Runtime Governance: Developer Guide

Chapter 1. Architecture Overview

This section will outline the architecture of the Runtime Governance architecture, prior to going into further details in the following sections.

1.1. Introduction

The main goal of this architecture is provide a modular and loose coupled solution for processing business activity information in real-time.

The architecture is comprised of the following four areas:

- Activity Collector - collect activity events as efficiently as possible
- Activity Server (and Store) - central activity event store and query
- Event Processor Network - generic event analysis, used to process the activity events
- Active Collections - active information management, used to post-process and cache information for end-user applications

The only mandatory part of this architecture could be considered the Activity Server, as it provides the central hub for storing and querying activity events. This means that the way in which events are processed, or presented to end users/applications could be replaced with other possibly more appropriate technology for a particular target environment.

Equally, the Event Processor Network and Active Collection mechanisms are information agnostic, so can be used to process and/or manage the presentation of any type of information.

1.2. Collection and Reporting

The first stage of the architecture performs the functions illustrated in this diagram:

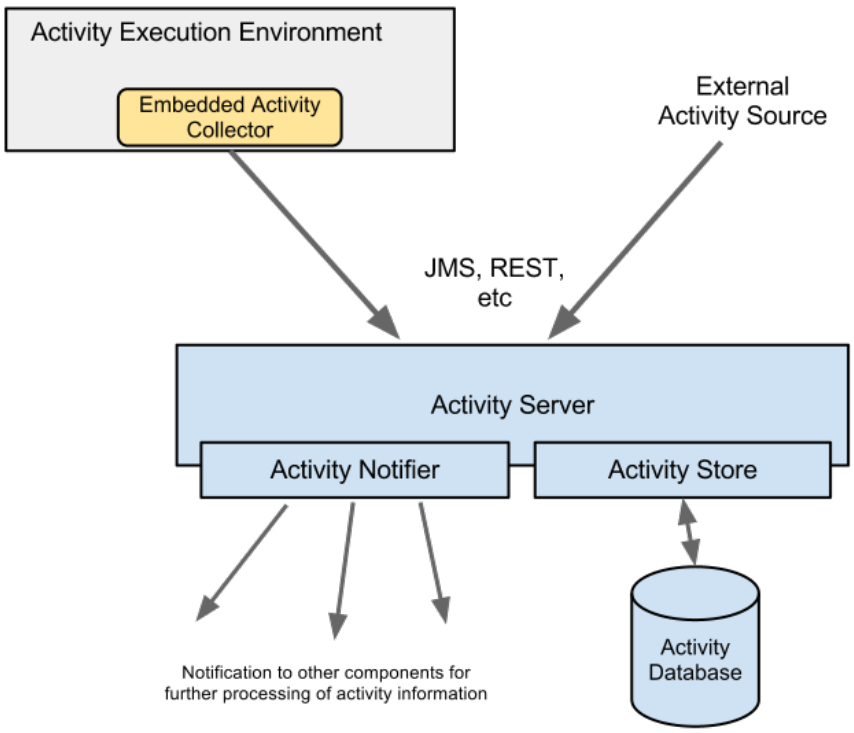

Figure 1.1.

1.2.1. Collection

The "Activity Collector" is an optional part of the architecture that is responsible for collecting information from the execution infrastructure as efficiently as possible.

The activity events associated with a particular thread are collected as a group, contained within an Activity Unit, to provide an implicit correlation of the activities that are associated with the same business transaction. Where relevant, the activity events may also be pre-processed to extract relevant context and property information prior to it being reported to the server.

Activity Units are then batched into further groups, and reported to the Activity Server at regular time intervals or if the batch gets too large.

Where the Activity Collector and Activity Server are co-located within the same execution environment, the Activity Units will be reported directly. Where the Activity Server is running remotely, then suitable connectors will be used to report the information. Current implementations exist for REST.

1.2.2. Reporting

The Activity Server provides a public API for reporting a list of Activity Units. This API can either be accessed directly (e.g. as a CDI component), or remotely via REST or JMS.

The Activity Server has three main responsibilities: * Ensure Ids are set and consistent * Store the events in a repository * Notify other interested modules

The last two responsibilities are discussed in the following sections.

1.2.3. Storage

This component simply records the activity events in a persistent store. A variety of implementations may be provided, including JPA, NoSQL variants, etc.

1.2.4. Notification

This component is simply an API used by other modules that are interested in being notified when activity events are reported.

1.3. Event Processing/Analysis

The following diagram illustrates how a node within an Event Processor Network functions to process the inbound event information.

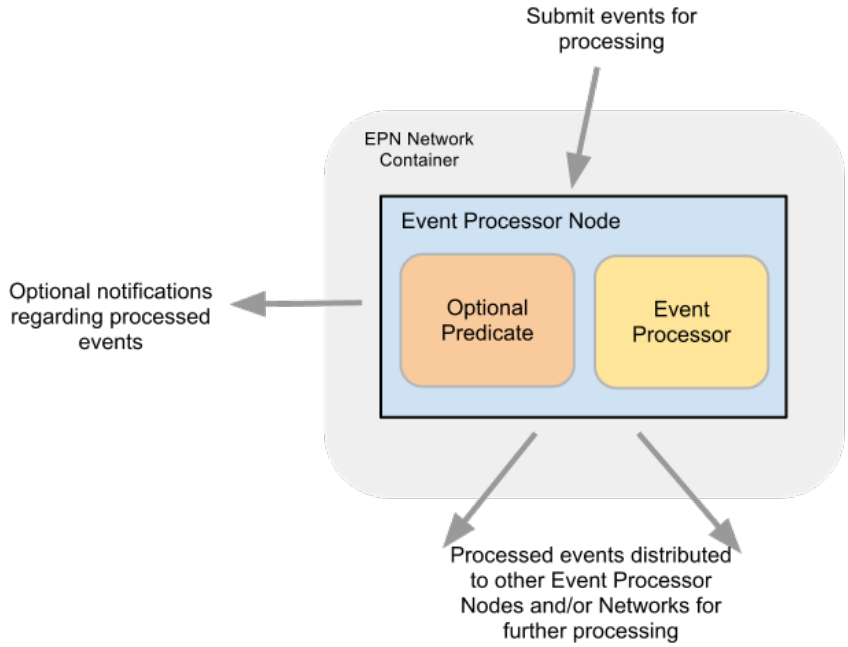

Figure 1.2.

