeOperands()
setIncludeParent()
setIncludePolicies()
setIncludeStatuses()
```

QueryCriteria setProcessId()

This method sets the ID of a particular process. The query will retrieve that process, along with whatever subprocesses and associated objects are specified in further properties. The setProcessLabelPath() method can be used when the ID of the process is not known.

QueryCriteria setProcessLabelPath()

This method sets the label path identifying of a particular process or group of processes. The query will retrieve those processes, along with whatever subprocesses and associated objects are specified in further properties. The setProcessId() method can be used when the ID of the process is known.

QueryCriteria setACE()

This method restricts the set of retrieved processes to those with an ACL containing the specified ACE. For example, this is useful for querying all the process instances where the user is a participant.

QueryCriteria setStateEquals()

This method sets the state that retrieved processes must be in. This is useful for restricting the query to processes that are currently running.

QueryCriteria setMustBeDefinition()

This method restricts the query to retrieve only process definitions, and not process instances.

QueryCriteria setMustBeInstance()

This method restricts the query to retrieve only process instances, and not process definitions.

QueryCriteria setMustBeRoot()

This method indicates that the retrieved processes must not have any parents.

This is useful when querying process definitions where the subprocesses should be filtered out.

QueryCriteria setHasStatusId()

This method specifies a status that must be in the set of current statuses for retrieved processes. The setHasStatusLabel() method can be used when the ID of the status is not known, or when any status with a particular label is sufficient. Use setHasAssociatedStatusId() to specify an associated status rather than a current status.

QueryCriteria setHasStatusLabel()

This method specifies a status that must be in the set of current statuses for retrieved processes. The setHasStatusId() method can be used when only a specific status with a known ID will satisfy the criteria. Use setHasAssociatedStatusLabel() to specify an associated status rather than a current status.

QueryCriteria setHasParentId()

This method specifies a process that must be the parent of every retrieved process. The setHasParentLabel() method can be used when the ID of the parent process is not known, or when any process with a certain label is sufficient.

QueryCriteria setHasParentLabel()

This method specifies a process that must be the parent of every retrieved process. The setHasParentId() method can be used when only a specific process with a known ID will satisfy the criteria.

QueryCriteria setHasAssociatedSubprocessId()

This method indicates that the retrieved processes must all have a subprocess with the specified ID. The setHasAssociatedSubprocessLabel() method can be used when the ID of the subprocess is not known, or when any subprocess with a certain label is sufficient.

QueryCriteria setHasAssociatedSubprocessLabel()

This method indicates that the retrieved processes must all have a subprocess with the specified label. The setHasAssociatedSubprocessId() method can be used when only a specific process with a known ID will satisfy the criteria.

QueryCriteria setHasAssociatedOperandId()

This method indicates that the retrieved processes must all have an operand with the specified ID. The setHasAssociatedOperandLabel() method can be used when the ID of the operand is not known, or when any operand with a certain label is sufficient.

QueryCriteria setHasAssociatedOperandLabel()

This method indicates that the retrieved processes must all have an operand with the specified label. The setHasAssociatedOperandId() method can be used when only a specific operand with a known ID will satisfy the criteria.

QueryCriteria setHasAssociatedStatusId()

This method indicates that the retrieved processes must all have an operand with the specified ID. The setHasAssociatedOperandLabel() method can be used when the ID of the operand is not known, or when any operand with a certain label is sufficient. Use setHasStatusId() to specify a status in the set of current statuses for a process, rather than just an associated status.

QueryCriteria setHasAssociatedStatusLabel()

This method indicates that the retrieved processes must all have a status with the specified label. The setHasAssociatedStatusId() method can be used when only a specific operand with a known ID will satisfy the criteria. Use setHasStatusLabel() to specify a status in the set of current statuses for a process, rather than just an associated status.

QueryCriteria setHasAssociatedPolicyId()

This method indicates that the retrieved processes must all have a policy with the specified ID. The setHasAssociatedPolicyLabel() method can be used when the ID of the policy is not known, or when any policy with a certain label is sufficient.

QueryCriteria setHasAssociatedPolicyLabel()

This method indicates that the retrieved processes must all have a policy with the specified label. The setHasAssociatedPolicyId() method can be used when only a specific policy with a known ID will satisfy the criteria.

QueryCriteria setDepth()

This method specifies the number of levels of subprocesses to include with each retrieved process that meets the query criteria. If a depth of 0 is specified, then no subprocesses will be retrieved. If a depth of -1 is specified, then all levels of subprocesses will be retrieved. The default is for no subprocesses to be retrieved, if the setDepth() method for a QueryCriteria object is not called.

QueryCriteria setIncludeParent()

This method specifies whether a query should return the parent process of each process meeting the query criteria.

QueryCriteria setIncludeOperands()

This method specifies whether the operands associated with each retrieved process should also be retrieved. Note that this does not include operands that are visible to a process, but associated with a parent or ancestor of the process.

QueryCriteria setIncludePolicies()

This method specifies whether the policies associated with each retrieved process should also be retrieved.

QueryCriteria setIncludeStatuses()

This method specifies whether the associated statuses and current statuses for each retrieved process should also be retrieved.

ReactorObjects

The ReactorObjects class is found in the com.oakgrovesystems.reactor package. It manages a collection of reactor objects that have been either retrieved from Reactor 5 Server or created directly by the application. Relationships between Reactor process mediation objects are maintained by indirect references, so the relationship can be preserved without retrieving the object itself from Reactor 5 Server.

The following methods manage the collection and provide access to process mediation objects:

```
add()
remove()
clear()
size()

getObjects()

getProcess()
getProcesses()
getParent()
getSubprocess()
getSubprocesses()
getRootProcesses()

getOperand()
getOperands()

getCurrentStatus()
getCurrentStatuses()
getStatus()
getStatuses()

getPolicies()
getPolicy()
```

ReactorObjects add()

This method adds an object to the collection.

ReactorObjects remove()

This method returns an object from the collection.

ReactorObjects clear()

This method removes all objects from the collection.

ReactorObjects size()

This method returns the number of objects in the collection.

ReactorObjects getObjects()

This method returns a Collection object containing all the objects.

ReactorObjects getProcess()

Depending on its arguments, this method returns a process associated with the specified object, or identified by the specified ID or label path.

ReactorObjects getProcesses()

This method returns a Set object holding all processes managed by the ReactorObjects object.

ReactorObjects getParent()

This method returns the parent process of the specified process object.

ReactorObjects getSubprocess()

When passed a Process object, this method returns a subprocess with the specified label.

ReactorObjects getSubprocesses()

This method returns a Set object containing all the subprocesses of a specified process.

ReactorObjects getRootProcesses()

This method returns a Set object containing all the processes that have no parent process.

ReactorObjects getOperand()

Depending on its arguments, this method returns an operand associated with the specified process, or identified by the specified ID or label path.

ReactorObjects getOperands()

This method returns a Set object containing the operands associated with the specified process.

ReactorObjects getCurrentStatus()

This method takes a Process object and the label of a Status object. If one of the current statuses of the process has the specified label, then that Status object is returned. Otherwise, the method returns null.

ReactorObjects getCurrentStatuses()

This method returns a Set object containing the current statuses for the specified process. Use the getStatuses() method to get the set of associated statuses.

ReactorObjects getStatus()

Depending on its arguments, this method returns a status associated with the specified process, or identified by the specified ID or label path.

ReactorObjects getStatuses()

This method returns a Set object containing the statuses associated with the specified process. Use getCurrentStatuses() to get the set of current statuses.

ReactorObjects getPolicy()

Depending on its arguments, this method returns a policy associated with the specified process, or identified by the specified ID or label path.

ReactorObjects getPolicies()

This method returns a Set object containing the policies associated with the specified process.

Request Types

These are the request types handled by the Reactor 5 Server:

- AddStatus
- CloneInstance
- Create
- Delete
- Get
- Lock
- Login
- Logout

- QueryAllObjects
- QueryProcessTree
- QueryProcesses
- RemoveStatus
- SetObjects
- Start
- Stop
- Unlock

AddStatus

The AddStatus request adds a Status object to the "current statuses" attribute of a Process. This is useful for triggering preconditions, change conditions, and policy execution. The Status change could also serve just to provide information, and not cause any action.

This request takes the following parameters:

Process ID	the process acquiring the new status
Process Label Path	the process acquiring the new status
Status ID	the status being added
Status Label Path	the status being added

Either the Process ID or Process Label Path may be specified, but not both. Setting one causes the other to become unset.

Either the Status ID or Status Label Path may be specified, but not both. Setting one causes the other to become unset.

The response to this request contains a result code, but does not contain any return values.

Sample XML request:

```
<?xml version="1.0" encoding="UTF-8"?>
<reactor_request>
<request_type>process_command_add_status</request_type>
<authentication_token>${authentication_token}</authentication_token>
<parameters>
  <process>
    <reactor_common:object_reference
```

```
                xmlns:reactor_common="reactor_common.dtd">
        <reactor_common:id>${process_id}</reactor_common:id>
      </reactor_common:object_reference>
    </process>
    <status>
      <reactor_common:object_reference
                xmlns:reactor_common="reactor_common.dtd">
        <reactor_common:id>${status_id}</reactor_common:id>
      </reactor_common:object_reference>
    </status>
  </parameters>
  </reactor_request>
```

Sample XML response:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
  <result_code>
    <code_number>200</code_number>
    <user_message>OK</user_message>
    <debug_info />
  </result_code>
  <return_values />
</response>
```

CloneInstance

The CloneInstance request makes a copy of a process definition and changes it to be a process instance. The process instance begins in an unstarted state, to allow for Operand values to be set before starting. Use the Start request to start the process instance.

This request takes the following parameters:

Process ID - the process acquiring the new status

Process Label Path - the process acquiring the new status

Clone Label - specifies a new label to give to the cloned process

Either the Process ID or Process Label Path may be specified, but not both. Setting one causes the other to become unset.

If the Clone Label is not specified, the cloned process keeps the same label as the original. The Clone Label parameter is useful for giving each process instance a label path that does not need to be relative to any object ID.

The response to this request contains a result code. If successful, the response also contains the following return value:

Clone ID - the ID of the new object resulting from the clone

Sample XML request:

```
<?xml version="1.0" encoding="UTF-8"?>
<reactor_request>
  <request_type>process_command_clone_instance</request_type>
  <authentication_token>${authentication_token}</authentication_token>
  <parameters>
    <reactor_common:object_reference
              xmlns:reactor_common="reactor_common.dtd">
      <reactor_common:id>${process_id}</reactor_common:id>
    </reactor_common:object_reference>
  </parameters>
</reactor_request>
```

Sample XML response:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
  <result_code>
    <code_number>200</code_number>
    <user_message>OK</user_message>
    <debug_info />
  </result_code>
  <return_values>
    <reactor_common:object_reference
              xmlns:reactor_common="reactor_common.dtd">
      <reactor_common:id>${instance_id}</reactor_common:id>
    </reactor_common:object_reference>
  </return_values>
</response>
```

Create

Use the Create request to add objects that do not already exist. Use the SetObjects request to modify existing objects or to set values of objects that might not exist.

This request takes the following parameters:

Objects - a set of Reactor objects

Use Label Paths - whether request should specify objects with

label paths instead of object IDs

If not specified, the request will not use label paths.

An application may need the request to use label paths instead of object IDs because the Reactor 5 Server will assign new IDs to created objects. This only an issue if the application stores and continues to use the IDs of new objects after sending the request.

The response to this request contains a result code, but does not contain any return values.

Sample XML request:

```
<?xml version="1.0" encoding="UTF-8"?>
<reactor_request>
  <request_type>process_command_create</request_type>
  <authentication_token>${authentication_token}</authentication_token>
  <parameters>
    <objects>
      <operand:operand xmlns:operand="operand.dtd">
        ...
      </operand:operand>
      <operand:operand xmlns:operand="operand.dtd">
        ...
      </operand:operand>
      <operand:operand xmlns:operand="operand.dtd">
        ...
      </operand:operand>
    </objects>
  </parameters>
</reactor_request>
```

Sample XML response:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
  <result_code>
    <code_number>200</code_number>
    <user_message>OK</user_message>
    <debug_info />
  </result_code>
  <return_values />
</response>
```

Delete

The Delete request removes an object from Reactor 5. Deleting a process also deletes all associated objects, with the exception of the parent process, but including all levels of sub-processes and their associated objects.

