



Developer's Guide

Table of Contents

Developer's Guide	1
Table of Contents	2
Overview	3
Threading Model	3
Installing Logical Components.....	3
Describing your RLC.....	4
Initialization	5
Browseable.....	5
Registering Subscriptions.....	5
Implementing the Interfaces	6
XML Examples	7
Conclusions	8

Introduction

Welcome to Rhombus IM Development Edition. This Guide is intended to help you take advantage of the Rhombus IM API sets to build sophisticated applications using our instant messaging technology.

To use the Development Edition, you must be using the Java programming language. Java version 1.4 is a requirement.

Overview

Rhombus IM is designed using a JMS (Java Message Service) publish/subscribe backbone. This design lends itself naturally to having many independent logical components providing messaging functionality in an asynchronous way. For the purposes of this document, we'll refer to logical components as RLCs (Rhombus Logical Components).

Most development tasks with the Rhombus IM API will revolve around creating and managing RLCs. These are defined below.

An **agent** is an RLC that does not have a lifecycle. It is the simplest component and simply registers its interest in a set of messages or queries, and the appropriate call is made on the object when these events occur.

A **service** is a more sophisticated RLC that supports a service lifecycle. A service can be initialized with user-defined properties at setup time, can be started and stopped (if allowed by the administration console) and can optionally have an identity on the system so that it can be directly addressed. Services are typically used for IM gateway providers and other sophisticated logic.

Threading Model

Rhombus IM uses a session-interleaved threading model, which synchronizes requests from a single user session while at the same time allowing a thread pool to work on multiple sessions in parallel. This allows Rhombus IM to take advantage of I/O latency and multiprocessor systems.

When you are designing your RLC, keep in mind that your logic may be called from multiple threads, so your logic must be **thread-safe**. However, you may make the assumption that two requests for the same user session will not occur simultaneously.

Installing Logical Components

The Rhombus IM logical framework is dictated by an XML descriptor located in the Rhombus IM installation directory. Rhombus IM ships with several versions of a logical framework with different sets of functionality trading off performance vs. features. When you add your RLC to the XML descriptor, make sure you are editing the correct file.

The Rhombus IM logical descriptor is defined by a DTD given in the file rhombus-logic.dtd. The DTD looks like this:

```
<!-- Rhombus IM 3.x logic XML DTD -->
<!ELEMENT rhombus-logic (logic | service)+>
```

```
<!ELEMENT logic          (presence?, message*, chatroom*, iq*)+>
<!ATTLIST logic          class      CDATA    #REQUIRED
                          name      CDATA    #REQUIRED>

<!ELEMENT service       (presence?, message*, chatroom*, iq*,
browseable*, init*)+>
<!ATTLIST service       class      CDATA    #REQUIRED
                          name      CDATA    #REQUIRED
                          jid      CDATA    #REQUIRED>

<!ELEMENT presence      EMPTY>

<!ELEMENT message       EMPTY>
<!ATTLIST message       domain    CDATA    #IMPLIED
                          jid      CDATA    #IMPLIED>

<!ELEMENT chatroom      EMPTY>
<!ATTLIST chatroom      jid      CDATA    #IMPLIED>

<!ELEMENT iq            EMPTY>
<!ATTLIST iq            namespace CDATA    #REQUIRED
                          to      CDATA    #REQUIRED>

<!ELEMENT browseable    EMPTY>
<!ATTLIST browseable    category  CDATA    #REQUIRED
                          subtype  CDATA    #REQUIRED>

<!ELEMENT init          (#PCDATA)>
<!ATTLIST init          key      CDATA    #REQUIRED>
```

Agents are defined by the **logic** tag, and services are defined by the **service** tag. Your rhombus root block contains a list of all agents and services to be registered in the infrastructure.

Describing your RLC

Every RLC listed in the logic XML file is instantiated once, according to the instructions given in the tags, and subscribed to a set of messages that it has expressed interest in. This section will elaborate on the process to register these actions.

The primary tags used to register an RLC are the **logic** and **service** tags. The logic tag describes an agent, and the service tag describes a service. Each of these has two required attributes, **class** and **name**. The class tag describes the Java class that is instantiated to implement the actions. The instantiated class must implement several interfaces (depending on its subscriptions) which will be described below. The name tag is simply a friendly, human-readable name that can be displayed in the administration console and to end-users, potentially.

A service requires an additional component, a JID (Jabber ID). This JID is used to identify the service when it is the target of requests for service. The JID may correspond to a user registered in the system, in which case the service appears in the User Directory, can be added to a user's roster, and messaged like any other human user in the system.

Initialization

The **init** tags are used to initialize your RLC service. The set of all keys provided by init tags and their values are used to create a **Properties** that is the parameter to the initialize call in the **Service** interface. If your RLC is not a service, you cannot use initialization tags.

Browseable

You can make your service appear in the Rhombus IM server browse results by adding one or more browseable tags. The browseable tags allow you to expose your service to end-users and specify information about the type of service that is provided by your RLC. Supported namespace information is automatically populated from the IQ tags that your service registers.