The Event Processor Network (EPN) is a graph based mechanism for processing a series of events. In the context of the infrastructure, one or more networks can be registered to receive the activity information (as notifications) from the Activity Server and process it (filter, transform, analyse, etc) using whatever means is appropriate.

Each network defines a graph of nodes connected by links that transfer the results from the source node to the target node. The graphs can subscribe to event subjects, to identify the information they are interested in, and nominate the node(s) within the network that will process the information received on that subject. The nodes can also publish their results to event subjects, for other networks to further process - so this provides a decoupled way for networks to exchange information.

Each node defines an optional predicate, that can be used to determine whether the event is of interest, and an event processor to perform the actual task. An example of an "out of the box" event processor is one used to trigger rules (using Drools) to process the events.

The Event Processor Network (EPN) can be versioned, so that when a new version of a network is deployed, any events that are being processed by the old version will continue to be processed

using that network, while new events being dispatched to the network will be handled by the newer version. This overcomes any potential issues where the structure of the network changes between versions.

The nodes can also be configured to generate different levels of notification, which can be used by other applications/components to observe the information being processed through the network (e.g. events that passed the predicate evaluation, results from the event processor, etc.). These notifications are distributed to named "notification subjects", enabling observing components to remain decoupled from the details of which (or how many) networks/nodes are producing results for that subject.

1.4. Active Collections

The Active Collection mechanism provides a capability for storing processed events (and derived analysis information) in a manner that can be accessed by end users, as well as enable those end users/applications to be actively notified when any changes occur.

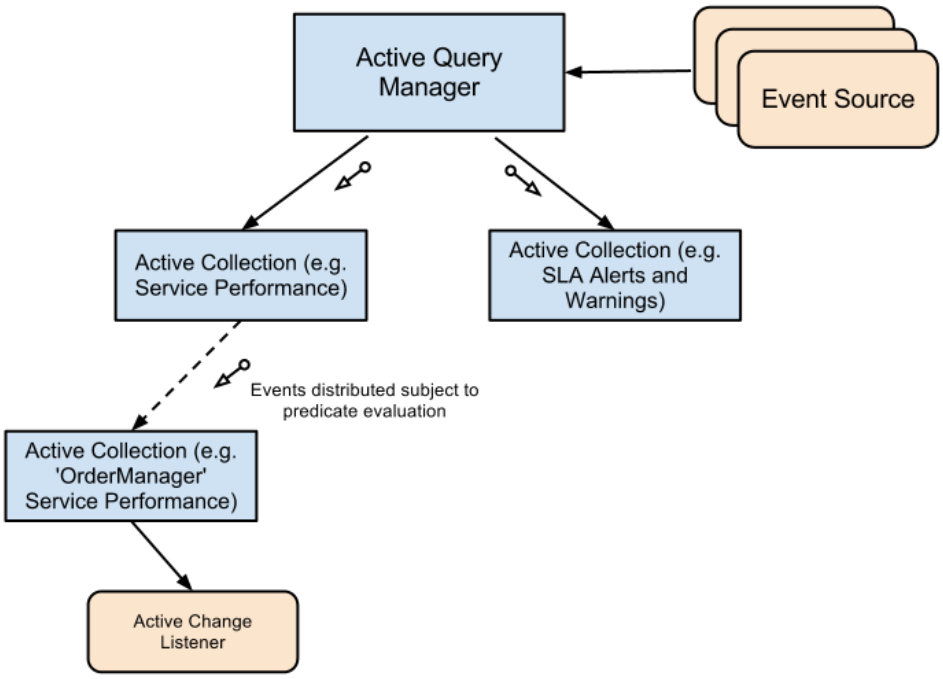

Figure 1.3.

The Active Collection mechanism is a variation on the standard collection concept, where interested parties can register interest in changes that occur to the contents of a collection (e.g. list, map, etc). This is one of the mechanisms that will be used to maintain information that is to be presented to users (e.g. via the Gadget Server).

The information within a particular Active Collection is managed by an Active Collection Source, which effectively acts as an adapter between the actual source of the information and the Active Collection. For example, an "out of the box" implementation of an Active Collection Source is provided to observe different types of information produced by an Event Processor Network.

The generic Active Collection Source implementation includes the ability to aggregate information which is then stored as a summary within the Active Collection, perform routine maintenance tasks and tidy up collection entries based on configured criteria (e.g. max size of the collection, max duration an item should exist in the collection, etc).

As well as creating these *top level* active collections, associated with configured Active Collection Sources, it is also possible to create derived (child) collections from these *top level* collections. These derived collections have a predicate that determines whether an entry in the parent collection is relevant to the child collection. This can be used to manage specific sub-sets, and essentially provides an *active query* mechanism, enabling interested clients to observe changes to that child collection.

Chapter 2. Reporting Activity

2.1. Activity Model

The section provides an overview of the Activity Model. This model defines the set of events (or situations) that can be reported to identify what is happening during the execution of a business transaction.