This request takes the following parameters:

ID - specifies the object to delete

Label Path - specifies the object to delete

Type - specifies the type of the object to delete

Delete Definitions Only - whether to only delete Process definitions

Either the ID or Label Path may be specified, but not both. Setting one causes the other to become unset.

The Type should specify a Process, Operand, Status, Policy, or Object.

The "Delete Definitions Only" parameter is used in the case where a label path is used to identify the object to be deleted and the label path specified resolves to exactly one Process definition and at least one other Process instance. Without setting this parameter's value to "true", the default is for Reactor to fail if a label path matches multiple objects.

The response to this request contains a result code, but does not contain any return values.

Sample XML request:

```
<?xml version="1.0" encoding="UTF-8"?>
<reactor_request>
  <request_type>process_command_delete</request_type>
  <authentication_token>${authentication_token}</authentication_token>
  <parameters>
    <type>process</type>
    <reactor_common:object_reference
              xmlns:reactor_common="reactor_common.dtd">
      <reactor_common:id>${operand_id}</reactor_common:id>
    </reactor_common:object_reference>
  </parameters>
</reactor_request>
```

Sample XML response:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
  <result_code>
    <code_number>200</code_number>
    <user_message>OK</user_message>
    <debug_info />
  </result_code>
  <return_values />
</response>
```

Get

The Get request retrieves one object from Reactor. Use the query requests to retrieve multiple objects.

This request takes the following parameters:

ID - specifies the object to acquire

Label Path - specifies the object to acquire

Type - specifies the type of the object to acquire

Either the ID or Label Path may be specified, but not both. Setting one causes the other to become unset.

The Type should specify a Process, Operand, Status, Policy, or Object.

The response to this request contains a result code. If successful, the response also contains the following return value:

Object - the requested Reactor 5 object

To get more than one object, use one of the query requests.

Sample XML request:

```
<?xml version="1.0" encoding="UTF-8"?>
<reactor_request>
  <request_type>process_object_get</request_type>
  <authentication_token>${authentication_token}</authentication_token>
  <parameters>
    <type>operand</type>
    <reactor_common:object_reference
              xmlns:reactor_common="reactor_common.dtd">
      <reactor_common:id>${operand_id}</reactor_common:id>
    </reactor_common:object_reference>
  </parameters>
</reactor_request>
```

Sample XML response:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
  <result_code>
    <code_number>200</code_number>
    <user_message>OK</user_message>
    <debug_info />
  </result_code>
  <return_values>
    <object>
```

```
        <operand:operand xmlns:operand="operand.dtd">
          ...
        </operand:operand>
      </object>
    </return_values>
  </response>
```

Lock

The Lock request acquires a role in the ACL of an element. If the role is already present in the ACL, the request is denied. The request can optionally take the name of an "eligible" role and the name of a "standby" role. If present, elements with the "eligible" role are modified to have the "standby" role. The request can also take a status. This will add a status to the process being locked, or to the associated process if the object is not a process.

This request takes the following parameters:

```
Object ID - specifies the object to lock
Object Label Path - specifies the object to lock
ACE - the ACE to create
Eligible Role - the role for ACEs which could have acquired the lock
Standby Role - the role for ACEs which can no longer acquire the lock
Status ID - specifies the status to add
Status Label Path - specifies the status to add
```

The Object ID and Object Label Path cannot both be specified. Similarly, the Status ID and Status Label Path cannot both be specified.

Sample XML request:

```
<?xml version="1.0" encoding="UTF-8"?>
<reactor_request xmlns:reactor_common="reactor_common.dtd">
  <request_type>
    process_command_lock
  </request_type>
  <authentication_token>
    ${authentication_token}
  </authentication_token>
  <parameters>
    <reactor_common:object_reference>
      <reactor_common:id>${id}</reactor_common:id>
    </reactor_common:object_reference>
    <reactor_common:ace>
      <reactor_common:profile>
        <reactor_common:user>${user}</reactor_common:user>
      </reactor_common:profile>
      <reactor_common:role>${lock_role}</reactor_common:role>
    </reactor_common:ace>
    <eligible_role>${eligible_role}</eligible_role>
    <standby_role>${standby_role}</standby_role>
  </parameters>
</reactor_request>
```

Sample XML response:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<response>
  <result_code>
    <code_number>200</code_number>
    <user_message>OK</user_message>
    <debug_info />
  </result_code>
  <return_values />
</response>
```

Login

The Login request gets an authentication token which will be required for all subsequent requests in a session.

This request takes the following parameters:

Username - username of the person or entity logging in

Password - password of the person or entity logging in

Both the username and password must be specified.

The response to this request contains a result code. If successful, the response also contains the following return values:

Authentication Token - for authenticating in subsequent requests

Token Expiration - specification of when the token expires, if ever

Sample XML request:

```
<?xml version="1.0" encoding="UTF-8"?>
<reactor_request>
  <request_type>authentication_login</request_type>
  <parameters>
    <login>
      <username>${username}</username>
      <password>${password}</password>
    </login>
  </parameters>
</reactor_request>
```

Note that Reactor does not trim leading and trailing whitespace around the password, since whitespace may be significant in the password. This is an exception, since whitespace around element content is usually trimmed.

Sample XML response:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<response>
  <result_code>
    <code_number>200</code_number>
    <user_message>OK</user_message>
    <debug_info />
  </result_code>
  <return_values>
    <authentication_token>${auth_token}</authentication_token>
    <token_expiration>never</token_expiration>
  </return_values>
</response>
```

Logout

The Logout request relinquishes an authentication token, ending a session.

This request takes the following parameters:

Token to Expire - the authentication token to log out

Note that the Logout request does not take the authentication token from the request itself, but from the parameter.

The response to this request contains a result code, but does not contain any return values.

Sample XML request:

```
<?xml version="1.0" encoding="UTF-8"?>
<reactor_request>
  <request_type>authentication_logout</request_type>
  <authentication_token>${authentication_token}</authentication_token>
  <parameters>
    <authentication_token>
      ${authentication_token}
    </authentication_token>
  </parameters>
</reactor_request>
```

Sample XML response:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
  <result_code>
    <code_number>200</code_number>
    <user_message>OK</user_message>
    <debug_info />
  </result_code>
  <return_values />
</response>
```

QueryAllObjects

The QueryAllObjects request is useful for archiving or exporting the entire Reactor 5 data set.

It takes one optional parameter:

Use Label Paths - XML response will use label paths instead of

IDs for object references

The response to this request contains a result code. If successful, the response also contains the following return value:

Objects - set of all Reactor objects

Sample XML request:

```
<?xml version="1.0" encoding="UTF-8"?>
<reactor_request>
  <request_type>process_object_query</request_type>
  <authentication_token>${authentication_token}</authentication_token>
  <parameters>
    <query>
      <all_objects />
    </query>
  </parameters>
</reactor_request>
```

Sample XML response:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
  <result_code>
    <code_number>200</code_number>
    <user_message>OK</user_message>
    <debug_info />
  </result_code>
  <return_values>
    <objects>
      <process:process xmlns:process="process.dtd">
        ...
      </process:process>
      ...
    </objects>
  </return_values>
</response>
```

QueryProcessTree

The QueryProcessTree request is useful for getting details about a specific process and its associated objects.

This request takes the following parameters:

ID - object ID of the Process

Label Path - label path of the Process

Definition Tree Only - true if label path applies only to definitions

Depth - levels of sub-processes to include

(-1 for all levels, 0 for no sub-processes)

Include Superprocess - whether parent process should be returned

Include Operands - whether operands should be returned

Include Statuses - whether statuses should be returned

Include Policies - whether policies should be returned

Either the ID or Label Path may be specified, but not both. Setting one causes the other to become unset.

The "Definition Tree Only" parameter is optional. It is used in the case where a label path is used to identify the process and the label path resolves to exactly one Process definition and at least one other Process instance. Without setting this parameter's value to "true", the default is for Reactor 5 to fail if a label path matches multiple objects.

The response to this request contains a result code. If successful, the response also contains the following return value:

Objects - specified Reactor objects

Sample XML request:

```
<?xml version="1.0" encoding="UTF-8"?>
  <reactor_request>
    <request_type>process_object_query</request_type>
    <authentication_token>${authentication_token}</authentication_token>
    <parameters>
      <query>
        <process_tree>
          <depth>-1</depth>
          <reactor_common:object_reference
                    xmlns:reactor_common="reactor_common.dtd">
            <reactor_common:id xmlns:reactor_common="reactor_common.dtd">
              ...
            </reactor_common:id>
          </reactor_common:object_reference>
          <getOperands/>
          <getStatuses/>
          <getPolicies/>
        </process_tree>
      </query>
    </parameters>
  </reactor_request>
```

Sample XML response:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
  <result_code>
    <code_number>200</code_number>
    <user_message>OK</user_message>
    <debug_info />
  </result_code>
  <return_values>
    <objects>
      <process:process xmlns:process="process.dtd">
        ...
      </process:process>
      ...
    </objects>
  </return_values>
</response>
```

QueryProcesses

The QueryProcesses request is useful for getting a set of processes where a user or group is assigned a particular role.

This request takes the following parameters:

ACE Profile **-** describes element in ACL of every Process to be returned

Must Be Definition **-** whether to return only process definitions

Must Be Instance **-** whether to return only process instances

Must Be Started **-** whether to return only started process instances

Only IDs **-** whether to return the IDs of the found processes instead of objects

The parameters specifying the type of process must be consistently set. For example, "Must Be Definition" cannot be true when either "Must Be Instance" or "Must Be Started" is true.

To find every process where a user holds a certain role: set the ACE Profile type to user, set the user name, and set the role name. Similar approaches allow queries for processes where a title or group holds a certain role.

To find all process definitions, leave the ACE Profile unspecified and set "Must Be Definition" to true.

The response to this request contains a result code. If successful, the response also contains the following return value:

Objects - specified Reactor objects

Sample XML request:

```
<?xml version="1.0" encoding="UTF-8"?>
<reactor_request>
  <request_type>process_object_query</request_type>
  <authentication_token>${authentication_token}</authentication_token>
  <parameters>
    <query>
      <processes>
        <started_instances />
        <ace_pattern>
          <principal_type>user</principal_type>
          <principal>bob</principal>
          <role>Initiative Participant</role>
        </ace_pattern>
      </processes>
    </query>
  </parameters>
</reactor_request>
```

Sample XML response:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
  <result_code>
    <code_number>200</code_number>
    <user_message>OK</user_message>
    <debug_info />
  </result_code>
  <return_values>
    <objects>
      <process:process xmlns:process="process.dtd">
        ...
      </process:process>
      ...
    </objects>
  </return_values>
</response>
```

RemoveStatus

The RemoveStatus request removes a Status object from the "current statuses" attribute of a Process. This is useful for triggering preconditions, change conditions, and policy execution. The Status change could also serve just to provide information, and not cause any action.

This request takes the following parameters:

Process ID - the process acquiring the new status

Process Label Path - the process acquiring the new status

Status ID - the status being added

Status Label Path - the status being added

Either the Process ID or Process Label Path may be specified, but not both. Setting one causes the other to become unset.

Either the Status ID or Status Label Path may be specified, but not both. Setting one causes the other to become unset.

The response to this request contains a result code, but does not contain any return values.