Your RLC must be a service to use this tag.

Registering Subscriptions

Your RLC can subsequently register its interest in various messages that may be passing through the system. All messages are up for grabs, it's up to you to filter them to your needs.

Messages

You can subscribe to messages passing through the infrastructure in three ways. One is to subscribe to all messages from everyone to everyone else. Be careful what you do in your processing block so that you do not add significant latency to messages passing through the server. This mode is triggered with a message tag with no attributes.

Another option is to subscribe to messages for a single domain. This is accomplished using the domain attribute. This setting is useful for IM gateways and other large scale message replication services.

The third option is to subscribe to messages for a single JID. In these cases the JID will likely correspond to the JID of the service itself. This is useful for creating bots.

Chatrooms

Similar to the message subscription block above, the chatroom tag subscribes your RLC to chatroom traffic. You can subscribe to either all chatroom traffic, or just the traffic for a particular room.

Presence

Subscribing to presence will inform you when users log on or off the system, or change their presence. Presence subscriptions cannot be filtered in this version.

IQ Namespaces

The IQ tag allows you to register interest in particular IQ (info-query) messages. Adding this tag requires you to implement an additional interface, and due to the query semantics imposed by the XMPP protocol, you must participate in a handling chain with other IQ handlers. The server can respond to each IQ only once, so your RLC will be added to the end of a handling chain. If a

handler registered before yours fully handles an IQ message, your RLC will never see it. The order of the XML file allows you to modify which handlers see messages before others.

You register your RLC for an IQ message with a single namespace and a filter for recipients of the IQ message. You can use the value "*" to indicate that all destinations are desired, or a different value, such as the JID of your service.

Implementing the Interfaces

Once you have decided the set of messages that your component will receive, you must ensure that your RLC implementation provides the correct interfaces to handle all of the messages that will be coming its way.

```
com.rhombus.jabber.protocol.JabberLogicHandler
```

This interface is required for **message**, **chatroom** and **presence** tags. It is the most rudimentary of all of the logical interfaces, with just two methods. The **onMessage** method is called when messages arrive. The **requiresAuthenticatedContext** method is used to potentially shortcut non-authentic methods from causing costly exceptions to be thrown by **onMessage**. Please refer to the Javadocs for more information on these methods.

You can use the AuthenticatedLogicAdapter abstract class as a starting point for your implementations. It will automatically reject any messages from unauthenticated sources and perform the requisite casts to reduce your implementation's complexity.

```
com.rhombus.jabber.protocol.InfoQueryHandler  
com.rhombus.jabber.protocol.UnauthenticatedInfoQueryHandler
```

These interfaces are required for RLCs registered for IQ namespaces. The Javadoc has detailed information about the semantics of using these handler interfaces.

```
com.rhombus.jabber.protocol.Service
```

The service interface is the most complex of the above interfaces, and provides a lifecycle for RLCs. The service interface extends the JabberLogicHandler, above.

There are several abstract base classes that provide a good starting point for your service implementations. These include `com.rhombus.jabber.agents.AbstractService`, which caches initialization arguments for later use, modifies the service presence on start and stop, and filters out unauthentic messages. Only one method is required to implement if you choose to extend this class.

Another choice is appropriate for creating an IM Gateway. The `AbstractGatewayService` class can be extended and used with a class that implements the `MessagingServiceProvider` interface to perform message relaying facilities in a very straightforward, easy-to-implement way. The `AbstractGatewayService` takes care of normalizing messages and provides an abstract back channel into the existing messaging system. Complete information on the use of this class is out of the scope of this document - please refer to the Javadoc for more information, or contact Rhombus IM Developer Support directly.

XML Examples

Below are a few examples of how to register RLCs in XML. These are taken from the IM server itself, and from our interoperability package.

User Directory

```
<!-- User Directory -->
<service name="User Directory"
  class="com.rhombus.jabber.agents.standard.UserDirectoryService"
  jid="directory">
  <iq namespace="jabber:iq:browse" to="directory"/>
  <browseable category="service" subtype="jud"/>
</service>
```

Conferencing Module

```
<!-- Conferencing -->
<service name="Conference Directory"
  class="com.rhombus.jabber.agents.standard.ConferenceDirectoryService"
  jid="conferencing">
  <init key="domains">*</init>
  <iq namespace="jabber:iq:browse" to="conferencing"/>
  <browseable category="service" subtype="conference"/>
</service>
```

AIM Transport

```
<service name="AIM Transport"
  class="com.rhombus.enterprise.aim.AimService"
  jid="aol.com">
  <init key="domains">*</init>
  <iq namespace="jabber:iq:register" to="aol.com"/>
  <presence/>
  <message domain="aol.com"/>
  <browseable category="transport" subtype="aim"/>
</service>
```