2.1.1. Activity Unit

The main (top level) model component is the Activity Unit. This component is a grouping capability to aggregate a set of activities (or situations) that relate to a particular transaction.

The Activity Unit has the following parts:

- id - this uniquely identifies the activity unit for historical retrieval purposes
- origin - this information identifies the environment in which the activities were recorded
- a set of contexts - provides contextual information to help relate the activities with other activities recorded in other units
- a group of activity types - the actual activities (or situations) that occurred

With the exception of the id field, these parts will be discussed in more detail below.

2.1.2. Origin

The Origin represents information about the source of the activities associated with the Activity Unit.

The information currently stored includes:

- principal - the user associated with the activities being performed, if available
- thread - can be useful in diagnostic situations in conjunction with the host information
- host - the host name
- node - the node name, for when the server is part of a cluster

2.1.3. Context

The context items represent information that can be used to correlate the activities within the unit against other Activity Units, as well as identify information information that may be useful when attempting to retrieve the unit.

The context has the following three pieces of information:

- type - the context type, explained below

- value - the value of the context information
- timeframe - optional value used with a *Link* context type, to identify the time period in which the context is valid

The different context types that can be defined are:

Type Constant	Description
Context.Type.Conversation	The conversation id, which can be used to correlate activities across service boundaries and is unique to a particular business transaction instance.
Context.Type.Endpoint	The endpoint id, which can be used to correlate activities within a service boundary (e.g. BPM process instance id), and which is also unique to a particular business transaction instance.
Context.Type.Message	The unique id for a message being sent and/or received. The message id may only be valid within the scope of an endpoint, as its value may not be carried with the message contents to the recipient. A common usage will be to correlate a response against the originating request within the same endpoint.
Context.Type.Link	This type represents a correlation between two activity events based on identify information that is only valid (i.e. unique) for a limited time period.

2.1.4. Activity Type

All activity events are derived from an Activity Type superclass. This class has the following information:

- activity unit id
- activity unit index
- timestamp
- principal
- a set of contexts
- a set of properties

The only piece of information that needs to be provided by the reporting component is the timestamp, and optionally some activity type specific contexts. The other information will be

initialized by the infrastructure prior to persisting the Activity Unit, as a way to enable the specific Activity Type instance to be located. This may be required during the analysis of Activity Units.

2.1.4.1. BPM

The BPM (Business Process Management) specific activity events are used to record the lifecycle and state transitions that occur when a business process (associated with a description language such as BPMN2 or WS-BPEL) is executed within a runtime engine, in support of a business transaction.

These business processes tend to be "long running", in that they handle multiple requests and responses over a period of time, all being correlated to the same process instance. This means that activities generated as a result of this execution must also be correlated to \(i) the specific XA transaction in which they are performed, (ii) the process instance that holds their state information in the BPM engine, and (iii) the conversation associated with the particular business transaction.

This does not mean that all Activity Units the contain activity information from the BPM engine need to have all three types of correlation information. For example, the initial Activity Unit for a business process instance may identify (i) and (ii), which will establish a unique process instance id. A subsequent Activity Unit may then define the same process id for (ii), as well as a conversation id (iii) that can then be used to tie any Activity Unit relates with the process instance id to that conversation - i.e. all Activity Units with the same process instance id become directly or indirectly correlated to the conversation id that may only be declared in some of the Activity Units.

Activity Type	Description
ProcessStarted	This activity type will be recorded when a process instance is initially started. Attributes include: process type, instance id and version
ProcessCompleted	This activity type will be recorded when a process instance completes. Attributes include: process type, instance id and status (either success or fail)
ProcessVariableSet	This activity type will be recorded when a process variable’s value is set or modified. Attributes include: process type, instance id and variable name/type/value

2.1.4.2. SOA

Activity Type	Description
RequestReceived and RequestSent	This activity type will be recorded when a service invocation (request) is received or sent.

Activity Type	Description
	message type, content and message id
ResponseReceived and ResponseSent	This activity type will be recorded when a service invocation returns. message type, content, message id and replyTo id (used to correlate the response to the original request)

2.2. Activity Collector

The *Activity Collector* is an embedded component that can be used to accumulate activity information from the infrastructure used in the execution of a business transaction. The activity information is then reported to the Activity Server (described in the following section) implicitly, using an appropriate Activity Logger implementation. The default Activity Logger implementation operates efficiently by providing a batching capability to send activity information to the server based either on a regular time interval, or a maximum number of activity units, whichever occurs first.

2.2.1. Finding the Activity Collector

The Activity Collector can be obtained using the Service Registry as follows:

```
ActivityCollector activityCollector=null;
....

ServiceRegistryUtil.addServiceListener(ActivityCollector.class, new
 ServiceListener<ActivityCollector>() {

    public void registered(ActivityCollector service) {
        activityCollector = service;
    }

    public void unregistered(ActivityCollector service) {
        activityCollector = null;
    }
});
```

2.2.2. Pre-Processing Activity Information

The ActivityCollector API provides a method to enable information associated with the activity event to be pre-processed, using configured information processors (see User Guide), to extract relevant properties that can be associated with the activity event.

These extracted properties can subsequently be used in further event analysis, to correlate the events and enable business relevant queries to be performed. The signature for this method is,

```
public String processInformation(String processor, String type,
        Object info, java.util.Map<String, Object> headers,
  ActivityType actType);
```

The *processor* parameter is an optional value that can be used to explicitly name the information processor to be used. If not specified, then all registered information processors will be checked to determine if they are relevant for the supplied information type.