Sample XML request:

```
<?xml version="1.0" encoding="UTF-8"?>
<reactor_request>
<request_type>process_command_remove_status</request_type>
<authentication_token>${authentication_token}</authentication_token>
<parameters>
  <process>
    <reactor_common:object_reference
              xmlns:reactor_common="reactor_common.dtd">
      <reactor_common:id>${process_id}</reactor_common:id>
    </reactor_common:object_reference>
  </process>
  <status>
    <reactor_common:object_reference
              xmlns:reactor_common="reactor_common.dtd">
      <reactor_common:id>${status_id}</reactor_common:id>
    </reactor_common:object_reference>
  </status>
</parameters>
</reactor_request>
```

Sample XML response:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
  <result_code>
    <code_number>200</code_number>
    <user_message>OK</user_message>
    <debug_info />
  </result_code>
  <return_values />
</response>
```

SetObjects

The SetObjects request sets the value of existing objects and creates new objects. Use the Create request where it would be an error to modify the value of existing objects.

This request takes the following parameters:

Objects To Update - a set of Reactor objects

Use Label Paths - whether request should specify objects with

label paths instead of object IDs

If not specified, the request will not use label paths.

An application may need the request to use label paths instead of object IDs because the Reactor 5 Server will assign new IDs to created objects. This only an issue if the application stores and continues to use the IDs of new objects after sending the request.

The response to this request contains a result code, but does not contain any return values.

Sample XML request:

```
<?xml version="1.0" encoding="UTF-8"?>
<reactor_request>
  <request_type>process_command_set</request_type>
  <authentication_token>authToken</authentication_token>
  <parameters>
    <use_id_object_references />
    <objects>
      <operand:operand xmlns:operand="operand.dtd"visible_in_entire_subtree="false">
        <reactor_common:idxmlns:reactor_common="reactor_common.dtd">402881e6:78a212:fb9d230b1f:-
         8000</reactor_common:id>
        <reactor_common:labelxmlns:reactor_common="reactor_common.dtd">Process
         Title</reactor_common:label>
        <reactor_common:descriptionxmlns:reactor_common="reactor_common.dtd"><![CDATA[Description]]
         ></reactor_common:description>
        <reactor_common:acl xmlns:reactor_common="reactor_common.dtd" />
        <operand:operand_value><![CDATA[New Value]]></operand:operand_value>
        <reactor_common:associated_processxmlns:reactor_common="reactor_common.dtd">
          <reactor_common:object_reference>
            <reactor_common:id>OperandObjectId</reactor_common:id>
          </reactor_common:object_reference>
        </reactor_common:associated_process>
        <reactor_common:metadata xmlns:reactor_common="reactor_common.dtd" />
      </operand:operand>
    </objects>
  </parameters>
</reactor_request>
```

Sample XML response:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
  <result_code>
    <code_number>200</code_number>
    <user_message>OK</user_message>
    <debug_info />
  </result_code>
  <return_values />
</response>
```

Start

The Start request changes the state of a process instance to be "started".

This request takes the following parameters:

Process ID - the process acquiring the new status

Process Label Path - the process acquiring the new status

Either the Process ID or Process Label Path may be specified, but not both. Setting one causes the other to become unset.

The response to this request contains a result code, but does not contain any return values.

Sample XML request:

```
<?xml version="1.0" encoding="UTF-8"?>
<reactor_request>
  <request_type>process_command_start</request_type>
  <authentication_token>${authentication_token}</authentication_token>
  <parameters>
    <reactor_common:object_reference
              xmlns:reactor_common="reactor_common.dtd">
      <reactor_common:id>${process_id}</reactor_common:id>
    </reactor_common:object_reference>
  </parameters>
</reactor_request>
```

Sample XML response:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
  <result_code>
    <code_number>200</code_number>
    <user_message>OK</user_message>
    <debug_info />
  </result_code>
  <return_values />
</response>
```

Stop

The Stop request changes the state of a process instance to be "finished".

This request takes the following parameters:

Process ID - the process acquiring the new status

Process Label Path - the process acquiring the new status

Either the Process ID or Process Label Path may be specified, but not both. Setting one causes the other to become unset.

The response to this request contains a result code, but does not contain any return values.

Sample XML request:

```
<?xml version="1.0" encoding="UTF-8"?>
<reactor_request>
  <request_type>process_command_stop</request_type>
  <authentication_token>${authentication_token}</authentication_token>
  <parameters>
    <reactor_common:object_reference
              xmlns:reactor_common="reactor_common.dtd">
      <reactor_common:id>${process_id}</reactor_common:id>
    </reactor_common:object_reference>
  </parameters>
</reactor_request>
```

Sample XML response:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
  <result_code>
    <code_number>200</code_number>
    <user_message>OK</user_message>
    <debug_info />
  </result_code>
  <return_values />
</response>
```

Unlock

The Unlock request release a role from the ACL of an element. If the role is not already present in the ACL, the request is denied. The request can optionally take the name of an "eligible" role and the name of a "standby" role. If present, elements with the "standby" role are modified to have the "role" role. The request can also take a status. This will remove a status from the process being unlocked, or from the associated process if the object is not a process.

This request takes the following parameters:

Object ID - specifies the object to unlock

Object Label Path - specifies the object to unlock

ACE - the ACE to remove

Eligible Role - the role for ACEs which will be able to acquire a lock

Standby Role - the role for ACEs which can not currently acquire a lock

Status ID - specifies the status to remove

Status Label Path - specifies the status to remove

The Object ID and Object Label Path cannot both be specified. Similarly, the Status ID and Status Label Path cannot both be specified.

Sample XML request:

```
<?xml version="1.0" encoding="UTF-8"?>
<reactor_request xmlns:reactor_common="reactor_common.dtd">
  <request_type>
    process_command_unlock
  </request_type>
  <authentication_token>
    ${authentication_token}
  </authentication_token>
  <parameters>
    <reactor_common:object_reference>
      <reactor_common:id>${id}</reactor_common:id>
    </reactor_common:object_reference>
    <reactor_common:ace>
      <reactor_common:profile>
        <reactor_common:user>${user}</reactor_common:user>
      </reactor_common:profile>
      <reactor_common:role>${lock_role}</reactor_common:role>
    </reactor_common:ace>
    <eligible_role>${eligible_role}</eligible_role>
    <standby_role>${standby_role}</standby_role>
  </parameters>
</reactor_request>
```

Sample XML response:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
  <result_code>
    <code_number>200</code_number>
    <user_message>OK</user_message>
    <debug_info />
  </result_code>
  <return_values />
</response>
```

SOAP Messages and Web Services

Some configurations of Reactor 5 Server require additional actions to deploy the SOAP interface. Administrators should refer to ReadmeSOAP.html in the Reactor 5 Server distribution for more details.

The request and response SOAP messages provide wrappers for the same XML used in the XML/HTTP interface. The format of the requests and responses is described in XML Schemas and a DTD that can be used to construct and validate the XML messages.

The XML Schemas are listed in Appendix C and can be found at these URLs:

http://www.oakgrovesystems.com/xml/R5.xsd

http://www.oakgrovesystems.com/xml/reactor_common.xsd http://www.oakgrovesystems.com/xml/process.xsd

http://www.oakgrovesystems.com/xml/operand.xsd

http://www.oakgrovesystems.com/xml/status.xsd

http://www.oakgrovesystems.com/xml/policy.xsd

The DTD is listed in Appendix B and can be found at this URL:

http://www.oakgrovesystems.com/xml/R5.dtd

Reactor 5 is also distributed with a WSDL file which can be used to deploy Reactor 5 Server as a web service. Future versions of Reactor 5 are expected to also make it possible to expose individual processes and activities as web services using WSFL. The SOAP interface is deployed using Apache SOAP, which makes it possible to use the SOAP interface on a wide variety of application servers. The WSDL file is listed in Appendix D and included in the webservices directory of the Reactor 5 Server distribution. The WSDL file can be accessed on the deployed server through the following URL, though the host name and port number of the server may vary for different configurations.

http://localhost:9090/ReactorServer/wsdl/ReactorServer.wsdl

The following shows a sample SOAP request and response, including the HTTP headers. The examples from Section 4.3 can be placed in the request and response elements at the appropriate parts of the SOAP messages. The XML request is passed as a string argument, and the XML response is also returned as a string argument.

This is the request SOAP message:

```
POST /soap/servlet/rpcrouter HTTP/1.0
Host: localhost
Content-Type: text/xml; charset=utf-8
Content-Length: 828
SOAPAction: "(empty soap action)"
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
       xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
       xmlns:xsd="http://www.w3.org/1999/XMLSchema">
<SOAP-ENV:Body>
<ns1:handleRequest xmlns:ns1="urn:reactor-service" SOAP-
       ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<request xsi:type="xsd:string">&lt;?xml version=&quot;1.0&quot;
       encoding=&quot;UTF-8&quot;?&gt;
&lt;reactor_request&gt;
    &lt;request_type&gt;authentication_login&lt;/request_type&gt;
    &lt;parameters&gt;
        &lt;login&gt;
            &lt;username&gt;admin&lt;/username&gt;
            &lt;password&gt;admin&lt;/password&gt;
        &lt;/login&gt;
    &lt;/parameters&gt;
&lt;/reactor_request&gt;
</request>
</ns1:handleRequest>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

This is the response SOAP message:


```
HTTP/1.0 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: 899
Set-Cookie: JSESSIONID=b41ogxyzm1;Path=/soap
Servlet-Engine: Tomcat Web Server/3.2.2 (JSP 1.1; Servlet 2.2; Java 1.3.0;
       java.vendor=Sun Microsystems Inc.)
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
       xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
       xmlns:xsd="http://www.w3.org/1999/XMLSchema">
<SOAP-ENV:Body>
<ns1:handleRequestResponse xmlns:ns1="urn:reactor-service" SOAP-
       ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<return xsi:type="xsd:string">&lt;?xml version=&quot;1.0&quot;
       encoding=&quot;UTF-8&quot;?&gt;
&lt;response&gt;&lt;result_code&gt;&lt;code_number&gt;200&lt;/code_number&gt;&lt
       ;user_message&gt;OK&lt;/user_message&gt;&lt;debug_info
       /&gt;&lt;/result_code&gt;&lt;return_values&gt;&lt;authentication_token&gt
       ;-2f4d386d:394894:e8ed3c0e36:-
       77d1&lt;/authentication_token&gt;&lt;token_expiration&gt;never&lt;/token_
       expiration&gt;&lt;/return_values&gt;&lt;/response&gt;
</return>
</ns1:handleRequestResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Appendix A: Label paths

The string representation of a label path looks like this:

```
operand://-2f4d386d:2c7568%3Ae8571b9182:-7cf4/Process3/Operand7
```

This is the BNF definition of the string label path syntax:

```
label_path      := type ':' path
path            := absolute_path | relative_path
absolute_path   := [ '//' ] '/' labels
relative_path   := '//' id '/' relative_labels
relative_labels := dots | labels | dots '/' labels
dots            := '.' | dotdots | '.' '/' dotdots
dotdots         := '..' | '..' '/' dotdots
```

Paths starting with an object that has no parent or associated process can have either of these formats:

```
<TYPE>:/<ABSOLUTE_PATH>
<TYPE>:///<ABSOLUTE_PATH>
```

Paths relative to an object ID can have either of these formats:

```
<TYPE>://<ID>/<RELATIVE_PATH>
<TYPE>://<ID>/
```

These are the valid <TYPE> values and their meanings:

```
def - object must be definition or associated with definition
inst - object must be instance or associated with instance
process - object must be either definition or instance
operand - object must be an operand
status - object must be a status
policy - object must be a policy
object - object can be any type
```

The <ID> is an object's unique ID. Note that this can not contain "/".

An <ABSOLUTE_PATH> is a list of labels, separated by "/". The first label specifies an object that has no parent or associated process. The last label is the label of the object being specified by the whole path.

A <RELATIVE_PATH> is like the absolute path, except that the labels are optional, and may be preceded by an optional list of "." and ".." elements. There can be at most one "." element at the beginning. There can be any number of ".." elements immediately preceding the labels. The "." makes the labels relative to the process or associated process of the object with the specified ID. If the ID specifies a process, then the "." refers to that process. Otherwise, the "." refers to the process associated with that object. The first ".." makes the labels relative to the parent of the process specified or associated with the ID. Each following ".." makes the labels relative to the parent one more level up.

The XML representation of a label path looks like this:

```
        <reactor_common:label_path type="operand">
                <reactor_common:root>
                        -2f4d386d%3A2c7568%3Ae8571b9182%3A-7cf4
                </reactor_common:root>
                <reactor_common:label>
                        Process3
                </reactor_common:label>
                <reactor_common:label>
                        Operand7
                </reactor_common:label>
        </reactor_common:label_path>
```

This is the DTD fragment for the label path format, including "xmlns" for namespaces.