MSN Transport

```
<service name="MSN Transport"
  class="com.rhombus.enterprise.msn.MsnService"
  jid="msn.com">
  <iq namespace="jabber:iq:register" to="msn.com"/>
  <presence/>
  <message domain="msn.com"/>
  <message domain="hotmail.com"/>
  <browseable category="transport" subtype="msn"/>
</service>
```

The Echo Agent Example

The echo agent is a simple example service that demonstrates many of the key functions of the Rhombus IM API. The echo agent will respond to any incoming message with the same text, slightly modified. It will show you how to receive messages, send messages and modify your presence.

```
/**
 * A simple echo processor that serves as a good example
 * of how an agent might be used. The <code>initialize</code>
 * method updates the echo user agent status to be available,
 * and all messages sent to the agent are echoed back to the
 * sender immediately.<P>
 *
 * @since 3.0
 */
public class EchoAgent
    extends AbstractService
    implements Service
{
    public EchoAgent()
    {
    }

    public void process(AuthenticatedContext ac, ReplySender replySender,
        ConnectedSession session, Element xml)
    {
        String body = XMLHelper.getChildText(xml, "body");
        if (body != null)
        {
            Element copyNode = (Element)xml.cloneNode(false);
            copyNode.setAttribute("to", ac.getLoginJID().toString());
            copyNode.setAttribute("from", xml.getAttribute("to"));
            XMLHelper.appendChildElement(copyNode,
                "body",
                "Hi, you said: " + body);

            replySender.sendMessage(ac.getLoginJID(),
                copyNode);
        }
    }
}

/**
 * This is a convenience implementation of a service that has a single function
 * and allocates no additional resources. This service implementation sets the
 * requested JID presence information to be available as appropriate in the
 * agent lifecycle, and forwards calls to a protected abstract <code>process</code>
 * method only when the service is enabled.<P>
 *
 * Simple services can be derived from this base class to simplify implementation
 * considerably.<P>
 */
public abstract class AbstractService
    implements Service
{
    private static final Logger log = Logger.getLogger(AbstractService.class);

    private boolean alive;
    private User agent;
    private Properties initParams;
```

```
public AbstractService()
{
    alive = false;
}

public void initialize(Context rootContext,
                      Sender sender,
                      JID jid,
                      Properties initParams)
{
    this.initParams = initParams;
    try
    {
        agent = (User)rootContext.getPrincipalHome().find(jid);
    }
    catch (CreateException e)
    {
        log.debug("Service JID not found for " + jid);
    }
}

/**
 * Returns the initialization parameters passed to the <code>initialize</code>
 * call. It is possible to override the call, but this is here as a convenience
 * to obtain the parameters without doing so.
 *
 * @return a Properties object containing the service initialization parameters.<P>
 */
protected Properties getInitParams()
{
    return initParams;
}

/**
 * Disposes of the agent and cleans up all necessary resources.<P>
 */
public void dispose()
{
    if (alive)
        stop();
}

/**
 * Stops the agent.
 */
public void stop()
{
    alive = false;
    if (agent != null)
        agent.updateUserStatus(new
UserStatusImpl(agent.getJID().withResource(agent.getName()), false));
}

/**
 * Starts the agent.
 */
public void start()
{
    if (agent != null)
        agent.updateUserStatus(new
UserStatusImpl(agent.getJID().withResource(agent.getName()), true));
    alive = true;
}
}
```



```
public boolean isStarted()
{
    return alive;
}

/**
 * Processes a message.<P>
 *
 * @param context the user context. This object may be either
 * an AuthenticatedContext or Context object, depending if the
 * user is logged in or not.
 * @param replySender a reply sender, used for responding
 * to the message request or sending other related messages.
 * @param session the session
 * @param xml the xml element
 */
public void onMessage(Context context, ReplySender replySender, ConnectedSession
session, Element xml)
    throws GeneralSecurityException
{
    // ignore messages sent by non-authenticated users.
    if (alive &&
        context instanceof AuthenticatedContext)
    {
        AuthenticatedContext ac = (AuthenticatedContext)context;
        process(ac, replySender, session, xml);
    }
    else
        throw new GeneralSecurityException("unauthentic");
}

public boolean requiresAuthenticatedContext()
{
    return true;
}

/**
 * Processes a message sent by an authenticated user. This method is only
 * called if the agent is started and active.<P>
 *
 * @param context the user context. This context is useful for getting
 * sender identity, login resource, etc.
 * @param replySender a reply sender, used for responding
 * to the message request or sending other related messages.
 * @param session the session
 * @param xml the xml element
 */
protected abstract void process(AuthenticatedContext context,
                                ReplySender replySender,
                                ConnectedSession session,
                                Element xml);
}
```

Conclusions

Hopefully this guide has given you a good start on the capabilities of the Rhombus IM server-side API. The JavaDoc API documentation will give you more information about actually using the API once you have created the framework and are receiving messages.

For development questions and support, please visit our Rhombus IM Developer Forums at <http://www.rhombusim.com/forums/index.php>. If you want a dedicated Rhombus IM developer to help you with your projects, we have developer support packages available for a reasonable fee.

Thanks for your continued support of Rhombus IM!