JBoss Communications JAIN SLEE SIP Wake Up Example User Guide

by Eduardo Martins

1. Document Conventions

This manual uses several conventions to highlight certain words and phrases and draw attention to specific pieces of information.

In PDF and paper editions, this manual uses typefaces drawn from the *Liberation Fonts* [https://fedorahosted.org/liberation-fonts/] set. The Liberation Fonts set is also used in HTML editions if the set is installed on your system. If not, alternative but equivalent typefaces are displayed. Note: Red Hat Enterprise Linux 5 and later includes the Liberation Fonts set by default.

1.1. Typographic Conventions

Four typographic conventions are used to call attention to specific words and phrases. These conventions, and the circumstances they apply to, are as follows.

`Mono-spaced Bold`

Used to highlight system input, including shell commands, file names and paths. Also used to highlight key caps and key-combinations. For example:

> To see the contents of the file `my_next_bestselling_novel` in your current working directory, enter the `cat my_next_bestselling_novel` command at the shell prompt and press **Enter** to execute the command.

The above includes a file name, a shell command and a key cap, all presented in Mono-spaced Bold and all distinguishable thanks to context.

Key-combinations can be distinguished from key caps by the hyphen connecting each part of a key-combination. For example:

> Press **Enter** to execute the command.
>
> Press **Ctrl**+**Alt**+**F1** to switch to the first virtual terminal. Press **Ctrl**+**Alt**+**F7** to return to your X-Windows session.

The first sentence highlights the particular key cap to press. The second highlights two sets of three key caps, each set pressed simultaneously.

If source code is discussed, class names, methods, functions, variable names and returned values mentioned within a paragraph will be presented as above, in `Mono-spaced Bold`. For example:

> File-related classes include `filesystem` for file systems, `file` for files, and `dir` for directories. Each class has its own associated set of permissions.

Proportional Bold

This denotes words or phrases encountered on a system, including application names; dialogue box text; labelled buttons; check-box and radio button labels; menu titles and sub-menu titles. For example:

> Choose **System > Preferences > Mouse** from the main menu bar to launch **Mouse Preferences**. In the **Buttons** tab, click the **Left-handed mouse** check box and click **Close** to switch the primary mouse button from the left to the right (making the mouse suitable for use in the left hand).
>
> To insert a special character into a **gedit** file, choose **Applications > Accessories > Character Map** from the main menu bar. Next, choose **Search > Find** from the **Character Map** menu bar, type the name of the character in the **Search** field and click **Next**. The character you sought will be highlighted in the **Character Table**. Double-click this highlighted character to place it in the **Text to copy** field and then click the **Copy** button. Now switch back to your document and choose **Edit > Paste** from the **gedit** menu bar.

The above text includes application names; system-wide menu names and items; application-specific menu names; and buttons and text found within a GUI interface, all presented in Proportional Bold and all distinguishable by context.

Note the **>** shorthand used to indicate traversal through a menu and its sub-menus. This is to avoid the difficult-to-follow 'Select **Mouse** from the **Preferences** sub-menu in the **System** menu of the main menu bar' approach.

`Mono-spaced Bold Italic` or ***Proportional Bold Italic***

Whether Mono-spaced Bold or Proportional Bold, the addition of Italics indicates replaceable or variable text. Italics denotes text you do not input literally or displayed text that changes depending on circumstance. For example:

> To connect to a remote machine using ssh, type `ssh` *`username@domain.name`* at a shell prompt. If the remote machine is `example.com` and your username on that machine is john, type `ssh john@example.com`.
>
> The `mount -o remount` *`file-system`* command remounts the named file system. For example, to remount the `/home` file system, the command is `mount -o remount /home`.
>
> To see the version of a currently installed package, use the `rpm -q` *`package`* command. It will return a result as follows: *`package-version-release`*.

Note the words in bold italics above username, domain.name, file-system, package, version and release. Each word is a placeholder, either for text you enter when issuing a command or for text displayed by the system.