The *type* parameter represents the information type. This can be in any form, as long as it matches the registered type defined in the information processor configuration.

The *info* parameter represents the actual information that will be processed.

The *headers* parameter represents any header information that may have accompanied the information (e.g. if the information was a message exchanged between two interacting parties).

The *actType* parameter represents the activity event that any extracted properties should be recorded against.

2.2.3. Validating the Activity Event

The activity collector provides a `validate` method that can be used to pre-process the activity event, using configured Activity Validators (see User Guide), before it is submitted to the activity server.

This mechanism can be used to process activity events in the execution environment, prior to it being distributed to the activity server which may be located on a separate server. It can also be used to identify invalid situations, resulting in an exception being thrown, which can be handled by the execution environment and used to block the business transaction associated with the activity event. An example of this usecase can be found in the "policy sync" quickstart.

2.2.4. Managing the Activity Scope

An Activity Scope is a way of grouping a range of different activity types, that will be reported to the activity server, into a single logical unit. It should generally represent the same scope as a XA transaction, to emcompass all of the work that was achieved within that transaction - and equally be discarded if the transaction is rolled back.

When the first activity is reported within the scope of a XA transaction, then the scope will automatically be started. When that transaction subsequently commits, the Activity Unit (i.e. the collection of activities accumulated during that scope) will be reported to the Activity Server.

However if activities are performed outside the scope of a XA transaction, then the component reporting the activity information can either explicitly start a scope, or just report the activity information.

If no scope exists, and an activity type is reported, then it will simply be reported to the activity server as a single event. The disadvantage of this approach is that it is less efficient, both in terms

of reporting due to the duplication of certain header information, and for subsequent analysis. Having multiple activity events defined in a single unit, related to the transaction, provides added value to inter-relating the different events - providing some implied correlation that would not exist if the events were independently reported to the Activity Server.

2.2.4.1. Starting the Scope

To start the scope, simply invoke the `startScope` method on the Activity Collector:

```
activityCollector.startScope();
```

Multiple invocations of the `startScope` method result in the creation of nested scopes.

If the application does not know whether a scope has already been started, and only wishes to start a single scope (i.e. as nested scopes are not wanted), then the following guard can be used:

```
boolean started=false;
if (!activityCollector.isScopeActive()) {
    activityCollector.startScope();
    started = true;
}
```

The `isScopeActive` method returns a boolean value to indicate whether the scope was previously started. If *true* is returned, then this component is also responsible for stopping the scope. If *false* is returned, then it means the scope has already been started, and therefore the component should **NOT** invoke the `endScope` method.

2.2.4.2. Ending the Scope

To stop the scope, simply invoke the `endScope` method on the Activity Collector:

```
if (started) {
    activityCollector.endScope();
}
```

If the `startScope` was invoked multiple times, the `endScope` method will end the most recently created (inner) scope.

2.2.5. Reporting an Activity Type

As described above, activity information is reported to the server as an Activity Unit, containing one or more actual activity events. The activity event is generically known as an Activity Type.

The Activity Collector mechanism removes the need for each component to report general information associated with the Activity Unit, and instead is only responsible for reporting the specific details associated with the situation that has occurred.

The set of different Activity Types that may be reported is outside the scope of this section of the documentation, and so for the purpose of illustration we will only be using a subset of the SOA related activity events. For more informaton on the available event types, please refer to the javadocs.

To report an event, simply create the specific Activity Type and invoke the `record` method:

```
org.overlord.rtgov.activity.model.RequestSent sentreq=new
 org.overlord.rtgov.activity.model.soa.RequestSent();

sentreq.setServiceType(serviceType);
sentreq.setOperation(opName);
sentreq.setContent(content);
sentreq.setMessageType(mesgType);
sentreq.setMessageId(messageId);

activityCollector.record(sentreq);
```

For certain types of event, it may also be appropriate to invoke an information processor(s) to extract relevant context and property information, that can then be associated with the activity event. This is achieved using the following:

```
Object modifiedContent=_activityCollector.processInformation(null,
          mesgType, content, headers, sentreq);

sentreq.setContent(modifiedContent);
```

The activity collector can be used to process relevant information, supplying the activity type to enable context and property information to be defined. The result of processing the information may be a modified version of the content, suitably obsfucated to hide any potentially sensitive information from being distributed by the governance infrastructure.

The first parameter to the *processInformation()* method is an optional information processor name - which can be used to more efficiently locate the relevant processor if the name is known.

2.2.6. Configuring an Activity Unit Logger

The Activity Unit Logger is the component responsible for logging the activity unit that is generated when the `endScope` method is invoked on the collector (either explicitly or implicitly by the XA resource manager).

This interface has three methods:

- init - this method initializes the activity unit logger implementation
- log - supplied the Activity Unit to be logged

- close - this method closes the activity unit logger implementation

2.2.6.1. Batched Activity Unit Logger (Abstract)

The Batched Activity Unit Logger is an abstract base class implementing the Activity Unit Logger interface. It provides the functionality to batch Activity Unit instances, and then forwarding them based on two properties:

- Maximum Time Interval - If the time interval expires, then the set of Activity Units will be sent.
- Maximum Unit Count - if the number of Activity Units reaches this max value, then the batch will be sent.

This implementation can be explicitly initialized when used in an embedded environment. If used within a JEE environment, then the `PostConstruct` and `PreDestroy` annotations enable it to be implicit initialized and tidied up when the concrete component's lifecycle is managed.

2.2.6.2. Activity Server Logger

This implementation of the Activity Unit Logger interface is derived from the Batched Activity Unit Logger, and therefore will send activity information in a batch periodically based on the configured properties. When the batch of Activity Units are sent, this implementation forwards them to an implementation of the Activity Server interface, injected explicitly or implicitly into the logger.

