JBoss Communications JAIN SLEE SIP B2BUA Example User Guide

by Eduardo Martins and Bartosz Baranowski

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 B2BUA Example**, or contact the authors.

When submitting a bug report, be sure to mention the manual's identifier: JAIN_SLEE_SipB2BUA_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 B2BUA Example

This example application is a SIP Back-to-Back User Agent, using the JAIN SIP Resource Adaptor specified in the JAIN SLEE 1.1 specification.

A SIP B2BUA application resides between both end points of a phone call or communications session, and divides the communication session into two call legs, mediating all SIP signaling between both parties, from call establishment to termination. Each session is tracked from beginning to end, allowing the application to provide features such as session accounting and billing, or fail over session routing.

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 deployed.

2.2. JBoss Communications JAIN SLEE SIP B2BUA Example Source Code

This section provides instructions on how to obtain and build the SIP B2BUA 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 2.1.0.GA.

```
[usr]$ svn co ?/2.1.0.GA slee-example-sip-b2bua-2.1.0.GA
```

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-b2bua-2.1.0.GA
[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 B2BUA 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 B2BUA Example

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

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

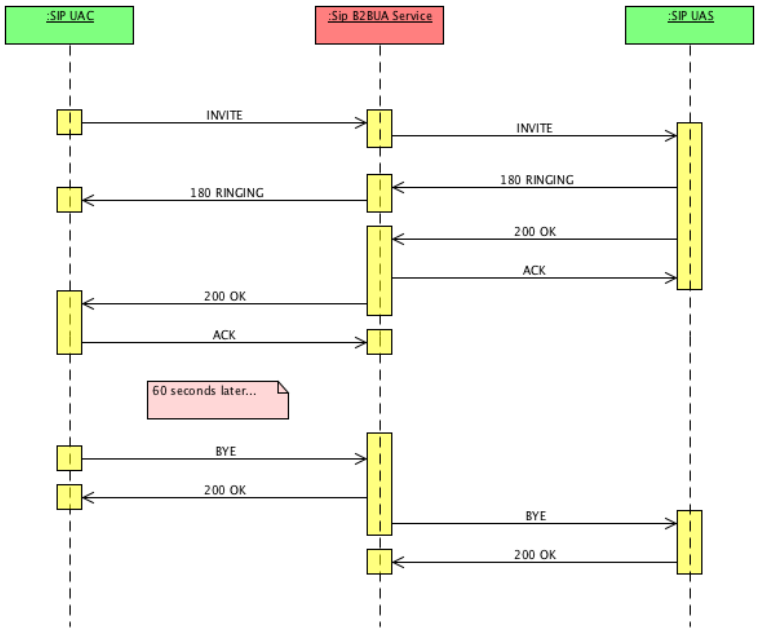
The script will delete the Example's deployable unit jar from the `default` JBoss Communications JAIN SLEE server profile deploy directory, to undeploy from another server profile use the argument `-Dnode=`.

Important

This procedure also uninstalls the Example's dependencies.

Design Overview

The SIP B2BUA Example is JAIN SLEE 1.1 Application which handles SIP sessions between a SIP UAC and UAS, through two different "legs", one per session party. In the originating session leg the application acts as UAS and processes the request as a UAC to the destination end, handling the signaling between end points back-to-back.Each side of a B2BUA operates as a standard SIP network element as specified in RFC 3261, the official SIP specification. The diagram below depicts the application behavior for all SIP messages exchanged by the session parties.



SIP B2BUA Example Functionality

Source Code Overview

The example application is defined by a service descriptor, which refers the Root SBB. The Root SBB does not defines child relations, which means the application is a single SBB.

Important

To obtain the example's complete source code please refer to *Section 2.2, "JBoss Communications JAIN SLEE SIP B2BUA 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>SimpleSip11B2BTestService</service-name>
      <service-vendor>org.mobicents</service-vendor>
      <service-version>1.0</service-version>
      <root-sbb>
         <sbb-name>SimpleSip11B2BTestSbb</sbb-name>
         <sbb-vendor>org.mobicents</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 B2BUA 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.example.sip11.b2b.SimpleSip11B2BTestSbb` 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:

```
public void setSbbContext(SbbContext context) {
    this.sbbContext = context;
    if (tracer == null) {
        tracer = sbbContext.getTracer(SimpleSip11B2BTestSbb.class
                .getSimpleName());
    }
    try {
        Context ctx = (Context) new InitialContext()
                .lookup("java:comp/env");
        sipActivityContextInterfaceFactory = (SipActivityContextInterfaceFactory) ctx
                .lookup("slee/resources/jainsip/1.2/acifactory");
        sipProvider = (SleeSipProvider) ctx
                .lookup("slee/resources/jainsip/1.2/provider");
    } catch (NamingException e) {
        tracer.severe(e.getMessage(), 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 application uses two CMP field, where it stores each session leg SIP Dialog, the accessors code is:

```
public abstract void setIncomingDialog(ActivityContextInterface aci);
public abstract ActivityContextInterface getIncomingDialog();

public abstract void setOutgoingDialog(ActivityContextInterface aci);
public abstract ActivityContextInterface getOutgoingDialog();
```

4.2.1.3. The SIP INVITE Event Handler Method

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

- Create the incoming and outgoing leg SIP Dialog activities and attach the SbbEntity, to receive further SIP messages on each leg.
- Store each SIP Dialog related ActivityContextInterface in a CMP field.
- Forward the INVITE SIP request to the outgoing session leg.

The event handler code:

```
    // Initial request
public void onInviteEvent(RequestEvent event, ActivityContextInterface aci) {
    // ACI is the server transaction activity
    final ServerTransaction st = event.getServerTransaction();
    try {
        // Create the dialogs representing the incoming and outgoing call
        // legs.
        final DialogActivity incomingDialog = (DialogActivity) sipProvider
                .getNewDialog(st);
        final DialogActivity outgoingDialog = sipProvider.getNewDialog(
                incomingDialog, true);
        // Obtain the dialog activity contexts and attach to them
        final ActivityContextInterface outgoingDialogACI = sipActivityContextInterfaceFactory
                .getActivityContextInterface(outgoingDialog);
        final ActivityContextInterface incomingDialogACI = sipActivityContextInterfaceFactory
                .getActivityContextInterface(incomingDialog);
        final SbbLocalObject sbbLocalObject = sbbContext
                .getSbbLocalObject();
        incomingDialogACI.attach(sbbLocalObject);
        outgoingDialogACI.attach(sbbLocalObject);
        // Record which dialog is which, so we can find the peer dialog
```

```
            // when forwarding messages between dialogs.
            setIncomingDialog(incomingDialogACI);
            setOutgoingDialog(outgoingDialogACI);
            forwardRequest(st, outgoingDialog);
        } catch (Throwable e) {
            tracer.severe("Failed to process incoming INVITE.", e);
            replyToRequestEvent(event, Response.SERVICE_UNAVAILABLE);
        }
    }
```

4.2.1.4. The 1xx Response Event Handler Method

Non final responses can be received from the outgoing session leg, and all besides 100 TRYING are processed and forwarded to the incoming leg.

The event handler method's code:

```
    public void on1xxResponse(ResponseEvent event, ActivityContextInterface aci) {
        if (event.getResponse().getStatusCode() == Response.TRYING) {
            // those are not forwarded to the other dialog
            return;
        }
        processResponse(event, aci);
    }
```

4.2.1.5. The 2xx Response Event Handler Method

Successful final responses can be received on both session legs: as a response to the INVITE sent to the outgoing leg, or as a response to a BYE sent to any of the session legs.

If it is a response to the INVITE the application sends an ACK immediately, to minimize the response retransmissions, and then forwards the message to the incoming leg.

In case of a BYE response, the application does not forwards it to the other leg, because the other leg's BYE was already replied.

The event handler method's code:

```
public void on2xxResponse(ResponseEvent event, ActivityContextInterface aci) {
    final CSeqHeader cseq = (CSeqHeader) event.getResponse().getHeader(
            CSeqHeader.NAME);
    if (cseq.getMethod().equals(Request.INVITE)) {
        // lets ack it ourselves to avoid UAS retransmissions due to
        // forwarding of this response and further UAC Ack
        // note that the app does not handles UAC ACKs
        try {
            final Request ack = event.getDialog().createAck(
                    cseq.getSeqNumber());
            event.getDialog().sendAck(ack);
        } catch (Exception e) {
            tracer.severe("Unable to ack INVITE's 200 ok from UAS", e);
        }
    } else if (cseq.getMethod().equals(Request.BYE)
            || cseq.getMethod().equals(Request.CANCEL)) {
        // not forwarded to the other dialog
        return;
    }
    processResponse(event, aci);
}
```

4.2.1.6. The BYE Request Event Handler Method

The BYE request is received when a party wants to end the session, in such scenario the application immediately replies with 200 OK, minimizing the request retransmissions, and then forwards the BYE to the other session leg.

The event handler method's code:

```
public void onBye(RequestEvent event, ActivityContextInterface aci) {
    // send back 200 ok for this dialog right away, to avoid retransmissions
    replyToRequestEvent(event, Response.OK);
    // forward to the other dialog
    processMidDialogRequest(event, aci);
}
```

4.2.1.7. The CANCEL Request Event Handler Method

The CANCEL request is received when a party wants to cancel the session, in such scenario the application first needs to check the state of the other leg, and if it is already established, a BYE request is sent instead of forwarding the CANCEL message,.

The event handler method's code:

```
public void onCancel(CancelRequestEvent event, ActivityContextInterface aci) {
    if (tracer.isInfoEnabled()) {
        tracer.info("Got a CANCEL request.");
    }

    try {
        this.sipProvider.acceptCancel(event, false);
        final ActivityContextInterface peerDialogACI = getPeerDialog(aci);
        final DialogActivity peerDialog = (DialogActivity) peerDialogACI
                .getActivity();
        final DialogState peerDialogState = peerDialog.getState();
        if (peerDialogState == null || peerDialogState == DialogState.EARLY) {
            peerDialog.sendCancel();
        } else {
            peerDialog.sendRequest(peerDialog.createRequest(Request.BYE));
        }
    } catch (Exception e) {
        tracer.severe("Failed to process cancel request", e);
    }
}
```

4.2.1.8. Helper Methods

The application defines a few helper methods to deal with message forwarding, providing example and optimal usage of the JAIN SIP RA APIs:

```
private void replyToRequestEvent(RequestEvent event, int status) {
    try {
        event.getServerTransaction().sendResponse(
                sipProvider.getMessageFactory().createResponse(status,
                        event.getRequest()));
```

```
    } catch (Throwable e) {
        tracer.severe("Failed to reply to request event:\n" + event, e);
    }
}

private void processMidDialogRequest(RequestEvent event,
        ActivityContextInterface dialogACI) {
    try {
        // Find the dialog to forward the request on
        ActivityContextInterface peerACI = getPeerDialog(dialogACI);
        forwardRequest(event.getServerTransaction(),
                (DialogActivity) peerACI.getActivity());
    } catch (SipException e) {
        tracer.severe(e.getMessage(), e);
        replyToRequestEvent(event, Response.SERVICE_UNAVAILABLE);
    }
}

private void processResponse(ResponseEvent event,
        ActivityContextInterface aci) {
    try {
        // Find the dialog to forward the response on
        ActivityContextInterface peerACI = getPeerDialog(aci);
        forwardResponse((DialogActivity) aci.getActivity(),
                (DialogActivity) peerACI.getActivity(), event
                        .getClientTransaction(), event.getResponse());
    } catch (SipException e) {
        tracer.severe(e.getMessage(), e);
    }
}

private ActivityContextInterface getPeerDialog(ActivityContextInterface aci)
        throws SipException {
    final ActivityContextInterface incomingDialogAci = getIncomingDialog();
    if (aci.equals(incomingDialogAci)) {
        return getOutgoingDialog();
    }
    if (aci.equals(getOutgoingDialog())) {
        return incomingDialogAci;
    }
    throw new SipException("could not find peer dialog");