Aside from standard usage for presenting the title of a work, italics denotes the first use of a new and important term. For example:

> When the Apache HTTP Server accepts requests, it dispatches child processes or threads to handle them. This group of child processes or threads is known as

a *server-pool*. Under Apache HTTP Server 2.0, the responsibility for creating and maintaining these server-pools has been abstracted to a group of modules called *Multi-Processing Modules* (*MPMs*). Unlike other modules, only one module from the MPM group can be loaded by the Apache HTTP Server.

1.2. Pull-quote Conventions

Two, commonly multi-line, data types are set off visually from the surrounding text.

Output sent to a terminal is set in `Mono-spaced Roman` and presented thus:

```
books        Desktop   documentation  drafts  mss    photos   stuff  svn
books_tests  Desktop1  downloads      images  notes  scripts  svgs
```

Source-code listings are also set in `Mono-spaced Roman` but are presented and highlighted as follows:

```
package org.jboss.book.jca.ex1;

import javax.naming.InitialContext;

public class ExClient
{
  public static void main(String args[])
      throws Exception
  {
    InitialContext iniCtx = new InitialContext();
    Object         ref    = iniCtx.lookup("EchoBean");
    EchoHome       home   = (EchoHome) ref;
    Echo           echo   = home.create();

    System.out.println("Created Echo");

    System.out.println("Echo.echo('Hello') = " + echo.echo("Hello"));
  }

}
```

1.3. Notes and Warnings

Finally, we use three visual styles to draw attention to information that might otherwise be overlooked.

Note

A note is a tip or shortcut or alternative approach to the task at hand. Ignoring a note should have no negative consequences, but you might miss out on a trick that makes your life easier.

Important

Important boxes detail things that are easily missed: configuration changes that only apply to the current session, or services that need restarting before an update will apply. Ignoring Important boxes won't cause data loss but may cause irritation and frustration.

Warning

A Warning should not be ignored. Ignoring warnings will most likely cause data loss.

2. Provide feedback to the authors!

If you find a typographical error in this manual, or if you have thought of a way to make this manual better, we would love to hear from you! Please submit a report in the the *Issue Tracker* [http://bugzilla.redhat.com/bugzilla/], against the product **JBoss Communications JAIN SLEE SIP Wake Up Example**, or contact the authors.

When submitting a bug report, be sure to mention the manual's identifier: JAIN_SLEE_SipWakeUp_EXAMPLE_User_Guide

If you have a suggestion for improving the documentation, try to be as specific as possible when describing it. If you have found an error, please include the section number and some of the surrounding text so we can find it easily.

Introduction to JBoss Communications JAIN SLEE SIP Wake Up Example

This example is a JAIN SLEE application which processes SIP messages from registered SIP UAs to act as an Wake Up system.

It is a simple example but provides usage of JAIN SLEE child relations, timers, null activities, sbb activity context interfaces and JAIN SIP RA code, thus being very useful for beginners.

The sender uses a specific MESSAGE format, containing a timeout T and wake up message M values, and then, when T seconds pass, the service will send a message containing M back to the UA.

Setup

2.1. Pre-Install Requirements and Prerequisites

Ensure that the following requirements have been met before continuing with the install.

2.1.1. Hardware Requirements

The Example doesn't change the JBoss Communications JAIN SLEE Hardware Requirements, refer to JBoss Communications JAIN SLEE documentation for more information.

2.1.2. Software Prerequisites

The Example requires JBoss Communications JAIN SLEE properly set, with SIP11 Resource Adaptor and SIP Services Example deployed.

2.2. JBoss Communications JAIN SLEE SIP Wake Up Example Source Code

This section provides instructions on how to obtain and build the SIP Wake Up Example from source code.