The Activity Server will be discussed in a subsequent section of this document. However, this can be used to either send the events directly to the Activity Server component, if co-located within the same server, or via a remote binding. For example,

```
import org.overlord.rtgov.activity.collector.ActivityCollector;
import
 org.overlord.rtgov.activity.collector.activity.server.ActivityServerLogger;
import org.overlord.rtgov.activity.server.rest.client.RESTActivityServer;

.....

RESTActivityServer restc=new RESTActivityServer();
restc.setServerURL(_activityServerURL);

ActivityServerLogger activityUnitLogger=new ActivityServerLogger();
activityUnitLogger.setActivityServer(restc);

activityUnitLogger.init();

_collector.setActivityUnitLogger(activityUnitLogger);
```

This shows a situation where an embedded Activity Collector is being initialized with an Activity Server Logger, which uses the REST Activity Server client implementation.

2.2.7. Configuring a Collector Context

The final component within the Collector architecture is the Collector Context. This interface provides the Activity Collector with information about the environment (e.g. principal, host, node, port), which can be used to complete the Origin information within an Activity Unit, as well as providing access to capabilities required from the environment (e.g. the Transaction Manager).

Each type of environment in which the collector may be used will provide an implementation of this interface. Depending upon the environment, this will either be implicitly injected into the Activity Collector, or be set explicitly using the setter method.

2.2.8. Simplified Activity Reporter for use by application components

Although the general Activity Collector mechanism can be used, as described in the previous sections, an injectable ActivityRecorder component is provided to enable applications to perform simple activity reporting tasks. Where injection is not possible, then a default implementation of the interface can be instantiated.

For example, the sample SwitchYard order management application uses this approach:

```
@Service(InventoryService.class)
public class InventoryServiceBean implements InventoryService {

    private final Map<String, Item> _inventory = new HashMap<String,
 Item>();

    private org.overlord.rtgov.client.ActivityReporter _reporter=
  new org.overlord.rtgov.client.DefaultActivityReporter();

    public InventoryServiceBean() {
        ....
    }

    @Override
    public Item lookupItem(String itemId) throws ItemNotFoundException {
        Item item = _inventory.get(itemId);

        if (item == null) {

            if (_reporter != null) {
                _reporter.logError("No item found for id '"+itemId+"'");
            }

            throw new ItemNotFoundException("We don't got any " + itemId);
        }

        ....
```

```
        return item;
    }
}
```

The ActivityReporter enables the application to perform the following tasks:

Method	Description
logInfo(String mesg)	Log some information
logWarning(String meg)	Log a warning
logError(String mesg)	Log an error
report(String type, Map<String,String> props)	Record a custom activity with a particular type and associated properties
report(ActivityType activity)	Record an activity

However this API cannot be used to control the scope of an ActivityUnit. It is expected that this would be handled by other parts of the infrastructure, so this API is purely intended to simplify the approach used for reporting additional incidental activities from within an application.

The maven dependency required to access the ActivityReporter is:

```
<dependency>
 <groupId>org.overlord.rtgov.integration</groupId>
 <artifactId>rtgov-client</artifactId>
 <version>${rtgov.version}</version>
</dependency>
```

2.3. Activity Server

The Activity Server is responsible for:

- Recording Activity Units describing the activities that occur during the execution of business transactions in a distributed environment.
- Query suport to retrieve previously recorded Activity Units

2.3.1. Recording Activity Units

The Activity Server can be used to record a list of Activity Units generated from activity that occurs durig the execution of a business transaction. The Activity Units represent the logical grouping of individual situations that occur within a transaction (e.g. XA) boundary.

This section will show the different ways this information can be recorded, using a variety of bindings.

Tip

Where possible, the Activity Collector mechanism described in the previous section should be used to aggregate and record the activity information, as this is more efficient that each system individually reporting events to the server.

2.3.1.1. Java API

First step is to obtain a reference to the Activity Server,

```
import org.overlord.rtgov.activity.server.ActivityServer;

....

private ActivityServer _activityServer=null;

....

// Get ActivityServer asynchronously
ServiceRegistryUtil.addServiceListener(ActivityServer.class, new
 ServiceListener<ActivityServer>() {

    public void registered(ActivityServer service) {
        _activityServer = service;
    }

    public void unregistered(ActivityServer service) {
        _activityServer = null;
    }
});

// Or synchronously
_activityServer =
 ServiceRegistryUtil.getSingleService(ActivityServer.class);
```

Once the reference to the Activity Server has been obtained, then call the `store` method to record a list of Activity Units.

```
import org.overlord.rtgov.activity.model.soa.RequestSent;
import org.overlord.rtgov.activity.model.ActivityUnit;

....

java.util.List<ActivityUnit> list=new .....;

RequestSent act=new RequestSent();
act.setServiceType(...);
```

```
...

list.add(act);

_activityServer.store(list);
```

2.3.1.2. REST Service

The Activity Server can be accessed as RESTful service, e.g.

```
import org.codehaus.jackson.map.ObjectMapper;
import org.overlord.rtgov.activity.model.ActivityUnit;

.....

java.util.List<ActivityUnit> activities=........
java.net.URL storeUrl = new java.net.URL(....);   // <host>/overlord-rtgov/
activity/store

java.net.HttpURLConnection connection = (java.net.HttpURLConnection)
 storeUrl.openConnection();

String userPassword = username + ":" + password;
String encoding =
 org.apache.commons.codec.binary.Base64.encodeBase64String(userPassword.getBytes());

connection.setRequestProperty("Authorization", "Basic " + encoding);

connection.setRequestMethod("POST");
connection.setDoOutput(true);
connection.setDoInput(true);
connection.setUseCaches(false);
connection.setAllowUserInteraction(false);
connection.setRequestProperty("Content-Type", "application/json");

java.io.OutputStream os=connection.getOutputStream();

ObjectMapper mapper=new ObjectMapper();    // Use jackson to serialize the
 activity units
mapper.writeValue(os, activities);

os.flush();
os.close();

java.io.InputStream is=connection.getInputStream();

byte[] result=new byte[is.available()];

is.read(result);
```

```
is.close();
```

See the REST API information in the docs folder of the distribution.