```
<!ELEMENT reactor_common:label (#PCDATA)>
<!ATTLIST reactor_common:label
          xmlns CDATA #FIXED "reactor_common.dtd">
          <!--
          The root is used when a label path starts from a particular
          object. It contains the ID of a Reactor object.
          -->
<!ELEMENT reactor_common:root (#PCDATA)>
<!ATTLIST reactor_common:root
          xmlns CDATA #FIXED "reactor_common.dtd">
          <!--
          The <label> elements in a <label_path> are ordered.  The order of
          <label> elements represents the order of traversal down the object
          tree from parent to child. If the object is at the root level,
             meaning that it has no parents, then the label path contains only
          the label of the object.  Otherwise, the label path starts with a
          top-level process and follows the chain of child processes down to
          the process containing the object.  The last <label> element in a
          <label_path> is the <label> of the object to which the <label_path>
          resolves. If specified, the type must be process, operand, status,
          policy, or object.
          -->
<!ELEMENT reactor_common:label_path
          ((reactor_common:root,reactor_common:label*)
          | (reactor_common:label+))>
<!ATTLIST reactor_common:label_path
          type (process | operand | status | policy | object) #IMPLIED
          xmlns CDATA #FIXED "reactor_common.dtd"
          xmlns:reactor_common CDATA #FIXED "reactor_common.dtd">
```

Appendix B: XML DTD

```
<!-- R5.dtd (Reactor 5.0.2, 2001 August 17) -->
<!-- Oak Grove Systems, Inc. -->
<!-- http://www.oakgrovesystems.com/ -->
<!--
          A note about XML Namespaces:
          Namespaces are not officially part of the XML 1.0 specification,
          so a DTD can not completely express the requirements for properly
          constructing XML documents with namespaces.
          In order to allow the XML to be properly validated, each element
          is defined to have an optional xmlns attribute. This is not an
          accurate expression of how the xmlns attribute should be used.
          Every element with a namespace in its name must be in a context
          where that namespace is defined. There must be an xmlns attribute
          in either that element or some element containing it. That xmlns
          attribute must specify the proper location of the DTD. It's not
          necessary for the location to actually contain the DTD, or even
          be reachable, but it is necessary for the location to be the
          same as used by the parser that will be reading the XML.
          Each namespace has a separate section in this DTD. The comments
          at the beginning of each section contain the xmlns attribute
          required for that namespace.
          See http://www.w3.org/TR/REC-xml-names/ for the official document
          describing XML Namespaces.
-->
<!-- ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
DEFAULT NAMESPACE (no xmlns required)
These elements are used for requests and responses. The parameters and
return values will sometimes contain elements from the other namespaces.
-->
<!ELEMENT reactor_request
          (request_type,
          authentication_token?,
          parameters?)>
<!ELEMENT request_type (#PCDATA)>
<!ELEMENT authentication_token (#PCDATA)> <!-- empty for login requests -->
<!ELEMENT parameters ANY>

<!ELEMENT response
          (result_code,
          return_values?)>
<!ELEMENT result_code
          (code_number,
          user_message,
          debug_info)>
<!ELEMENT code_number (#PCDATA)>
<!ELEMENT user_message (#PCDATA)>
<!ELEMENT debug_info (#PCDATA)>
<!ELEMENT return_values ANY>

<!-- For authentication_login requests and responses: -->
<!ELEMENT login
          (username,
          password)>
<!ELEMENT username (#PCDATA)>
<!ELEMENT password (#PCDATA)>
```

```
<!ELEMENT token_expiration (#PCDATA)>
<!-- For process_command_* and process_object_* requests and responses: -->
          <!-- This is used in a 'process_object_get' request
          to specify the type of object being retrieved. -->
<!ELEMENT type (#PCDATA)>
          <!-- The 'query' tag goes inside the 'parameters' tag of a
          'reactor_request' document of type 'process_object_query' -->
<!ELEMENT query
          (all_objects
          | process_tree
          | processes)>
          <!-- This returns all stored objects (useful for debugging) -->
<!ELEMENT all_objects EMPTY>
          <!-- This returns a process and specified associated objects. -->
<!ELEMENT process_tree
          (depth?,
          reactor_common:object_reference,
          getSuperprocess?,
          getOperands?,
          getStatuses?,
          getPolicies?)>
          <!-- The integer depth of the object tree to return.
          0 means just the root, a negative number means the entire tree.
          Default is 0. -->
<!ELEMENT depth (#PCDATA)>
          <!-- If the following elements are included and have the string
          "true" as the content (without quotes), then the process tree will
          contain the superprocess of the root process, operands, statuses,
          and policies (respectively) -->
<!ELEMENT getSuperprocess (#PCDATA)> <!-- (true|false) -->
<!ELEMENT getOperands (#PCDATA)> <!-- (true|false) -->
<!ELEMENT getStatuses (#PCDATA)> <!-- (true|false) -->
<!ELEMENT getPolicies (#PCDATA)> <!-- (true|false) -->
          <!-- This returns a collection of processes matching the
          specified criteria. -->
<!ELEMENT processes
          ((definitions | (instances|started_instances) )?,
          ace_pattern?,
          references_only?)>
          <!-- If this element is included, the query only returns processes
          that are definitions, not instantiated processes. -->
<!ELEMENT definitions EMPTY>
          <!-- If this element is included, the query only returns
          instantiated processes, not definitions. The instantiated processes
          could be in any state (unstarted, started, finished) -->
<!ELEMENT instances EMPTY>
          <!-- If this element is included, the query only returns
          instantiated that are in the "started" state. -->
<!ELEMENT started_instances EMPTY>
          <!-- If this element is included, references will be returned
          instead of whole processes. -->
<!ELEMENT references_only EMPTY>
          <!-- If included, the query only returns processes with one or
          more ACEs in their ACLs that match the specified pattern. -->
<!ELEMENT ace_pattern
          (principal_type?,
          principal?,
          role?)>
<!ELEMENT principal_type (#PCDATA)> <!-- (user|group|title) -->
<!ELEMENT principal (#PCDATA)> <!-- name of the user, group or title -->
```

```
<!ELEMENT role (#PCDATA)> <!-- eg: Initiative participant -->
          <!-- This is used to specify the process in a
          'process_command_add_status' request. -->
<!ELEMENT process (reactor_common:object_reference)>
          <!-- This is used to specify the status in a
          'process_command_add_status' request. -->
<!ELEMENT status (reactor_common:object_reference)>
          <!-- This is used when including a set of objects in both
          requests and responses. -->
<!ELEMENT objects ANY>
          <!-- To get object_reference elements that use 'id' instead of
          'label_path' in a response, include the 'use_id_object_references'
          element in the 'parameters' of a request. -->
<!ELEMENT use_id_object_references EMPTY>

<!-- ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
NAMESPACE reactor_common (xmlns:reactor_common="reactor_common.dtd")
These elements are general, and are used in the other namespaces.
-->
<!ELEMENT reactor_common:label (#PCDATA)>
<!ATTLIST reactor_common:label
          xmlns CDATA #FIXED "reactor_common.dtd">
          <!-- Wrap the contents in CDATA before adding to this element! -->
          <!-- e.g. <![CDATA[...contents...]]> -->
<!ELEMENT reactor_common:description (#PCDATA)>
<!ATTLIST reactor_common:description
          xmlns CDATA #FIXED "reactor_common.dtd"
          xmlns:reactor_common CDATA #FIXED "reactor_common.dtd">
<!ELEMENT reactor_common:acl
          (reactor_common:ace)*>
<!ATTLIST reactor_common:acl
          xmlns CDATA #FIXED "reactor_common.dtd"
          xmlns:reactor_common CDATA #FIXED "reactor_common.dtd">
<!ELEMENT reactor_common:ace
          (reactor_common:profile+,
          reactor_common:role+)>
<!ATTLIST reactor_common:ace
          xmlns CDATA #FIXED "reactor_common.dtd"
          xmlns:reactor_common CDATA #FIXED "reactor_common.dtd">
<!ELEMENT reactor_common:profile
          (reactor_common:user
          | reactor_common:title
          | reactor_common:group)>
<!ATTLIST reactor_common:profile
          xmlns CDATA #FIXED "reactor_common.dtd"
          xmlns:reactor_common CDATA #FIXED "reactor_common.dtd">
<!ELEMENT reactor_common:user (#PCDATA)>
<!ATTLIST reactor_common:user
          xmlns CDATA #FIXED "reactor_common.dtd"
          xmlns:reactor_common CDATA #FIXED "reactor_common.dtd">
<!ELEMENT reactor_common:title (#PCDATA)>
<!ATTLIST reactor_common:title
          xmlns CDATA #FIXED "reactor_common.dtd"
          xmlns:reactor_common CDATA #FIXED "reactor_common.dtd">
<!ELEMENT reactor_common:group (#PCDATA)>
<!ATTLIST reactor_common:group
          xmlns CDATA #FIXED "reactor_common.dtd"
          xmlns:reactor_common CDATA #FIXED "reactor_common.dtd">
<!ELEMENT reactor_common:role (#PCDATA)>
<!ATTLIST reactor_common:role
```

```
          xmlns CDATA #FIXED "reactor_common.dtd"
          xmlns:reactor_common CDATA #FIXED "reactor_common.dtd">
<!ELEMENT reactor_common:associated_process
          (reactor_common:object_reference)?>
<!ATTLIST reactor_common:associated_process
          xmlns CDATA #FIXED "reactor_common.dtd"
          xmlns:reactor_common CDATA #FIXED "reactor_common.dtd">
<!ELEMENT reactor_common:object_reference
          (reactor_common:id
          | reactor_common:label_path)>
<!ATTLIST reactor_common:object_reference
          xmlns CDATA #FIXED "reactor_common.dtd"
          xmlns:reactor_common CDATA #FIXED "reactor_common.dtd">
<!ELEMENT reactor_common:id (#PCDATA)>
<!ATTLIST reactor_common:id
          xmlns CDATA #FIXED "reactor_common.dtd"
          xmlns:reactor_common CDATA #FIXED "reactor_common.dtd">
          <!--
          The root is used when a label path starts from a particular
          object. It contains the ID of a Reactor object.
          -->
<!ELEMENT reactor_common:root (#PCDATA)>
<!ATTLIST reactor_common:root
          xmlns CDATA #FIXED "reactor_common.dtd">
          <!--
          The <label> elements in a <label_path> are ordered.  The order of
          <label> elements represents the order of traversal down the object
          tree from parent to child. If the object is at the root level,
          meaning that it has no parents, then the label path contains only
          the label of the object.  Otherwise, the label path starts with a
          top-level process and follows the chain of child processes down to
          the process containing the object.  The last <label> element in a
          <label_path> is the <label> of the object to which the <label_path>
          resolves. If specified, the type must be process, operand, status,
          policy, or object.
          -->
<!ELEMENT reactor_common:label_path
          ((reactor_common:root,reactor_common:label*)
          | (reactor_common:label+))>
<!ATTLIST reactor_common:label_path
          type (process | operand | status | policy | object) #IMPLIED
          xmlns CDATA #FIXED "reactor_common.dtd"
          xmlns:reactor_common CDATA #FIXED "reactor_common.dtd">

<!-- ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
NAMESPACE process (xmlns:process="process.dtd")
These elements define processes, with their attributes and associations.
Attributes are contained in the process element. The process element also
contains references to associated objects.
-->
<!ELEMENT process:process
          (reactor_common:id?,
          reactor_common:label,
          reactor_common:description,
          reactor_common:acl,
          process:start_date,
          process:end_date,
          process:timers,
          process:timer_ids,
          process:state,
```

