

JBoss Communications JAIN SLEE SIP JDBC Registrar Example User Guide

by Eduardo Martins, Bartosz Baranowski, and Alexandre Mendonça

Preface	v
1. Document Conventions	v
1.1. Typographic Conventions	v
1.2. Pull-quote Conventions	vii
1.3. Notes and Warnings	vii
2. Provide feedback to the authors!	viii
1. Introduction to JBoss Communications JAIN SLEE SIP JDBC Registrar Example....	1
2. Setup	3
2.1. Pre-Install Requirements and Prerequisites	3
2.1.1. Hardware Requirements	3
2.1.2. Software Prerequisites	3
2.2. JBoss Communications JAIN SLEE SIP JDBC Registrar Example Source Code....	3
2.2.1. Release Source Code Building	3
2.2.2. Development Trunk Source Building	4
2.3. Installing JBoss Communications JAIN SLEE SIP JDBC Registrar Example	4
2.4. Uninstalling JBoss Communications JAIN SLEE SIP JDBC Registrar Example.....	4
3. Design Overview	7
3.1. Example Components	7
3.2. Registrar and Datasource Operations	7
4. Source Code Overview	9
4.1. Interfaces and contracts	9
4.1.1. Registration Binding	9
4.1.2. Data Source Child	11
4.1.3. Data Source Parent	12
4.2. SIPRegistrarSbb	13
4.2.1. Local interface	13
4.2.2. Child relation	16
4.2.3. REGISTER handler	17
4.2.4. TIMER handler	22
4.3. DataSourceChild SBB	23
4.3.1. Local interface	23
4.3.2. Parent relation	26
4.3.3. JDBC RA Tasks	26
4.3.4. JDBC Task Exception handler	37
4.3.5. JDBC Task Result handler	38
5. Running the Example	39
5.1. Configuration	39
6. Traces and Alarms	41
6.1. Tracers	41
6.2. Alarms	41
A. Revision History	43
Index	45

Preface

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/) [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 Of 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/) [http://bugzilla.redhat.com/bugzilla/], against the product **JBoss Communications JAIN SLEE SIP JDBC Registrar Example**, or contact the authors.

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

JDBC Registrar Example

This example is a JAIN SLEE application which processes SIP Messages from SIP UAs. To be precise it consumes REGISTER requests and acts as registrar.

Registrar routines can be found in [section 10](http://tools.ietf.org/html/rfc3261#section-10) [<http://tools.ietf.org/html/rfc3261#section-10>] of RFC3261.

It is not a trivial example as it provides usage of

- child relations
- timers
- null activities
- SBB activity context interfaces
- activity context interfaces variables
- SBB local interfaces
- JAIN SIP RA code
- JDBC RA code
- SQL and java.sql API
- JMX and SLEE integration

Thus it should be considered as target for more advanced users. For trivial JDBC RA example please refer to `jdbc-demo` source code and documentation.

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 and JDBC Resource Adaptors deployed.

2.2. JBoss Communications JAIN SLEE SIP JDBC Registrar Example Source Code

This section provides instructions on how to obtain and build the SIP JDBC Registrar 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 1.0.0.FINAL.

```
[usr]$ svn co ?/1.0.0.FINAL slee-example-sip-jdbc-registrar-1.0.0.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-jdbc-registrar-1.0.0.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.

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 JDBC Registrar 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=`.

2.4. Uninstalling JBoss Communications JAIN SLEE SIP JDBC Registrar Example

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