2.3.2. Retrieve an Activity Unit

The Activity Server can be used to retrieve a specific Activity Unit from the Activity Server. The Activity Unit represents a grouping of Activity Events that occurred within the same business transaction scope. This section will show the different ways this information can be queried, using a variety of bindings.

2.3.2.1. Java API

Once the reference to the Activity Server has been obtained (as shown previously), then invoke the `getActivityUnit` method to retrieve the required information.

```
import org.overlord.rtgov.activity.model.ActivityUnit;

....
String id="....";

ActivityUnit au=_activityServer.getActivityUnit(id);
```

2.3.2.2. REST Service

The Activity Server can be accessed as RESTful service, e.g.

```
import org.codehaus.jackson.map.ObjectMapper;
import org.codehaus.jackson.type.TypeReference;
import org.overlord.rtgov.activity.model.ActivityUnit;

.....

java.net.URL queryUrl = new java.net.URL(....);   // <host>/overlord-rtgov/
activity/unit?id=<id>

java.net.HttpURLConnection connection = (java.net.HttpURLConnection)
 queryUrl.openConnection();

String userPassword = username + ":" + password;
String encoding =
 org.apache.commons.codec.binary.Base64.encodeBase64String(userPassword.getBytes());

connection.setRequestProperty("Authorization", "Basic " + encoding);

connection.setRequestMethod("GET");
connection.setDoOutput(true);
connection.setDoInput(true);
```

```
connection.setUseCaches(false);
connection.setAllowUserInteraction(false);
connection.setRequestProperty("Content-Type", "application/json");

java.io.InputStream is=connection.getInputStream();

ActivityUnit au = mapper.readValue(is, ActivityUnit.class);

is.close();
```

See the REST API documentation in the docs folder of the distribution.

2.3.3. Retrieve a list of Activity Events

The Activity Server can be used to query a list of Activity Type (events) from the Activity Server. This section will show the different ways this information can be queried, using a variety of bindings.

2.3.3.1. Java API

Once the reference to the Activity Server has been obtained (as described previously), then the `getActivityTypes` method can be invoked to obtain the list of events.

```
import org.overlord.rtgov.activity.model.ActivityUnit;
import org.overlord.rtgov.activity.model.Context;

....
String value="...."; // Conversation id
Context context=new Context(Context.Type.Conversation, value);

java.util.List<ActivityType> list=_activityServer.getActivityTypes(context);

// or, if wanting to define a time range

long startTime=...;
long endTime=...;

java.util.List<ActivityType> list=_activityServer.getActivityTypes(context,
 startTime, endTime);
```

2.3.3.2. REST Service

The Activity Server can be accessed as RESTful service, e.g.

```
import org.codehaus.jackson.map.ObjectMapper;
import org.codehaus.jackson.type.TypeReference;
import org.overlord.rtgov.activity.model.ActivityType;
```

```
.....

java.net.URL queryUrl = new java.net.URL(....);   // <host>/overlord-rtgov/
activity/events?type=<type>&value=<value>

// Note: add optional query parameters
 &from=<fromTimestamp>&to=<toTimestamp> to define a time frame

java.net.HttpURLConnection connection = (java.net.HttpURLConnection)
 queryUrl.openConnection();

String userPassword = username + ":" + password;
String encoding =
 org.apache.commons.codec.binary.Base64.encodeBase64String(userPassword.getBytes());

connection.setRequestProperty("Authorization", "Basic " + encoding);

connection.setRequestMethod("GET");
connection.setDoOutput(true);
connection.setDoInput(true);
connection.setUseCaches(false);
connection.setAllowUserInteraction(false);
connection.setRequestProperty("Content-Type", "application/json");

java.io.InputStream is=connection.getInputStream();

java.util.List<ActivityType> activities = mapper.readValue(is, new
 TypeReference<java.util.List<ActivityType>>() {});

is.close();
```

See the REST API documentation in the docs folder of the distribution.

Chapter 3. Event Processing

The EventProcessor, and supporting components, can be used either directly within the Activity Collection mechanism or from nodes within an Event Processor Network. This section of the Developer Guide will discuss how custom Predicates and Event Processors are defined.

3.1. Custom Predicate

The `org.overlord.rtgov.ep.Predicate` abstract class is responsible for determining whether an event is suitable to be processed by a particular node within the Event Processor Network.

To create a custom implementation simply derive a class from the Predicate abstract class. This class provides the following methods:

Method	Description
void init()	This method is called when the predicate is first initialized as part of the Event Processor Network. A custom implementation does not need to override this method if not required.
boolean evaluate(Object event)	This method determines whether the supplied event should be processed by the node.

3.2. Custom Event Processor

The `org.overlord.rtgov.ep.EventProcessor` abstract class is responsible for processing an event routed to a particular node within the Event Processor Network.