2.2.1. Release Source Code Building

1. **Downloading the source code**

Important

Subversion is used to manage its source code. Instructions for using Subversion, including install, can be found at *http://svnbook.red-bean.com*

Use SVN to checkout a specific release source, the base URL is ?, then add the specific release version, lets consider 2.1.2.FINAL.

```
[usr]$ svn co ?/2.1.2.FINAL slee-example-sip-wake-up-2.1.2.FINAL
```

2. **Building the source code**

Important

Maven 2.0.9 (or higher) is used to build the release. Instructions for using Maven2, including install, can be found at *http://maven.apache.org*

Use Maven to build the deployable unit binary.

```
[usr]$ cd slee-example-sip-wake-up-2.1.2.FINAL
[usr]$ mvn install
```

Once the process finishes you should have the `deployable-unit` jar file in the `target` directory, if JBoss Communications JAIN SLEE is installed and environment variable JBOSS_HOME is pointing to its underlying JBoss Enterprise Application Platform directory, then the deployable unit jar will also be deployed in the container.

Important

This procedure does not install the Example's dependencies

2.2.2. Development Trunk Source Building

Similar process as for *Section 2.2.1, "Release Source Code Building"*, the only change is the SVN source code URL, which is NOT AVAILABLE.

2.3. Installing JBoss Communications JAIN SLEE SIP Wake Up Example

To install the Example simply execute provided ant script `build.xml` default target:

```
[usr]$ ant
```

The script will copy the Example's deployable unit jar to the `default` JBoss Communications JAIN SLEE server profile deploy directory, to deploy to another server profile use the argument `-Dnode=`.

Important

This procedure also installs the Example's dependencies.

2.4. Uninstalling JBoss Communications JAIN SLEE SIP Wake Up Example

To uninstall the Example simply execute provided ant script `build.xml` `undeploy` target:

```
[usr]$ ant undeploy-all
```

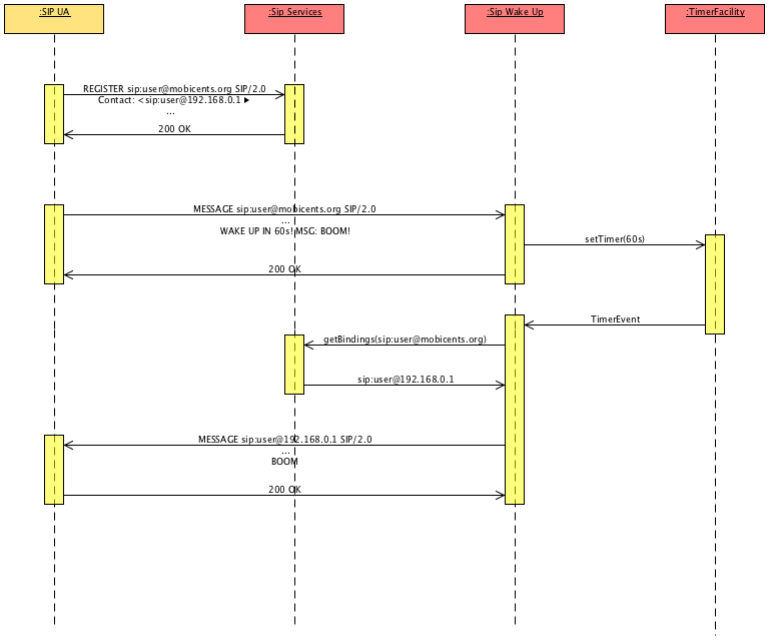
The script will delete the Example's deployable unit jar from the `default` JBoss Communications JAIN SLEE server profile deploy directory, to undeploy from another server profile use the argument `-Dnode=`.

Important

This procedure also uninstalls the Example's dependencies.

Design Overview

The SIP Wake Up Example is JAIN SLEE 1.1 Application which handles SIP MESSAGE requests containing a specific content format, to trigger a wake up message. The wake up message and target user are extracted from the SIP MESSAGE request, as the duration of the timer to set. Once the timer expires the application will get all SIP entities registered as the target user, from SIP Services application, and a SIP MESSAGE request is sent for each of those entities. The diagram below depicts this behavior.