}
```

```
private void forwardRequest(ServerTransaction st, DialogActivity out)
        throws SipException {
    final Request incomingRequest = st.getRequest();
    if (tracer.isInfoEnabled()) {
        tracer.info("Forwarding request " + incomingRequest.getMethod()
                + " to dialog " + out);
    }
    // Copies the request, setting the appropriate headers for the dialog.
    Request outgoingRequest = out.createRequest(incomingRequest);
    // Send the request on the dialog activity
    final ClientTransaction ct = out.sendRequest(outgoingRequest);
    // Record an association with the original server transaction,
    // so we can retrieve it when forwarding the response.
    out.associateServerTransaction(ct, st);
}

private void forwardResponse(DialogActivity in, DialogActivity out,
        ClientTransaction ct, Response receivedResponse)
        throws SipException {
    // Find the original server transaction that this response
    // should be forwarded on.
    final ServerTransaction st = in.getAssociatedServerTransaction(ct);
    // could be null
    if (st == null)
        throw new SipException(
                "could not find associated server transaction");
    if (tracer.isInfoEnabled()) {
        tracer.info("Forwarding response "
                + receivedResponse.getStatusCode() + " to dialog " + out);
    }
    // Copy the response across, setting the appropriate headers for the
    // dialog
    final Response outgoingResponse = out.createResponse(st,
            receivedResponse);
    // Forward response upstream.
    try {
        st.sendResponse(outgoingResponse);
    } catch (InvalidArgumentException e) {
        tracer.severe("Failed to send response:\n" + outgoingResponse, e);
        throw new SipException("invalid response", 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:

```
    <sbb-classes>
      <sbb-abstract-class>

   <sbb-abstract-class-name>org.mobicents.slee.example.sip11.b2b.SimpleSip11B2BTestSbb</
sbb-abstract-class-name>
          <cmp-field>
            <cmp-field-name>incomingDialog</cmp-field-name>
          </cmp-field>
          <cmp-field>
            <cmp-field-name>outgoingDialog</cmp-field-name>
          </cmp-field>
      </sbb-abstract-class>
    </sbb-classes>
```

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

```
    <!-- INITIALS EVENT, OUT OF DIALOG -->
    <event event-direction="Receive" initial-event="True">
      <event-name>InviteEvent</event-name>
      <event-type-ref>
        <event-type-name>javax.sip.message.Request.INVITE</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>

    <!--  EVERYTHING ELSE HAPPENS IN DIALOG -->
    <event event-direction="Receive" initial-event="False">
```

```
        <event-name>1xxResponse</event-name>
        <event-type-ref>
            <event-type-name>javax.sip.message.Response.PROVISIONAL</event-type-name>
            <event-type-vendor>net.java.slee</event-type-vendor>
            <event-type-version>1.2</event-type-version>
        </event-type-ref>
    </event>
    <event event-direction="Receive" initial-event="False">
        <event-name>2xxResponse</event-name>
        <event-type-ref>
            <event-type-name>javax.sip.message.Response.SUCCESS</event-type-name>
            <event-type-vendor>net.java.slee</event-type-vendor>
            <event-type-version>1.2</event-type-version>
        </event-type-ref>
    </event>

    <event event-direction="Receive" initial-event="False">
        <event-name>Bye</event-name>
        <event-type-ref>
            <event-type-name>javax.sip.Dialog.BYE</event-type-name>
            <event-type-vendor>net.java.slee</event-type-vendor>
            <event-type-version>1.2</event-type-version>
        </event-type-ref>
    </event>
    <event event-direction="Receive" initial-event="False">
        <event-name>Cancel</event-name>
        <event-type-ref>
            <event-type-name>javax.sip.message.Request.CANCEL</event-type-name>
            <event-type-vendor>net.java.slee</event-type-vendor>
            <event-type-version>1.2</event-type-version>
        </event-type-ref>
    </event>
```

Note that there is a single event defined as initial, which triggers the sbb logic, remaining events all happen in activities that the service instance is already attached, abstracting the application from calculating which session it handles.

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 provide two different session flows provided, successful call handling (the `run_*_DIALOG.*` scripts) and call cancellation (the `run_*_CANCEL.*` scripts). For each scenario first start the `uas` script, then the `uac` script, which triggers the calls. All the traffic should be printed in the application server console. The usage of SIPP scripts requires SIPP to be in `$PATH` environment variable.

Traces and Alarms

6.1. Tracers

Important

Spaces where introduced in log4j category name to properly render page. Please remove them when using copy/paste.

The example Application uses a single JAIN SLEE 1.1 Tracer, named `SimpleSip11B2BTestSbb`. The related log4j category is `javax.slee.SbbNotification[service=ServiceID[name=SimpleSip11B2BTestService,vendor=org.mobicents,version=1.0], sbb=SbbID[name=SimpleSip11B2BTestSbb,vendor=org.mobicents,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 B2BUA Example User Guide.

Index

F