```
          process:current_statuses,
          process:precondition*,
          process:change_condition*,
          process:operands,
          process:statuses,
          process:policies,
          process:superprocess,
          process:subprocesses)>
<!ATTLIST process:process
          definition (true|false) #REQUIRED
          xmlns CDATA #FIXED "process.dtd"
          xmlns:process CDATA #FIXED "process.dtd"
          xmlns:reactor_common CDATA #FIXED "reactor_common.dtd">
<!ELEMENT process:start_date (#PCDATA)>
<!ATTLIST process:start_date
          xmlns CDATA #FIXED "process.dtd"
          xmlns:process CDATA #FIXED "process.dtd">
<!ELEMENT process:timers
          (process:timer_spec)*>
<!ATTLIST process:timers
          xmlns CDATA #FIXED "process.dtd"
          xmlns:process CDATA #FIXED "process.dtd">
<!ELEMENT process:timer_spec
          ((process:timer_description),
          (process:event),
          (process:calendar)?,
          (process:schedule_expression | process:schedule_date)?,
          (process:repeat_expression)?,
          (process:end_expression | process:end_date | process:repeat_count))>
<!ATTLIST process:timer_spec
          xmlns CDATA #FIXED "process.dtd"
          xmlns:process CDATA #FIXED "process.dtd">
<!ELEMENT process:timer_description (#PCDATA)>
<!ATTLIST process:timer_description
          xmlns CDATA #FIXED "process.dtd"
          xmlns:process CDATA #FIXED "process.dtd">
<!ELEMENT process:event
          (process:event_name,
          (process:event_attribute*))>
<!ATTLIST process:event
          xmlns CDATA #FIXED "process.dtd"
          xmlns:process CDATA #FIXED "process.dtd">
<!ELEMENT process:event_name (#PCDATA)>
<!ATTLIST process:event_name
          xmlns CDATA #FIXED "process.dtd"
          xmlns:process CDATA #FIXED "process.dtd">
<!ELEMENT process:event_attribute
          (process:att_key,
          process:att_value)>
<!ATTLIST process:event_name
          xmlns CDATA #FIXED "process.dtd"
          xmlns:process CDATA #FIXED "process.dtd">
<!ELEMENT process:att_key (#PCDATA)>
<!ATTLIST process:att_key
          xmlns CDATA #FIXED "process.dtd"
          xmlns:process CDATA #FIXED "process.dtd">
<!ELEMENT process:att_value (#PCDATA)>
<!ATTLIST process:att_value
          xmlns CDATA #FIXED "process.dtd"
          xmlns:process CDATA #FIXED "process.dtd">
```

```
<!ELEMENT process:calendar (#PCDATA)>
<!ATTLIST process:calendar
          xmlns CDATA #FIXED "process.dtd"
          xmlns:process CDATA #FIXED "process.dtd">
<!ELEMENT process:schedule_expression (#PCDATA)>
<!ATTLIST process:schedule_expression
          xmlns CDATA #FIXED "process.dtd"
          xmlns:process CDATA #FIXED "process.dtd">
<!ELEMENT process:schedule_date (#PCDATA)>
<!ATTLIST process:schedule_date
          xmlns CDATA #FIXED "process.dtd"
          xmlns:process CDATA #FIXED "process.dtd">
<!ELEMENT process:repeat_expression (#PCDATA)>
<!ATTLIST process:repeat_expression
          xmlns CDATA #FIXED "process.dtd"
          xmlns:process CDATA #FIXED "process.dtd">
<!ELEMENT process:end_expression (#PCDATA)>
<!ATTLIST process:end_expression
          xmlns CDATA #FIXED "process.dtd"
          xmlns:process CDATA #FIXED "process.dtd">
<!ELEMENT process:end_date (#PCDATA)>
<!ATTLIST process:end_date
          xmlns CDATA #FIXED "process.dtd"
          xmlns:process CDATA #FIXED "process.dtd">
<!ELEMENT process:repeat_count (#PCDATA)>
<!ATTLIST process:repeat_count
          xmlns CDATA #FIXED "process.dtd"
          xmlns:process CDATA #FIXED "process.dtd">
<!ELEMENT process:timer_ids
          (process:timer_id)*>
<!ATTLIST process:timer_ids
          xmlns CDATA #FIXED "process.dtd"
          xmlns:process CDATA #FIXED "process.dtd">
<!ELEMENT process:timer_id (#PCDATA)>
<!ATTLIST process:timer_id
          xmlns CDATA #FIXED "process.dtd"
          xmlns:process CDATA #FIXED "process.dtd">
<!ELEMENT process:state EMPTY>
<!ATTLIST process:state
          state_name (unstarted|started|finished) #REQUIRED>
<!ELEMENT process:current_statuses
          (reactor_common:object_reference)*>
<!ATTLIST process:current_statuses
          xmlns CDATA #FIXED "process.dtd"
          xmlns:process CDATA #FIXED "process.dtd">
<!ELEMENT process:precondition
          (process:condition)?>
<!ATTLIST process:precondition
          xmlns CDATA #FIXED "process.dtd"
          xmlns:process CDATA #FIXED "process.dtd">
<!ELEMENT process:change_condition
          (process:condition,
          process:change+)>
<!ATTLIST process:change_condition
          xmlns CDATA #FIXED "process.dtd"
          xmlns:process CDATA #FIXED "process.dtd">
<!ELEMENT process:condition
          (process:conjunction
          | process:disjunction
          | process:inversion
```

```
          | process:process_state_equals
          | process:process_has_status)>
<!ATTLIST process:condition
          xmlns CDATA #FIXED "process.dtd"
          xmlns:process CDATA #FIXED "process.dtd">
<!ELEMENT process:conjunction
          (process:condition)+>
<!ATTLIST process:conjunction
          xmlns CDATA #FIXED "process.dtd"
          xmlns:process CDATA #FIXED "process.dtd">
<!ELEMENT process:disjunction
          (process:condition)+>
<!ATTLIST process:disjunction
          xmlns CDATA #FIXED "process.dtd"
          xmlns:process CDATA #FIXED "process.dtd">
<!ELEMENT process:inversion
          (process:condition)>
<!ATTLIST process:inversion
          xmlns CDATA #FIXED "process.dtd"
          xmlns:process CDATA #FIXED "process.dtd">
<!ELEMENT process:process_state_equals
          (reactor_common:object_reference,
          process:state)>
<!ATTLIST process:process_state_equals
          xmlns CDATA #FIXED "process.dtd"
          xmlns:process CDATA #FIXED "process.dtd">
<!ELEMENT process:process_has_status
          (reactor_common:object_reference,
          reactor_common:object_reference)>
<!ATTLIST process:process_has_status
          xmlns CDATA #FIXED "process.dtd"
          xmlns:process CDATA #FIXED "process.dtd">
<!ELEMENT process:change
          (process:state_change
          | process:status_addition
          | process:status_removal)>
<!ATTLIST process:change
          xmlns CDATA #FIXED "process.dtd"
          xmlns:process CDATA #FIXED "process.dtd">
<!ELEMENT process:state_change
          (process:state)>
<!ATTLIST process:state_change
          xmlns CDATA #FIXED "process.dtd"
          xmlns:process CDATA #FIXED "process.dtd">
<!ELEMENT process:status_addition
          (reactor_common:object_reference)>
<!ATTLIST process:status_addition
          xmlns CDATA #FIXED "process.dtd"
          xmlns:process CDATA #FIXED "process.dtd">
<!ELEMENT process:status_removal
          (reactor_common:object_reference)>
<!ATTLIST process:status_removal
          xmlns CDATA #FIXED "process.dtd"
          xmlns:process CDATA #FIXED "process.dtd">
          <!-- According to MOF/XMI, the five elements below would be
          "associations", whereas the elements above would be "attributes" -->
<!ELEMENT process:operands
          (reactor_common:object_reference)*>
<!ATTLIST process:operands
          xmlns CDATA #FIXED "process.dtd"
```

```
          xmlns:process CDATA #FIXED "process.dtd">
<!ELEMENT process:statuses
          (reactor_common:object_reference)*>
<!ATTLIST process:statuses
          xmlns CDATA #FIXED "process.dtd"
          xmlns:process CDATA #FIXED "process.dtd">
<!ELEMENT process:policies
          (reactor_common:object_reference)*>
<!ATTLIST process:policies
          xmlns CDATA #FIXED "process.dtd"
          xmlns:process CDATA #FIXED "process.dtd">
          <!-- superprocess is empty if process is at the top level -->
<!ELEMENT process:superprocess
          (reactor_common:object_reference)?>
<!ATTLIST process:superprocess
          xmlns CDATA #FIXED "process.dtd"
          xmlns:process CDATA #FIXED "process.dtd">
<!ELEMENT process:subprocesses
          (reactor_common:object_reference)*>
<!ATTLIST process:subprocesses
          xmlns CDATA #FIXED "process.dtd"
          xmlns:process CDATA #FIXED "process.dtd">

<!-- ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
NAMESPACE operand (xmlns:operand="operand.dtd")
Note that the associated process for an operand is associated by using
an object_reference element.
-->
<!ELEMENT operand:operand
          (reactor_common:id?,
          reactor_common:label,
          reactor_common:description,
          reactor_common:acl,
          operand:operand_value,
          reactor_common:associated_process)>
<!ATTLIST operand:operand
          visible_in_entire_subtree (true|false) #REQUIRED
          xmlns CDATA #FIXED "operand.dtd"
          xmlns:operand CDATA #FIXED "operand.dtd"
          xmlns:reactor_common CDATA #FIXED "reactor_common.dtd">
          <!-- Wrap the contents in CDATA before adding to this element! -->
          <!-- e.g. <![CDATA[...contents...]]> -->
<!ELEMENT operand:operand_value (#PCDATA)>
<!ATTLIST operand:operand_value
          xmlns CDATA #FIXED "operand.dtd"
          xmlns:operand CDATA #FIXED "operand.dtd">

<!-- ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
NAMESPACE status (xmlns:status="status.dtd")
Note that the associated process for a status is associated by using
an object_reference element.
-->
<!ELEMENT status:status
          (reactor_common:id?,
          reactor_common:label,
          reactor_common:description,
          reactor_common:acl,
          reactor_common:associated_process)>
<!ATTLIST status:status
          applicable_to_entire_subtree (true|false) #REQUIRED
```

```
          xmlns CDATA #FIXED "status.dtd"
          xmlns:status CDATA #FIXED "status.dtd"
          xmlns:reactor_common CDATA #FIXED "reactor_common.dtd">

<!-- :::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
NAMESPACE policy (xmlns:policy="policy.dtd")
Note that the associated process for a policy is associated by using
an object_reference element.
-->
<!ELEMENT policy:policy
          (reactor_common:id?,
          reactor_common:label,
          reactor_common:description,
          reactor_common:acl,
          policy:event_name,
          policy:event_attributes,
          policy:policy_type,
          policy:language?,
          policy:policy_definition,
          policy:security_policy,
          reactor_common:associated_process)>
<!ATTLIST policy:policy
          xmlns CDATA #FIXED "policy.dtd"
          xmlns:policy CDATA #FIXED "policy.dtd"
          xmlns:reactor_common CDATA #FIXED "reactor_common.dtd">
<!ELEMENT policy:event_name (#PCDATA)>
          <!-- e.g. "ProcessStateChange", "ProcessLabelChange",
               "StatusCreation", "OperandValueChanged" -->
<!ATTLIST policy:event_name
          xmlns CDATA #FIXED "policy.dtd"
          xmlns:policy CDATA #FIXED "policy.dtd">
<!ELEMENT policy:event_attributes ANY>
          <!-- information about the event (what happened?, why did it happen?,
               who triggered it?, when did it happen?, etc.) -->
<!ATTLIST policy:event_attributes
          xmlns CDATA #FIXED "policy.dtd"
          xmlns:policy CDATA #FIXED "policy.dtd">
          <!-- for future use -->
<!ELEMENT policy:event_source
          (reactor_common:object_reference
          | policy:service_name)>
          <!-- e.g. ID of the process object involved -->
<!ATTLIST policy:event_source
          xmlns CDATA #FIXED "policy.dtd"
          xmlns:policy CDATA #FIXED "policy.dtd">
          <!-- for future use -->
<!ELEMENT policy:service_name (#PCDATA)>
<!ATTLIST policy:service_name
          xmlns CDATA #FIXED "policy.dtd"
          xmlns:policy CDATA #FIXED "policy.dtd">
<!ELEMENT policy:policy_type (#PCDATA)>
          <!-- "JavaClass", "BSF", etc. -->
<!ATTLIST policy:policy_type
          xmlns CDATA #FIXED "policy.dtd"
          xmlns:policy CDATA #FIXED "policy.dtd">
<!ELEMENT policy:language (#PCDATA)>
          <!-- java -->
<!ATTLIST policy:language
          xmlns CDATA #FIXED "policy.dtd"
          xmlns:policy CDATA #FIXED "policy.dtd">
```