To create a custom implementation simply derive a class from the EventProcessor abstract class. This class provides the following methods:

Method	Description
java.util.Map<String,Service> services getServices()	This method returns the map of services available to the Event Processor.
void setServices(java.util.Map<String,Service> services)	This method sets the map of services available to the Event Processor.
java.util.Map<String,Object> services getParameters()	This method returns the map of parameters available to the Event Processor.
void setParameters(java.util.Map<String,Object> parameters)	This method sets the map of parameters available to the Event Processor.
void init()	This method is called when the event processor is first initialized as part of

Method	Description
	the Event Processor Network. A custom implementation does not need to override this method if not required.
Serializable process(String source, Serializable event, int retriesLeft) throws Exception	This method processes the supplied event, indicating the source of the event and how many retries are left (so that suitable error handling can be performed in no more retries remain.

3.3. Custom Services

The `org.overlord.rtgov.common.service.Service` abstract class is used to provide services for use by event processors, e.g. CacheManager.

To create a custom implementation simply derive a class from the Service abstract class. This class provides the following methods:

Method	Description
void init()	This method is called when the service is first initialized. A custom implementation does not need to override this method if not required.

3.4. Packaging

The custom predicate and/or event processor implementations must be available to the classloader when an Event Processor Network or Activity Validator referencing the implementations is loaded. This can either be achieved by packaging the implementations with the Event Processor Network or Activity Validator configuration, or by installing them in a common location used by the container in which the Event Processor Network/Activity Validator is being loaded.

Chapter 4. Active Collections

The Active Collection mechanism provides a means of actively managing a collection of information. For a more details explanation of the mechanism, see the User Guide.

This section explains how to:

- implement an Active Collection Source, which can be used to subscribe to a source of information which can result in data being inserted, updated and removed from an associated active collection.
- implement an Active Change Listener that can associated with an Active Collection Source, and automatically notified of changes to an associated Active Collection
- write a custom application for accessing Active Collections

4.1. Active Collection Source

The Active Collection Source can be considered the adapter between the actual source of events/ information and the Active Collection. The Active Collection Source is responsible for managing the insertion, update and deletion of the objects within the associated Active Collection, based on situations that occur in the source.

An example of a derived Active Collection Source implementation, that is packaged with the infrastructure, can be used to listen for events produced by nodes in an Event Processor Network and insert these events in the Active Collection.

To create a new type of Active Collection Source, simply derive a class from the `org.overlord.rtgov.active.collection.ActiveCollectionSource` class and implement the following methods:

Method	Description
void init()	This method is invoked when the Active Collection Source is registered, and should be used to create the *subscription* to the relevant source of information. The implementation of this method MUST call the init() method on the super class first.
void close()	This method is invoked when the Active Collection Source is unregistered, and should be used to unsubscribe from the source of information. The implementation of this method MUST call the close() method on the super class first.

When a situation occurs on the source, that requires a change in the associated Active Collection, then the derived implementation can call one of the follow methods on the Active Collection Source:

Method	Description
public void insert(Object key, Object value)	This method is called to insert a new element into the collection. The value is the information to be inserted. The key is potentially optional, depending on the nature of the active collection: List - the key is optional. If specified, then it MUST be an integer representing the index where the value should be inserted. Map - the key represents the map key to be associated with the value, and is therefore not optional.
public void update(Object key, Object value)	This method is called to update an existing element within the collection. The value is the information to be updated. The key is potentially optional, depending on the nature of the active collection: List - the key is optional. If specified, then it MUST be an integer representing the index of the value to be updated. If not specified, then the value will be used to locate the index within the list. Map - the key represents the map key associated with the value, and is therefore not optional.
public void remove(Object key, Object value)	This method is called to remove an element from the collection. The value is the information to be updated. The key is potentially optional, depending on the nature of the active collection: List - the key is optional. If specified, then it MUST be an integer representing the index of the value to be removed. If not specified, then the value will be used to locate the index within the list.

Method	Description
	Map - the key represents the map key associated with the value, and is therefore not optional. However in this situation the value is optional.

4.2. Active Change Listeners

This section explains how to implement a listener to deal with changes that occur within an Active Collection.

The first sub-section details with general implementations of this interface, that may be used within custom applications. The second sub-section will deal with a specific type of listener that can be configured with an Active Change Source (discussed in the previous section), and automatically initialized when the Active Change Source is registered.

4.2.1. Active Change Listener

The `org.overlord.rtgov.active.collection.ActiveChangeListener` interface can be implemented by any component that is interested in being informed when a change occurs to an associated Active Collection. The Active Collection API supports add and remove methods to register and unregister these active change listeners.

The methods that need to be implemented for an active change listener are:

Method	Description
void inserted(Object key, Object value)	Called when a new value is inserted into the collection, with the key being dependent upon the type of collection: List - the key will be the index Map - the key will be the key information used in the map's key/value pair
void updated(Object key, Object value)	Called when an existing value is updated within the collection, with the key being dependent upon the type of collection: List - the key will be the index Map - the key will be the key information used in the map's key/value pair
void removed(Object key, Object value)	Called when an existing value is removed from the collection, with the key being dependent upon the type of collection: List - the key will be the index

Method	Description
	Map - the key will be the key information used in the map's key/value pair

4.2.2. Abstract Implementation

If the active change listener implementation is derived from the `org.overlord.rtgov.active.collection.AbstractActiveChangeListener` abstract class then it can be registered with the Active Collection Source configuration, and automatically initialized when the source is registered.

The benefit of this approach is that it does not require the user to write custom code to register the Active Collection Listener against the Active Collection.

An example of this type of implementation is the `org.overlord.rtgov.active.collection.jmx.JMXNotifier` which automatically generates JMX notifications when an object is added to the associated active collection.