SIP Wake Up Example Functionality

The MESSAGE format to be used is:

```
[any text string]WAKE UP IN [timer value in seconds]s! MSG: [msg to send back to UA]![any text string].
```

The parts out of are case sensitive tokens used by the service to parse the message. Also note that the spaces between tokens and values are required. A message that does not complaints with this format will produce error behavior in the service.

Source Code Overview

The example application is defined by a service descriptor, which refers the included root SBB. The root SBB uses the Location Service SBB (from SIP Services Example) as a child, to retrieve the SIP entities registered.

Important

To obtain the example's complete source code please refer to *Section 2.2, "JBoss Communications JAIN SLEE SIP Wake Up Example Source Code"*.

4.1. Service Descriptor

The service descriptor is plain simple, it just defines the service ID, the ID of the root SBB and its default priority. The complete XML is:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE service-xml PUBLIC
   "-//Sun Microsystems, Inc.//DTD JAIN SLEE Service 1.1//EN"
   "http://java.sun.com/dtd/slee-service-xml_1_1.dtd">
<service-xml>
   <service>
      <service-name>Wake Up Service</service-name>
      <service-vendor>NIST</service-vendor>
      <service-version>1.0</service-version>
      <root-sbb>
         <sbb-name>Wake Up Sbb</sbb-name>
         <sbb-vendor>NIST</sbb-vendor>
         <sbb-version>1.0</sbb-version>
      </root-sbb>
      <default-priority>0</default-priority>
   </service>
</service-xml>
```

4.2. The Root SBB

The SIP Wake Up Example's Root SBB is composed by the abstract class and the XML descriptor.

4.2.1. The Root SBB Abstract Class

The class `org.mobicents.slee.examples.wakeup.WakeupSbb` includes all the service logic for the example.

4.2.1.1. The setSbbContext(SbbContext) method

The `javax.slee.SbbObject`'s `setSbbContext(SbbContext)` is used by SBBs to store the SBB's context into a class field. The SBB should take the opportunity to also store objects, such as SLEE facilities, which are reused by all service logic entities, a.k.a. SbbEntities, and are stored in the JNDI environment.

The class fields and `setSbbContext(SbbContext)` method's and related code:

```
// the Sbb's context
private SbbContext sbbContext;

// the Sbb's single tracer
private Tracer tracer = null;

// cached objects in Sbb's environment, lookups are expensive
private SleeSipProvider sipProvider;
private TimerFacility timerFacility;
private NullActivityContextInterfaceFactory nullACIFactory;
private NullActivityFactory nullActivityFactory;

/*
 * (non-Javadoc)
 *
 * @see javax.slee.Sbb#setSbbContext(javax.slee.SbbContext)
 */
public void setSbbContext(SbbContext context) {
    // save the sbb context in a field
    this.sbbContext = context;
    // get the tracer if needed
    this.tracer = context.getTracer(WakeUpSbb.class.getSimpleName());
    // get jndi environment stuff
    try {
        final Context myEnv = (Context) new InitialContext();
        // slee facilities
        this.timerFacility = (TimerFacility) myEnv
                .lookup(TimerFacility.JNDI_NAME);
        this.nullACIFactory = (NullActivityContextInterfaceFactory) myEnv
                .lookup(NullActivityContextInterfaceFactory.JNDI_NAME);
```

```
            this.nullActivityFactory = (NullActivityFactory) myEnv
                    .lookup(NullActivityFactory.JNDI_NAME);
            // the sbb interface to interact with SIP resource adaptor
            this.sipProvider = (SleeSipProvider) myEnv
                    .lookup("java:comp/env/slee/resources/jainsip/1.2/provider");
        } catch (Exception e) {
            tracer.severe("Failed to set sbb context", e);
        }
    }
```

4.2.1.2. CMP Fields Accessors

For each CMP field, which will hold the service logic instance data, the application defines two abstract methods, the getter and the setter. SLEE is responsible for the implementation of those methods.

The CMP field's accessors code:

```
    public abstract void setSender(Address sender);
    public abstract Address getSender();

    public abstract void setCallId(CallIdHeader callId);
    public abstract CallIdHeader getCallId();

    public abstract void setBody(String body);
    public abstract String getBody();
```

4.2.1.3. The SIP MESSAGE event handler

The SIP MESSAGE is the starting point of each instance of the service logic, its responsibility is:

- Extract the relevant message information and store in CMP fields, the correct place holders for service logic instance data event handler is the entry point.
- Set the timer with the duration extract from the SIP MESSAGE request on a new Null Activity, needed to keep the service logic instance alive. Recall that SLEE garbage collects all SBBEntities which are not attached to a single ActivityContext, and at this point the entity is only attached to the SIP ServerTransaction activity, that is goin to end once the application returns a final response.

- Reply the successful processing of the SIP request.

The event handler code:

```
/**
 * Event handler for the SIP MESSAGE from the UA
 *
 * @param event
 * @param aci
 */
public void onMessageEvent(javax.sip.RequestEvent event,
        ActivityContextInterface aci) {

    final Request request = event.getRequest();
    try {
        // message body should be *FIRST_TOKEN<timer value in
        // seconds>MIDDLE_TOKEN<msg to send back to UA>LAST_TOKEN*
        final String body = new String(request.getRawContent());
        final int firstTokenStart = body.indexOf(FIRST_TOKEN);
        final int timerDurationStart = firstTokenStart + FIRST_TOKEN_LENGTH;
        final int middleTokenStart = body.indexOf(MIDDLE_TOKEN,
                timerDurationStart);
        final int bodyMessageStart = middleTokenStart + MIDDLE_TOKEN_LENGTH;
        final int lastTokenStart = body.indexOf(LAST_TOKEN,
                bodyMessageStart);
        if (firstTokenStart > -1 && middleTokenStart > -1
                && lastTokenStart > -1) {
            // extract the timer duration
            final int timerDuration = Integer.parseInt(body.substring(
                    timerDurationStart, middleTokenStart));
            // create a null AC and attach the sbb local object
            final ActivityContextInterface timerACI = this.nullACIFactory
                    .getActivityContextInterface(this.nullActivityFactory
                            .createNullActivity());
            timerACI.attach(sbbContext.getSbbLocalObject());
            // set the timer on the null AC, because the one from this event
            // will end as soon as we send back the 200 ok
            this.timerFacility.setTimer(timerACI, null, System
                    .currentTimeMillis()
                    + (timerDuration * 1000), new TimerOptions());
            // extract the body message
            final String bodyMessage = body.substring(bodyMessageStart,
                    lastTokenStart);
```

```
            // store it in a cmp field
            setBody(bodyMessage);
            // do the same for the call id
            setCallId((CallIdHeader) request.getHeader(CallIdHeader.NAME));
            // also store the sender's address, so we can send the wake up
            // message
            final FromHeader fromHeader = (FromHeader) request
                    .getHeader(FromHeader.NAME);
            if (tracer.isInfoEnabled()) {
                tracer.info("Received a valid message from "
                        + fromHeader.getAddress()
                        + " requesting a reply containing '" + bodyMessage
                        + "' after " + timerDuration + "s");
            }
            setSender(fromHeader.getAddress());
            // finally reply to the SIP message request
            sendResponse(event, Response.OK);
        } else {
            // parsing failed
            tracer.warning("Invalid msg '" + body + "' received");
            sendResponse(event, Response.BAD_REQUEST);
        }
    } catch (Throwable e) {
        // oh oh something wrong happened
        tracer.severe("Exception while processing MESSAGE", e);
        try {
            sendResponse(event, Response.SERVER_INTERNAL_ERROR);
        } catch (Exception f) {
            tracer.severe("Exception while sending SERVER INTERNAL ERROR",
                    f);
        }
    }
}
```

4.2.1.4. Location Service SBB Child Relation

The SBB uses SIP Service's Location Service to retrieve the URIs of all entities registered with the target address, the child relation method is an abstract class that SLEE implements.

The child relation's getter code:

```
/**
 * Child relation to the location service
 * @return
 */
public abstract ChildRelation getLocationChildRelation();
```

4.2.1.5. The TimerEvent handler

The JAIN SLEE TimerEvent handler is invoked when the duration requested by the SIP Message has passed, it is the final "piece" of the service instance logic, and its responsibility is:

- Retrieve all instance data from CMP fields.
- Create a Location Service child SBB and retrieve the target's registered URIs.
- Send the wake up message(s).

The event handler code:

```
/**
 * Event handler from the timer event, which signals that a message must be
 * sent back to the UA
 *
 * @param event
 * @param aci
 */
public void onTimerEvent(TimerEvent event, ActivityContextInterface aci) {
    // detaching so the null AC is claimed after the event handling
    aci.detach(sbbContext.getSbbLocalObject());
    // get data from cmp fields
    String body = getBody();
    CallIdHeader callId = getCallId();
    Address sender = getSender();
    try {
        // create headers needed to create a out-of-dialog request
        AddressFactory addressFactory = sipProvider.getAddressFactory();
        Address fromNameAddress = addressFactory
                .createAddress("sip:wakeup@mobicents.org");
        fromNameAddress.setDisplayName("Wake Up Service");
        HeaderFactory headerFactory = sipProvider.getHeaderFactory();
        FromHeader fromHeader = headerFactory.createFromHeader(
                fromNameAddress, null);
```

```
        List<ViaHeader> viaHeaders = new ArrayList<ViaHeader>(1);
        ListeningPoint listeningPoint = sipProvider.getListeningPoints()[0];
        ViaHeader viaHeader = sipProvider.getHeaderFactory()
                .createViaHeader(listeningPoint.getIPAddress(),
                        listeningPoint.getPort(),
                        listeningPoint.getTransport(), null);
        viaHeaders.add(viaHeader);
        ContentTypeHeader contentTypeHeader = headerFactory
                .createContentTypeHeader("text", "plain");
        CSeqHeader cSeqHeader = headerFactory.createCSeqHeader(2L,
                Request.MESSAGE);
        MaxForwardsHeader maxForwardsHeader = headerFactory
                .createMaxForwardsHeader(70);
        // create location service child sbb
        final LocationSbbLocalObject locationChildSbb = (LocationSbbLocalObject)
           getLocationChildRelation().create();
        // get sender bindings and send a message to each
        MessageFactory messageFactory = sipProvider.getMessageFactory();
        for (RegistrationBinding registration : locationChildSbb
                .getBindings(sender.getURI().toString()).values()) {
           try {
              // create request uri
              URI requestURI = addressFactory.createURI(registration
                      .getContactAddress());
              // create to header
              ToHeader toHeader = headerFactory.createToHeader(sender,
                      null);
              // create request
              Request request = messageFactory.createRequest(requestURI,
                      Request.MESSAGE, callId, cSeqHeader, fromHeader,
                      toHeader, viaHeaders, maxForwardsHeader,
                      contentTypeHeader, body);
              // create client transaction and send request
              ClientTransaction clientTransaction = sipProvider
                      .getNewClientTransaction(request);
              clientTransaction.sendRequest();
           } catch (Throwable f) {
              tracer.severe("Failed to create and send message", f);
           }
        }
     } catch (Throwable e) {
        tracer.severe("Failed to create message headers", e);
     }
  }
```

4.2.2. Root SBB XML Descriptor

The Root SBB XML Descriptor has to be provided and match the abstract class code.

First relevant part is the declaration of the `sbb-classes` element, where the sbb class abstract name must be specified, along with the cmp fields and child relation.:

```
    <sbb-classes>
      <sbb-abstract-class>
        <sbb-abstract-class-name>org.mobicents.slee.examples.wakeup.WakeUpSbb</sbb-
abstract-class-name>
        <cmp-field>
          <cmp-field-name>body</cmp-field-name>
        </cmp-field>
        <cmp-field>
          <cmp-field-name>callId</cmp-field-name>
        </cmp-field>
        <cmp-field>
          <cmp-field-name>sender</cmp-field-name>
        </cmp-field>
        <get-child-relation-method>
          <sbb-alias-ref>LocationSbb</sbb-alias-ref>
          <get-child-relation-method-name>
            getLocationChildRelation
          </get-child-relation-method-name>
          <default-priority>0</default-priority>
        </get-child-relation-method>
      </sbb-abstract-class>
    </sbb-classes>
```

Then the events handled by the SBB must be specified too:

```
    <event event-direction="Receive" initial-event="True">
      <event-name>MessageEvent</event-name>
      <event-type-ref>
```

```
            <event-type-name>javax.sip.message.Request.MESSAGE</event-type-name>
            <event-type-vendor>net.java.slee</event-type-vendor>
            <event-type-version>1.2</event-type-version>
        </event-type-ref>
        <initial-event-select variable="ActivityContext" />
    </event>

    <event event-direction="Receive" initial-event="False">
        <event-name>TimerEvent</event-name>
        <event-type-ref>
            <event-type-name>javax.slee.facilities.TimerEvent</event-type-name>
            <event-type-vendor>javax.slee</event-type-vendor>
            <event-type-version>1.0</event-type-version>
        </event-type-ref>
    </event>
```

Finally, the SIP11 Resource Adaptor must be specified also, otherwise SLEE won't put its SBB Interface in the SBB's JNDI Context:

```
    <resource-adaptor-type-binding>
        <resource-adaptor-type-ref>
            <resource-adaptor-type-name>
                JAIN SIP
            </resource-adaptor-type-name>
            <resource-adaptor-type-vendor>
                javax.sip
            </resource-adaptor-type-vendor>
            <resource-adaptor-type-version>
                1.2
            </resource-adaptor-type-version>
        </resource-adaptor-type-ref>
        <activity-context-interface-factory-name>
            slee/resources/jainsip/1.2/acifactory
        </activity-context-interface-factory-name>
        <resource-adaptor-entity-binding>
            <resource-adaptor-object-name>
                slee/resources/jainsip/1.2/provider
            </resource-adaptor-object-name>
            <resource-adaptor-entity-link>
                SipRA
```

```
            </resource-adaptor-entity-link>
        </resource-adaptor-entity-binding>
    </resource-adaptor-type-binding>
```

Running the Example

To easiest way to try the example application is to start the JAIN SLEE container, then use SIPP scripts, `run.sh` or `run.bat` depending on which Operating System being used, in `sipp` directory. The scripts will send the SIP MESSAGE request and handle the remaining SIP messages exchanged, all the traffic should be printed in the application server console. The usage of SIPP scripts requires SIPP to be in `$PATH` environment variable.

To use a real SIP UA client, such as X-Lite, configure it with `127.0.0.1` as the domain, and then send a MESSAGE with correct format to `sip:wakeup@mobicents.org`. Note that unless `sip:wakeup@mobicents.org`is added to the contact list, some SIP UA clients may ignore the wake up messages.

Traces and Alarms

6.1. Tracers

The example Application uses a single JAIN SLEE 1.1 Tracer, named `WakeUpSbb`. The related log4j category is `javax.slee.SbbNotification[service=ServiceID[name=Wake Up Service,vendor=NIST,version=1.0],sbb=SbbID[name=Wake Up Sbb,vendor=NIST,version=1.0]]`.

6.2. Alarms

The example Application does not sets JAIN SLEE Alarms.

Appendix A. Revision History

Revision History

Revision 1.0 Tue Dec 30 2009 EduardoMartins

Creation of the JBoss Communications JAIN SLEE SIP Wake Up Example User Guide.

Index

F