```
<!ELEMENT policy:policy_definition
          (policy:classname
          | policy:URL
          | policy:source_code)>
          <!-- code or reference to code -->
<!ATTLIST policy:policy_definition
          xmlns CDATA #FIXED "policy.dtd"
          xmlns:policy CDATA #FIXED "policy.dtd">
<!ELEMENT policy:security_policy (#PCDATA)>
          <!-- the Java Security policy enforced during the execution
               of this policy -->
<!ATTLIST policy:security_policy
          xmlns CDATA #FIXED "policy.dtd"
          xmlns:policy CDATA #FIXED "policy.dtd">
<!ELEMENT policy:classname (#PCDATA)>
<!ATTLIST policy:classname
          xmlns CDATA #FIXED "policy.dtd"
          xmlns:policy CDATA #FIXED "policy.dtd">
<!ELEMENT policy:URL (#PCDATA)>
<!ATTLIST policy:URL
          xmlns CDATA #FIXED "policy.dtd"
          xmlns:policy CDATA #FIXED "policy.dtd">
          <!-- Wrap the contents in CDATA before adding to this element! -->
          <!-- e.g. <![CDATA[...contents...]]> -->
<!ELEMENT policy:source_code (#PCDATA)>
<!ATTLIST policy:source_code
          xmlns CDATA #FIXED "policy.dtd"
          xmlns:policy CDATA #FIXED "policy.dtd">
```

Appendix C: XML Schemas

R5.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema
        xmlns:xsd='http://www.w3.org/2001/XMLSchema'
        xmlns:reactor_common='reactor_common.dtd'
        xmlns:process='process.dtd'
        xmlns:operand='operand.dtd'
        xmlns:status='status.dtd'
        xmlns:policy='policy.dtd'>
 <xsd:import namespace='reactor_common.dtd'
       schemaLocation='reactor_common.xsd'/>
 <xsd:import namespace='process.dtd' schemaLocation='process.xsd'/>
 <xsd:import namespace='operand.dtd' schemaLocation='operand.xsd'/>
 <xsd:import namespace='status.dtd' schemaLocation='status.xsd'/>
 <xsd:import namespace='policy.dtd' schemaLocation='policy.xsd'/>
 <xsd:complexType name='reactor_request' mixed='true'>
  <xsd:sequence>
   <xsd:element ref='request_type'/>
   <xsd:element ref='authentication_token' minOccurs='0'/>
   <xsd:element ref='parameters' minOccurs='0'/>
  </xsd:sequence>
 </xsd:complexType>
 <xsd:element name='reactor_request' type='reactor_request'/>
 <xsd:element name='request_type' type='xsd:string'/>
 <xsd:element name='authentication_token' type='xsd:string'/>
 <xsd:element name='parameters'>
  <xsd:complexType>
   <xsd:sequence>
    <xsd:any minOccurs='0' maxOccurs='unbounded'/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
 <xsd:complexType name='response' mixed='true'>
  <xsd:sequence>
   <xsd:element ref='result_code'/>
   <xsd:element ref='return_values' minOccurs='0'/>
  </xsd:sequence>
 </xsd:complexType>
 <xsd:element name='response' type='response'/>
 <xsd:element name='result_code'>
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element ref='code_number'/>
    <xsd:element ref='user_message'/>
    <xsd:element ref='debug_info'/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
 <xsd:element name='code_number' type='xsd:int'/>
 <xsd:element name='user_message' type='xsd:string'/>
 <xsd:element name='debug_info' type='xsd:string'/>
 <xsd:element name='return_values'>
  <xsd:complexType>
   <xsd:sequence>
```

```
    <xsd:any minOccurs='0' maxOccurs='unbounded'/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
 <xsd:element name='login'>
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element ref='username'/>
    <xsd:element ref='password'/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
 <xsd:element name='username' type='xsd:string'/>
 <xsd:element name='password' type='xsd:string'/>
 <xsd:element name='token_expiration' type='xsd:string'/>
 <!-- For process_command_* and process_object_* requests and responses: -->
 <!-- This is used in a 'process_object_get' request
 to specify the type of object being retrieved. -->
 <xsd:element name='type' type='xsd:string'/>
 <xsd:element name='query'>
  <xsd:complexType>
   <xsd:choice>
    <xsd:element ref='all_objects'/>
    <xsd:element ref='process_tree'/>
    <xsd:element ref='processes'/>
   </xsd:choice>
  </xsd:complexType>
 </xsd:element>
 <xsd:element name='all_objects'/>
 <xsd:element name='process_tree'>
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element ref='depth' minOccurs='0'/>
    <xsd:element ref='reactor_common:object_reference'/>
    <xsd:element ref='getSuperprocess' minOccurs='0'/>
    <xsd:element ref='getOperands' minOccurs='0'/>
    <xsd:element ref='getStatuses' minOccurs='0'/>
    <xsd:element ref='getPolicies' minOccurs='0'/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
 <xsd:element name='depth' type='xsd:int'/>
 <xsd:element name='getSuperprocess'/>
 <xsd:element name='getOperands'/>
 <xsd:element name='getStatuses'/>
 <xsd:element name='getPolicies'/>
 <xsd:element name='processes'>
  <xsd:complexType>
   <xsd:sequence>
    <xsd:choice minOccurs='0'>
     <xsd:element ref='definitions'/>
     <xsd:choice>
      <xsd:element ref='instances'/>
      <xsd:element ref='started_instances'/>
     </xsd:choice>
    </xsd:choice>
    <xsd:element ref='ace_pattern' minOccurs='0'/>
    <xsd:element ref='references_only' minOccurs='0'/>
   </xsd:sequence>
  </xsd:complexType>
```

```
 </xsd:element>
 <xsd:element name='definitions'/>
 <xsd:element name='instances'/>
 <xsd:element name='started_instances'/>
 <xsd:element name='references_only'/>
 <xsd:element name='ace_pattern'>
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element ref='principal_type' minOccurs='0'/>
    <xsd:element ref='principal' minOccurs='0'/>
    <xsd:element ref='role' minOccurs='0'/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
 <!-- user|group|title -->
 <xsd:element name='principal_type' type='xsd:string'/>
 <xsd:element name='principal' type='xsd:string'/>
 <xsd:element name='role' type='xsd:string'/>
 <xsd:element name='process'>
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element ref='reactor_common:object_reference'/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
 <xsd:element name='status'>
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element ref='reactor_common:object_reference'/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
 <xsd:element name='objects'>
  <xsd:complexType>
   <xsd:sequence>
    <!-- set of process|operand|status|policy objects -->
    <xsd:any minOccurs='0' maxOccurs='unbounded'/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
 <xsd:element name='use_id_object_references'/>
</xsd:schema>
```

reactor_common.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<schema
        xmlns='http://www.w3.org/2001/XMLSchema'
        xmlns:reactor_common='reactor_common.dtd'
        targetNamespace='reactor_common.dtd'>
 <element name='label' type='string'/>
 <element name='description' type='string'/>
 <element name='acl'>
  <complexType>
   <sequence>
    <element ref='reactor_common:ace' minOccurs='0' maxOccurs='unbounded'/>
   </sequence>
  </complexType>
 </element>
```

```
<element name='ace'>
 <complexType>
  <sequence>
   <element ref='reactor_common:profile' maxOccurs='unbounded'/>
   <element ref='reactor_common:role' maxOccurs='unbounded'/>
  </sequence>
 </complexType>
</element>
<element name='profile'>
 <complexType>
  <sequence>
   <choice>
    <element ref='reactor_common:user'/>
    <element ref='reactor_common:title'/>
    <element ref='reactor_common:group'/>
   </choice>
  </sequence>
 </complexType>
</element>
<element name='user' type='string'/>
<element name='title' type='string'/>
<element name='group' type='string'/>
<element name='role' type='string'/>
<element name='associated_process'>
 <complexType>
  <sequence>
   <element ref='reactor_common:object_reference' minOccurs='0'/>
  </sequence>
 </complexType>
</element>
<element name='object_reference'>
 <complexType>
  <sequence>
   <choice>
     <element ref='reactor_common:id'/>
     <element ref='reactor_common:label_path'/>
   </choice>
  </sequence>
 </complexType>
</element>
<element name='id' type='string'/>
<element name='root' type='string'/>
<element name='label_path'>
 <complexType>
  <sequence>
   <choice>
    <sequence>
     <element ref='reactor_common:root'/>
     <element ref='reactor_common:label' minOccurs='0' maxOccurs='unbounded'/>
    </sequence>
    <sequence>
     <element ref='reactor_common:label' maxOccurs='unbounded'/>
    </sequence>
   </choice>
  </sequence>
  <attribute name='type' use='optional'>
   <simpleType>
    <restriction base='string'>
     <enumeration value='process'/>
     <enumeration value='operand'/>
```