The implementations derived from this abstract active change listener implementation are no different from order active change listener implementations, with the exception that they can be serialized as part of the Active Collection Source configuration, and they support lifecycle methods for initialization and closing:

Method	Description
void init()	This method can be overridden to initialize the active change listener implementation. The super class init() method MUST be called first.
void close()	This method can be overridden to close the active change listener implementation. The super class close() method MUST be called first.

4.3. Accessing Active Collections

This section explains how to:

- retrieve an existing active collection
- create a derived active collection
- register for active change notifications

4.3.1. Retrieve an Active Collection

There are two ways to retrieve an active collection.

4.3.1.1. Directly accessing the ActiveCollectionManager

As discussed in a previous section, Active Collections are created as a bi-product of registering an Active Collection Source. The Active Collection Source is registered with an Active Collection Manager, which creates the collection to be updated from the source. This Active Collection then becomes available for applications to retrieve from the manager, for example:

```
import org.overlord.rtgov.active.collection.ActiveCollectionManager;
import org.overlord.rtgov.active.collection.ActiveCollectionManagerAccessor;
import org.overlord.rtgov.active.collection.ActiveList;

.....

ActiveCollectionManager
 acmManager=ActiveCollectionManagerAccessor.getActiveCollectionManager();

ActiveList list = (ActiveList)
          acmManager.getActiveCollection(listName);
```

This is the approach used to retrieve what can be considered "top level" active collections. These are the collections directly maintained by the Active Collection Manager, each with an associated Active Collection Source defining the origin of the collection changes. The following section shows how further active collections can be derived from these "top level" collections, to refine the information.

The maven dependency required to access the ActiveCollectionManager and active collections is:

```
  <dependency>
   <groupId>org.overlord.rtgov.active-queries</groupId>
   <artifactId>active-collection</artifactId>
   <version>${rtgov.version}</version>
   <scope>provided</scope>
  </dependency>
```

4.3.1.2. Injectable Collection Manager

The other approach is aimed at simplifying the use of active collections from within a client application. It offers a simple API, and associated default implementation, that can be injected using CDI. Under the covers, it simply performs the same tasks as described in the previous section.

```
    @Inject
    private org.overlord.rtgov.client.CollectionManager
 _collectionManager=null;

    private org.overlord.rtgov.active.collection.ActiveMap _principals=null;
```

```
    protected void init() {

        if (_collectionManager != null) {
            _principals = _collectionManager.getMap(PRINCIPALS);
        }

        .......
    }
```

If injection is not possible (e.g. when using SwitchYard Auditors), then a default implementation can be directly instantiated with the class `org.overlord.rtgov.client.DefaultCollectionManager`.

The maven dependencies required to access the CollectionManager, and the subsequent active collections, are:

```
<dependency>
 <groupId>org.overlord.rtgov.integration</groupId>
 <artifactId>rtgov-client</artifactId>
 <version>${rtgov.version}</version>
</dependency>
<dependency>
 <groupId>org.overlord.rtgov.active-queries</groupId>
 <artifactId>active-collection</artifactId>
 <version>${rtgov.version}</version>
 <scope>provided</scope>
</dependency>
```

4.3.2. Create a Derived Active Collection

The "top level" active collections defined in the previous section reflect the information changes as identified by their associated Active Collection Source. However in some situations, only a subset of the information is of interest to an application. For these situations, it is possible to *derive* a child active collection by specifying:

- parent - the parent collection from which the child may be derived. Although this will generally be the name of a "top level" collection, it is possible to derive a collection from another child collection, enabling a tree to be formed.
- predicate - a predicate is specified to determine whether information in a parent collection (and subsequently its changes), are relevant to the child collection.
- properties - used to initialize the derived collection.

Currently the only property that can be set is a boolean named *active*, which defaults to true.

If the *active* property is true, then when a child collection is initially created, the predicate will be used to filter the contents of the parent collection to identify the initial subset of values that are

relevant for the child collection. Once initialized, the child collection effectively subscribes to the change notifications of the parent collection, and uses the predicate to determine whether the change is applicable, and if so, applies the change to the child collection.

If the *active* property is false, then whenever the derived collection is queried, the predicate will be applied to the parent collection to obtain the current set of results. This configuration should only be used where the predicate is based on volatile information, and therefore the results in the derived collection would be changing independently of changes applied to the parent collection.

```
import org.overlord.rtgov.active.collection.predicate.Predicate;
import org.overlord.rtgov.active.collection.ActiveCollectionManager;
import org.overlord.rtgov.active.collection.ActiveList;

.....

Predicate predicate=.....;

ActiveList parent = (ActiveList)acmManager.getActiveCollection(parentName);

if (parent != null) {
 java.util.Map<String,Object> properties=.....;

        alist = (ActiveList)acmManager.create(childName,
                    parent, predicate, properties);
}
```

4.3.3. Register for Active Change Notifications

Once an Active Collection has been retrieved (or created in the case of a child collection), then the information can be accessed using methods appropriate to the collection type, e.g. list or map.

However being active collections, an important source of information is the change notifications, to enable the application to understand what changes are occuring and when.

To receive change notifications, the application needs to register an Active Change Listener (discussed in the previous sections). This can be achieved using the *`addActiveChangeListener`* method on the collection, and simularly use the *`removeActiveChangeListener`* method to unregister for change notifications.

For example,

```
import org.overlord.rtgov.active.collection.ActiveList;
import org.overlord.rtgov.active.collection.ActiveChangeListener;

.....

ActiveList list=.....;
```

```
list.addActiveChangeListener(new ActiveChangeListener() {
    public void inserted(Object key, Object value) {
        ....
    }
    public void updated(Object key, Object value) {
        ....
    }
    public void removed(Object key, Object value) {
        ....
    }
});
```