```
      <enumeration value='status'/>
      <enumeration value='policy'/>
      <enumeration value='object'/>
     </restriction>
    </simpleType>
   </attribute>
  </complexType>
 </element>
</schema>
```

process.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<schema
        xmlns='http://www.w3.org/2001/XMLSchema'
        xmlns:reactor_common='reactor_common.dtd'
        xmlns:process='process.dtd'
        targetNamespace='process.dtd'>
 <import namespace='reactor_common.dtd' schemaLocation='reactor_common.xsd'/>
 <element name='process'>
  <complexType>
   <sequence>
    <element ref='reactor_common:id' minOccurs='0'/>
    <element ref='reactor_common:label'/>
    <element ref='reactor_common:description'/>
    <element ref='reactor_common:acl'/>
    <element ref='process:start_date'/>
    <element ref='process:end_date'/>
    <element ref='process:timers'/>
    <element ref='process:timer_ids'/>
    <element ref='process:state'/>
    <element ref='process:current_statuses'/>
    <element ref='process:precondition' minOccurs='0'/>
    <element ref='process:change_condition' minOccurs='0'/>
    <element ref='process:operands'/>
    <element ref='process:statuses'/>
    <element ref='process:policies'/>
    <element ref='process:superprocess'/>
    <element ref='process:subprocesses'/>
   </sequence>
   <attribute name='definition' use='required' type='boolean'/>
  </complexType>
 </element>
 <element name='start_date' type='string'/>
 <element name='timers'>
  <complexType>
   <sequence>
    <element ref='process:timer_spec' minOccurs='0' maxOccurs='unbounded'/>
   </sequence>
  </complexType>
 </element>
 <element name='timer_spec'>
  <complexType>
   <sequence>
    <element ref='process:timer_description'/>
    <element ref='process:event'/>
    <element ref='process:calendar' minOccurs='0'/>
    <choice minOccurs='0'>
     <element ref='process:schedule_expression'/>
     <element ref='process:schedule_date'/>
    </choice>
    <element ref='process:repeat_expression' minOccurs='0'/>
    <choice>
     <element ref='process:end_expression'/>
     <element ref='process:end_date'/>
     <element ref='process:repeat_count'/>
    </choice>
   </sequence>
  </complexType>
 </element>
```

```
<element name='timer_description' type='string'/>
<element name='event'>
 <complexType>
  <sequence>
   <element ref='process:event_name'/>
   <element ref='process:event_attribute' minOccurs='0' maxOccurs='unbounded'/>
  </sequence>
 </complexType>
</element>
<element name='event_name' type='string'/>
<element name='event_attribute'>
 <complexType>
  <sequence>
   <element ref='process:att_key'/>
   <element ref='process:att_value'/>
  </sequence>
 </complexType>
</element>
<element name='att_key' type='string'/>
<element name='att_value' type='string'/>
<element name='calendar' type='string'/>
<element name='schedule_expression' type='string'/>
<element name='schedule_date' type='string'/>
<element name='repeat_expression' type='string'/>
<element name='end_expression' type='string'/>
<element name='end_date' type='string'/>
<element name='repeat_count' type='int'/>
<element name='timer_ids'>
 <complexType>
  <sequence>
   <element ref='process:timer_id' minOccurs='0' maxOccurs='unbounded'/>
  </sequence>
 </complexType>
</element>
<element name='timer_id' type='string'/>
<element name='state'>
 <complexType>
  <attribute name='state_name' use='required'>
   <simpleType>
    <restriction base='string'>
     <enumeration value='unstarted'/>
     <enumeration value='started'/>
     <enumeration value='finished'/>
    </restriction>
   </simpleType>
  </attribute>
 </complexType>
</element>
<element name='current_statuses'>
 <complexType>
  <sequence>
   <element ref='reactor_common:object_reference' minOccurs='0'
      maxOccurs='unbounded'/>
  </sequence>
 </complexType>
</element>
<element name='precondition'>
 <complexType>
  <sequence>
   <element ref='process:condition' minOccurs='0'/>
```

```
   </sequence>
  </complexType>
 </element>
 <element name='change_condition'>
  <complexType>
   <sequence>
    <element ref='process:condition' minOccurs='0'/>
    <element ref='process:change' maxOccurs='unbounded'/>
   </sequence>
  </complexType>
 </element>
 <element name='condition'>
  <complexType>
   <sequence>
    <choice>
     <element ref='process:conjunction'/>
     <element ref='process:disjunction'/>
     <element ref='process:inversion'/>
     <element ref='process:process_state_equals'/>
     <element ref='process:process_has_status'/>
    </choice>
   </sequence>
  </complexType>
 </element>
 <element name='conjunction'>
  <complexType>
   <sequence>
    <element ref='process:condition' maxOccurs='unbounded'/>
   </sequence>
  </complexType>
 </element>
 <element name='disjunction'>
  <complexType>
   <sequence>
    <element ref='process:condition' maxOccurs='unbounded'/>
   </sequence>
  </complexType>
 </element>
 <element name='inversion'>
  <complexType>
   <sequence>
    <element ref='process:condition'/>
   </sequence>
  </complexType>
 </element>
 <element name='process_state_equals'>
  <complexType>
   <sequence>
    <element ref='reactor_common:object_reference'/>
    <element ref='process:state'/>
   </sequence>
  </complexType>
 </element>
 <element name='process_has_status'>
  <complexType>
   <sequence>
    <element ref='reactor_common:object_reference' minOccurs='2' maxOccurs='2'/>
   </sequence>
  </complexType>
 </element>
```

```
<element name='change'>
 <complexType>
  <sequence>
   <choice>
    <element ref='process:state_change'/>
    <element ref='process:status_addition'/>
    <element ref='process:status_removal'/>
   </choice>
  </sequence>
 </complexType>
</element>
<element name='state_change'>
 <complexType>
  <sequence>
   <element ref='process:state'/>
  </sequence>
 </complexType>
</element>
<element name='status_addition'>
 <complexType>
  <sequence>
   <element ref='reactor_common:object_reference'/>
  </sequence>
 </complexType>
</element>
<element name='status_removal'>
 <complexType>
  <sequence>
   <element ref='reactor_common:object_reference'/>
  </sequence>
 </complexType>
</element>
<element name='operands'>
 <complexType>
  <sequence>
   <element ref='reactor_common:object_reference' minOccurs='0'
      maxOccurs='unbounded'/>
  </sequence>
 </complexType>
</element>
<element name='statuses'>
 <complexType>
  <sequence>
   <element ref='reactor_common:object_reference' minOccurs='0'
      maxOccurs='unbounded'/>
  </sequence>
 </complexType>
</element>
<element name='policies'>
 <complexType>
  <sequence>
   <element ref='reactor_common:object_reference' minOccurs='0'
      maxOccurs='unbounded'/>
  </sequence>
 </complexType>
</element>
<element name='superprocess'>
 <complexType>
  <sequence>
   <element ref='reactor_common:object_reference' minOccurs='0'/>
```

```
   </sequence>
  </complexType>
 </element>
 <element name='subprocesses'>
  <complexType>
   <sequence>
    <element ref='reactor_common:object_reference' minOccurs='0'
       maxOccurs='unbounded'/>
   </sequence>
  </complexType>
 </element>
</schema>
```

operand.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<schema
        xmlns='http://www.w3.org/2001/XMLSchema'
        xmlns:reactor_common='reactor_common.dtd'
        xmlns:operand='operand.dtd'
        targetNamespace='operand.dtd'>
 <import namespace='reactor_common.dtd' schemaLocation='reactor_common.xsd'/>
 <element name='operand'>
  <complexType>
   <sequence>
    <element ref='reactor_common:id' minOccurs='0'/>
    <element ref='reactor_common:label'/>
    <element ref='reactor_common:description'/>
    <element ref='reactor_common:acl'/>
    <element ref='operand:operand_value'/>
    <element ref='reactor_common:associated_process'/>
   </sequence>
   <attribute name='visible_in_entire_subtree' use='required' type='boolean'/>
  </complexType>
 </element>
 <element name='operand_value' type='string'/>
</schema>
```

status.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<schema
        xmlns='http://www.w3.org/2001/XMLSchema'
        xmlns:reactor_common='reactor_common.dtd'
        xmlns:status='status.dtd'
        targetNamespace='status.dtd'>
 <import namespace='reactor_common.dtd' schemaLocation='reactor_common.xsd'/>
 <element name='status'>
  <complexType>
   <sequence>
    <element ref='reactor_common:id' minOccurs='0'/>
    <element ref='reactor_common:label'/>
    <element ref='reactor_common:description'/>
    <element ref='reactor_common:acl'/>
    <element ref='reactor_common:associated_process'/>
   </sequence>
   <attribute name='applicable_to_entire_subtree' use='required'
       type='boolean'/>
  </complexType>
```

```
 </element>
</schema>
```

policy.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<schema
        xmlns='http://www.w3.org/2001/XMLSchema'
        xmlns:reactor_common='reactor_common.dtd'
        xmlns:policy='policy.dtd'
        targetNamespace='policy.dtd'>
 <import namespace='reactor_common.dtd' schemaLocation='reactor_common.xsd'/>
 <element name='policy'>
  <complexType>
   <sequence>
    <element ref='reactor_common:id' minOccurs='0'/>
    <element ref='reactor_common:label'/>
    <element ref='reactor_common:description'/>
    <element ref='reactor_common:acl'/>
    <element ref='policy:event_name'/>
    <element ref='policy:event_attributes'/>
    <element ref='policy:policy_type'/>
    <element ref='policy:language' minOccurs='0'/>
    <element ref='policy:policy_definition'/>
    <element ref='policy:security_policy'/>
    <element ref='reactor_common:associated_process'/>
   </sequence>
  </complexType>
 </element>
 <element name='event_name' type='string'/>
 <element name='event_attributes'>
  <complexType>
   <sequence>
    <any minOccurs='0' maxOccurs='unbounded'/>
   </sequence>
  </complexType>
 </element>
 <element name='event_source'>
  <complexType>
   <sequence>
    <choice>
     <element ref='reactor_common:object_reference'/>
     <element ref='policy:service_name'/>
    </choice>
   </sequence>
  </complexType>
 </element>
 <element name='service_name' type='string'/>
 <element name='policy_type' type='string'/>
 <element name='language' type='string'/>
 <element name='policy_definition'>
  <complexType>
   <sequence>
    <choice>
     <element ref='policy:classname'/>
     <element ref='policy:URL'/>
     <element ref='policy:source_code'/>
```

```
    </choice>
   </sequence>
  </complexType>
 </element>
 <element name='security_policy' type='string'/>
 <element name='classname' type='string'/>
 <element name='URL' type='string'/>
 <element name='source_code' type='string'/>
</schema>
```

Appendix D: WSDL

These files were generated by IBM WebSphere Toolkit, version 2.4.

ReactorService.wsdl

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="ReactorService"
  targetNamespace="http://www.oakgrovesystems.com/ReactorService"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:interface="http://www.oakgrovesystems.com/ReactorService-interface"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:types="http://www.oakgrovesystems.com/ReactorService"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<import
    location="http://localhost:8080/ReactorServer/wsdl/ReactorService-
       interface.wsdl"
    namespace="http://www.oakgrovesystems.com/ReactorService-interface">
</import>
<service name="ReactorService">
  <documentation>IBM WSTK V2.4 generated service definition file</documentation>
  <port binding="interface:ReactorServiceBinding"
        name="ReactorServicePort">
    <soap:address location="http://localhost:8080/soap/servlet/rpcrouter"/>
  </port>
</service>
</definitions>
```

ReactorService-interface.wsdl

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="ReactorService"
  targetNamespace="http://www.oakgrovesystems.com/ReactorService-interface"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://www.oakgrovesystems.com/ReactorService-interface"
  xmlns:types="http://www.oakgrovesystems.com/ReactorService-interface/types/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<message name="InhandleRequestRequest">
  <part name="meth1_inType1" type="xsd:string"/>
</message>
<message name="OuthandleRequestResponse">
  <part name="meth1_outType" type="xsd:string"/>
</message>

<portType name="ReactorService">
  <operation name="handleRequest">
    <input message="tns:InhandleRequestRequest"/>
    <output message="tns:OuthandleRequestResponse"/>
  </operation>
</portType>

<binding name="ReactorServiceBinding"
         type="tns:ReactorService">
  <soap:binding style="rpc"
         transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="handleRequest">
    <soap:operation soapAction="urn:reactor-service"/>
    <input>
      <soap:body
         encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
         namespace="urn:reactor-service"
         use="encoded"/>
    </input>
    <output>
      <soap:body
         encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
         namespace="urn:reactor-service" use="encoded"/>
    </output>
  </operation>
</binding>
</definitions>
```

Appendix E: Result Codes

```
200 OK

400 Bad Request

404 Bad Parameters

405 Bad Request Type

406 Can Not Deliver Format

407 Could Not Parse Request

408 Bad Response

410 Unauthorized

411 Login Failed

412 Couldn't Authenticate Due to Server Configuration

413 Rejected by ACL

414 Duplicate object

420 Not Found

421 Invalid License

500 Internal Error

501 Not Implemented

502 Service Not Initialized

503 Service Temporarily Unavailable

504 Service Not Found

505 Failed Assertion

506 Version Not Supported
```

Appendix F: API JavaDocs

See the HTML JavaDocs included with the Reactor 5 distribution in this directory:

```
..\docs\api
```

Or online at:

http://www.oakgrovesystems.com/r5/r5api

Appendix G: Java Code Examples

Request Types, Java code examples

AddStatus

Sample Java code:

```
ReactorObjectId processId = ...
ReactorObjectId statusId = ...
String authToken = ...
AddStatus request = new AddStatus();
request.setProcessId(processId);
request.setStatusId(statusId);
request.setAuthToken(authToken);
ReactorProxy reactor = ...
ReactorResponse response = reactor.handleRequest(request);
if (response.failed()) {
    ...
}
```

CloneInstance

Sample Java code:

```
ReactorObjectId processId = ...
String authToken = ...
CloneInstance request = new CloneInstance();
request.setProcessId(processId);
request.setAuthToken(authToken);
ReactorProxy reactor = ...
ReactorResponse baseResponse = reactor.handleRequest(request);
if (baseResponse.failed()) {
    ...
}
CloneInstanceResponse response = (CloneInstanceResponse) baseResponse;
String instanceId = response.getInstanceId();
```

Create

Sample Java code:

```
Operand operand1 = ...
Operand operand2 = ...
Operand operand3 = ...
Set objects = new HashSet();
objects.add(operand1);
objects.add(operand2);
objects.add(operand3);
String authToken = ...
Create request = new Create();
request.setObjects(objects);
request.setAuthToken(authToken);
ReactorProxy reactor = ...
```

```
ReactorResponse baseResponse = reactor.handleRequest(request);
if (baseResponse.failed()) {
    ...
}
```

Delete

Sample Java code:

```
ReactorObjectId id = ...
String authToken = ...
Delete request = new Delete();
request.setId(id);
request.setType(Operand.class);
request.setAuthToken(authToken);
ReactorProxy reactor = ...
ReactorResponse response = reactor.handleRequest(request);
if (response.failed()) {
    ...
}
```

Get

Sample Java code:

```
ReactorObjectId id = ...
String authToken = ...
Get request = new Get();
request.setId(id);
request.setType(Operand.class);
request.setAuthToken(authToken);
ReactorProxy reactor = ...
ReactorResponse baseResponse = reactor.handleRequest(request);
if (baseResponse.failed()) {
    ...
}
GetResponse response (GetResponse) baseResponse;
Operand operand = (Operand) response.getObject();
```

Lock

Sample Java code:

```
ReactorObjectId objectId = ...
ACE lockAce = ...
String authToken = ...

Lock request = new Lock();
request.setId(objectId);
request.setLockACE(ace);
request.setEligibleRole("Eligible Participant");
request.setStandbyRole("Standby Participant");

ReactorProxy reactor = ...
ReactorResponse response = reactor.handleRequest(request);
if (response.failed()) {
    ...
}
```

Login

Sample Java code:

```
String username = ...
String password = ...
Login request = new Login();
request.setUsername(username);
request.setPassword(password);
ReactorProxy reactor = ...
ReactorResponse baseResponse = reactor.handleRequest(request);
if (baseResponse.failed()) {
    ...
}
LoginResponse response (LoginResponse) baseResponse;
String authToken = response.getAuthToken();
```

Logout

Sample Java code:

```
String authToken = ...
Logout request = new Logout();
request.setTokenToExpire(authToken);
request.setAuthToken(authToken);
ReactorProxy reactor = ...
ReactorResponse response = reactor.handleRequest(request);
if (response.failed()) {
    ...
}
```

QueryAllObjects

Sample Java code:

```
String authToken = ...
QueryAllObjects request = new QueryAllObjects();
request.setAuthToken(authToken);
ReactorProxy reactor = ...
ReactorResponse baseResponse = reactor.handleRequest(request);
if (baseResponse.failed()) {
    ...
}
QueryResponse response (QueryResponse) baseResponse;
Set objects = response.getObjects();
```

QueryProcessTree

Sample Java code:

```
ReactorObjectId processId = ...
String authToken = ...
QueryProcessTree request = new QueryProcessTree();
request.setId(processId);
request.setDepth(-1);
request.setIncludeSuperprocess(false);
request.setIncludeOperands(true);
request.setIncludeStatuses(true);
```

```
request.setIncludePolicies(true);
request.setAuthToken(authToken);
ReactorProxy reactor = ...
ReactorResponse baseResponse = reactor.handleRequest(request);
if (baseResponse.failed()) {
    ...
}
QueryResponse response (QueryResponse) baseResponse;
Set objects = response.getObjects();
```

QueryProcesses

Sample Java code:

```
ACE profile = new ACE(ACE.USER_TYPE, "bob", "Initiative Participant");
String authToken = ...
QueryProcesses request = new QueryProcesses();
request.setACE(profile);
request.setMustBeStarted(true);
request.setAuthToken(authToken);
ReactorProxy reactor = ...
ReactorResponse baseResponse = reactor.handleRequest(request);
if (baseResponse.failed()) {
    ...
}
QueryResponse response (QueryResponse) baseResponse;
Set objects = response.getObjects();
```

RemoveStatus

Sample Java code:

```
ReactorObjectId processId = ...
ReactorObjectId statusId = ...
String authToken = ...
RemoveStatus request = new RemoveStatus();
request.setProcessId(processId);
request.setStatusId(processId);
request.setAuthToken(authToken);
ReactorProxy reactor = ...
ReactorResponse response = reactor.handleRequest(request);
if (response.failed()) {
    ...
}
```

SetObjects

Sample Java code:

```
Operand operand1 = ...
Operand operand2 = ...
Operand operand3 = ...
Set objects = new HashSet();
objects.add(operand1);
objects.add(operand2);
objects.add(operand3);
String authToken = ...
SetObjects request = new SetObjects();
request.setObjectsToUpdate(objects);
```

```
request.setAuthToken(authToken);
ReactorProxy reactor = ...
ReactorResponse baseResponse = reactor.handleRequest(request);
if (baseResponse.failed()) {
    ...
}
```

Start

Sample Java code:

```
ReactorObjectId processId = ...
String authToken = ...
Start request = new Start();
request.setProcessId(processId);
request.setAuthToken(authToken);
ReactorProxy reactor = ...
ReactorResponse response = reactor.handleRequest(request);
if (response.failed()) {
    ...
}
```

Stop

Sample Java code:

```
ReactorObjectId processId = ...
String authToken = ...
Stop request = new Stop();
request.setProcessId(processId);
request.setAuthToken(authToken);
ReactorProxy reactor = ...
ReactorResponse response = reactor.handleRequest(request);
if (response.failed()) {
    ...
}
```

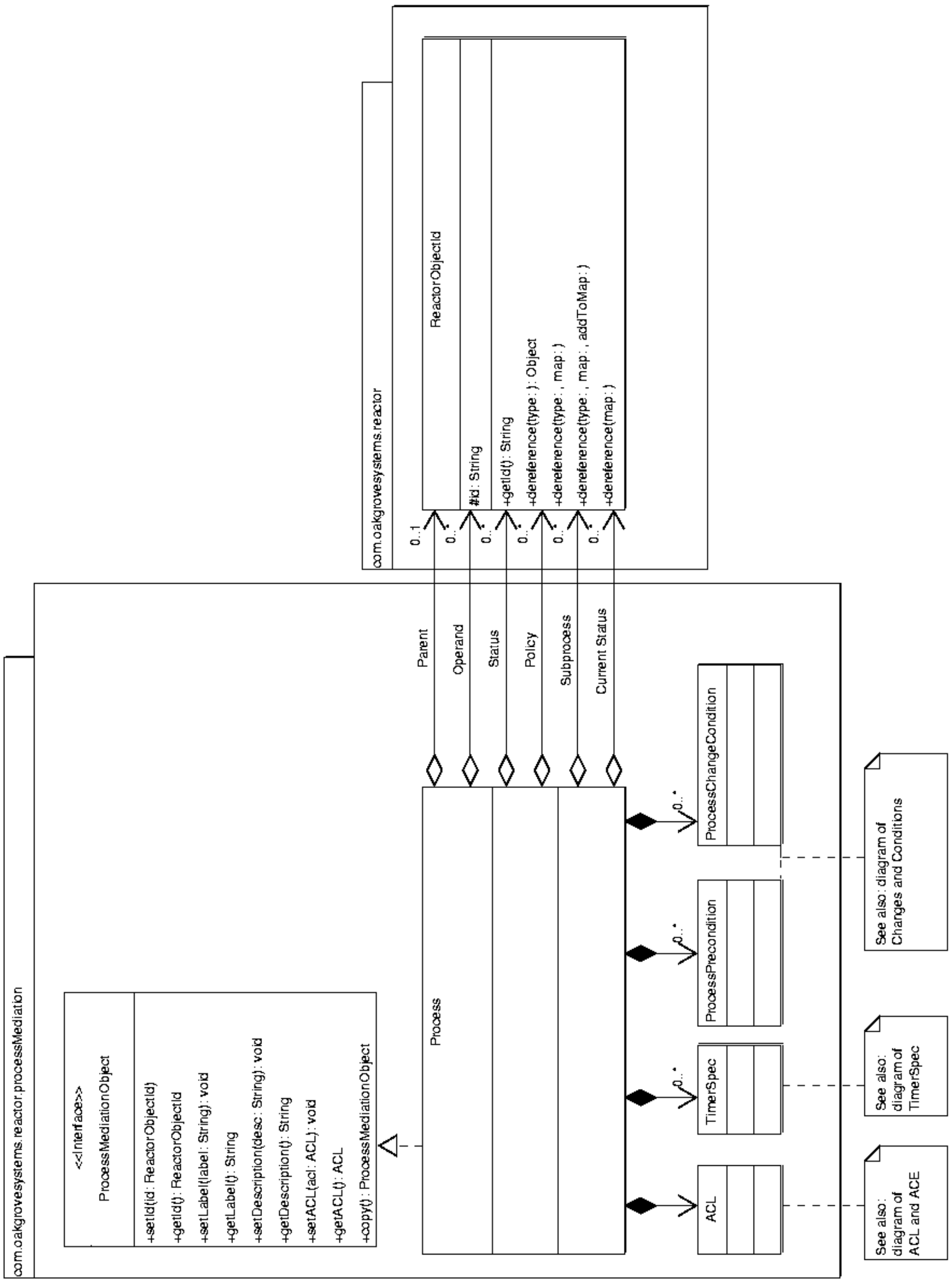
Unlock

Sample Java code:

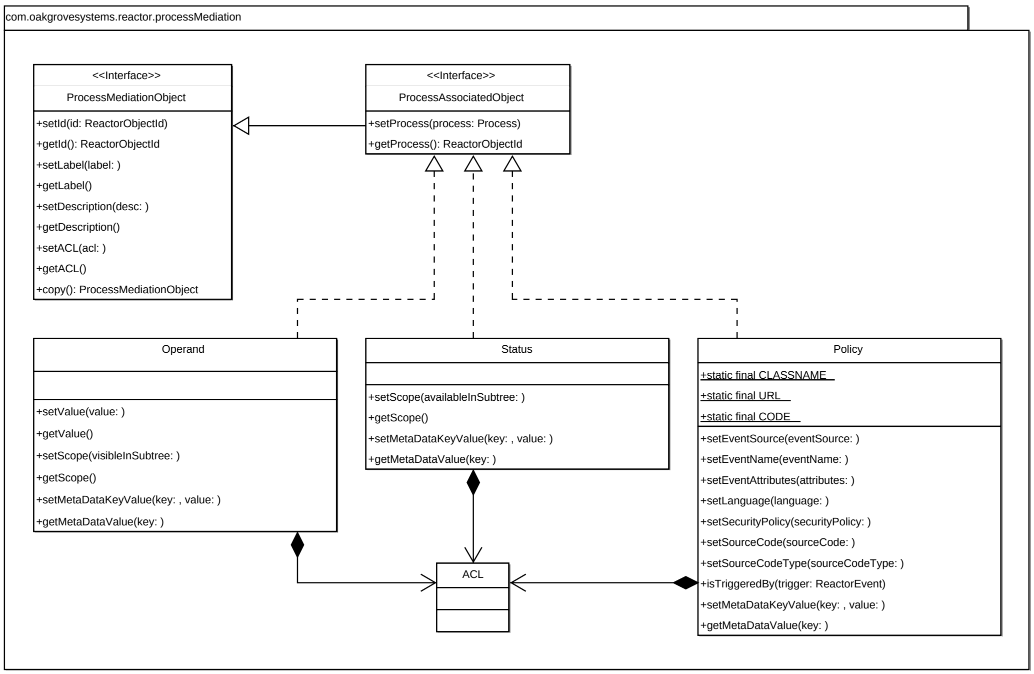
```
ReactorObjectId objectId = ...
ACE lockAce = ...
String authToken = ...

Unlock request = new Lock();
request.setId(objectId);
request.setLockACE(ace);
request.setEligibleRole("Eligible Participant");
request.setStandbyRole("Standby Participant");

ReactorProxy reactor = ...
ReactorResponse response = reactor.handleRequest(request);
if (response.failed()) {
    ...
}
```

Appendix H: Additional UML Diagrams

Process Details



Associated Objects



Changes and Conditions