```
[usr]$ ant undeploy
```

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=`.

Design Overview

The SIP JDBC Registrar Example is JAIN SLEE 1.1 Application which handles incoming SIP REGISTER requests. Depending on message content, it performs registrar routines and respond to UAC with proper message.

3.1. Example Components

Example consist of following components:

SIP Registrar Sbb

Is entry point of examples service. It consumes REGISTER requests and issues calls to its child SBB in order to perform specific operations on data storage, ie. fetch contacts for particular AOR. It performs partial validation of REGISTER requests.

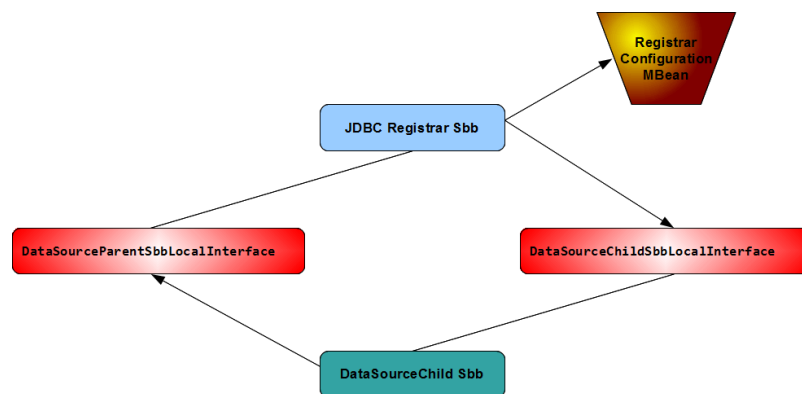
DataSourceChild SBB

Is responsible for performing operations on persistent data store. It exposes operations which allow other SBBs to fetch, remove or update AOR contacts.

RegistrarConfiguration

Is a simple JMX bean which exposes configuration parameters used in registrar routines to check if message has correct content.

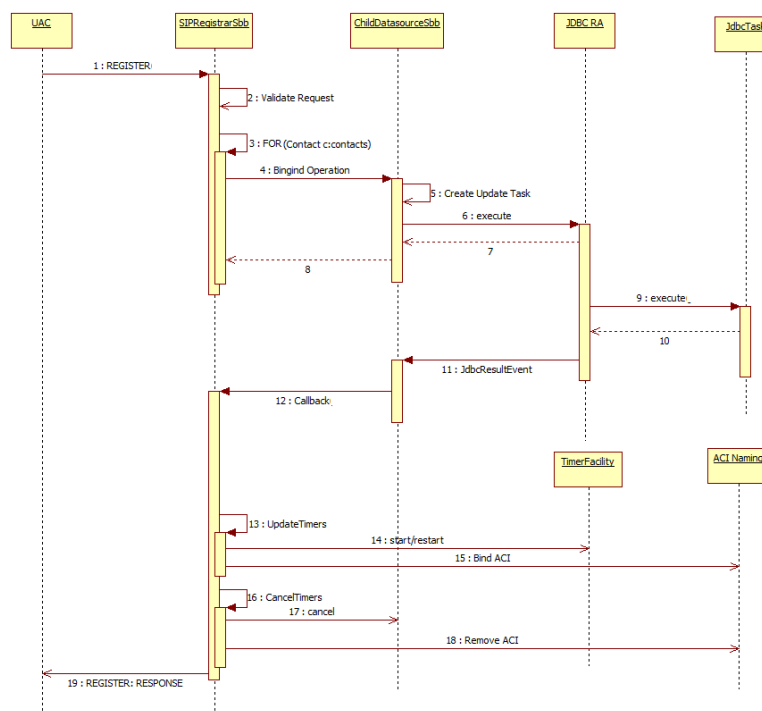
Root SBB of service is `SIP Registrar SBB`, which receives messages as first. Relation between SBB s look as follows:



SIP JDBC Registrar Example components relations

3.2. Registrar and Datasource Operations

To better understand source code, it is required that you are familiar atleast with general design of examples service. Below you can find a diagram which depicts general call flow in example:



SIP JDBC Registrar Example call flow

Source Code Overview



Important

To obtain the example's complete source code please refer to [Section 2.2, “JBoss Communications JAIN SLEE SIP JDBC Registrar Example Source Code”](#).

Chapter [Chapter 3, Design Overview](#) explains top level view of example. This chapter explains how components perform their tasks. For more detailed explanation of JSLEE related source code and xml descriptors, please refer to simpler examples, like `sip-wakeup` or `jdbc-demo`.

4.1. Interfaces and contracts

This section outlines interfaces declared by this example. Those interfaces declare contracts between example components. Thus it is important to understand them well, since such design allows one to introduce his own implementation of examples component.

4.1.1. Registration Binding

The `RegistrationBinding` class is a simple java bean. It is used to pass information about user contact from `DataSourceChild SBB` to `SIPRegistrar SBB`. It has simple set of fields which correspond to contact data like contact address, call id, etc:

```
public class RegistrationBinding {
    private String sipAddress;
    private String contactAddress;
    private long expires;
    private long registrationDate;
    private float qValue;
    private String callId;
    private long cSeq;

    public RegistrationBinding(String sipAddress, String contactAddress, long expires, long registrationDate, float va
        super();
        this.sipAddress = sipAddress;
        this.contactAddress = contactAddress;
        this.expires = expires;
        this.registrationDate = registrationDate;
        this.qValue = value;
        this.callId = callId;
        this.cSeq = seq;
}
```

```
}  
  
....  
  
}
```

Since `RegistrationBinding` follows java bean pattern, it also defines set of simple accessors:

```
public class RegistrationBinding {  
  
    public String getSipAddress() {  
        return sipAddress;  
    }  
  
    public void setSipAddress(String sipAddress) {  
        this.sipAddress = sipAddress;  
    }  
  
    public String getContactAddress() {  
        return contactAddress;  
    }  
  
    public void setContactAddress(String contactAddress) {  
        this.contactAddress = contactAddress;  
    }  
  
    ...  
  
    // --- logic methods  
  
    /**  
     * Returns number of mseconds till this entry expires May be 0 or -ve if  
     * already expired  
     */  
    public long getExpiresDelta() {  
        return ((getExpires() - (System.currentTimeMillis() - getRegistrationDate())) / 1000);  
    }  
  
    ....  
}
```

```
}
```

4.1.2. Data Source Child

The Data Source Child (DataSourceChild SBB) as mentioned is responsible for proper storing user contact data. To allow that, there must be a definition of methods, which will allow to manipulate user data. Such definition can be found in `org.mobicens.slee.example.sjr.data.DataSourceChildSbbLocalInterface`. The said interface is defined as follows:

```
public interface DataSourceChildSbbLocalInterface {  
  
    public void init();  
  
    public void getBindings(String address);  
  
    public void removeBinding(String contact, String address);  
  
    public void removeBindings(String address, String callId, long cSeq);  
  
    public void updateBindings(String address, String callId, long cSeq,  
        List<ContactHeader> contacts);  
  
}
```

Defined methods have following contracts:

`init()`

allows to initialize data storage. It is invoked once, when service starts.

`getBindings(String address)`

request DataSourceChild SBB to fetch contact addresses for passed AOR. Result is passed to parent as SBB Local Interface invocation([Section 4.1.3, "Data Source Parent"](#)).

`removeBinding(String contact, String address)`

request DataSourceChild SBB to remove one contact address for passed AOR. Result is passed to parent as SBB Local Interface invocation([Section 4.1.3, "Data Source Parent"](#)).

`removeBindings(String address, String callId, long cSeq)`

request `DataSourceChild SBB` to remove all contacts addresses for passed AOR. Result is passed to parent as `SBB Local Interface` invocation([Section 4.1.3, "Data Source Parent"](#)).

`updateBindings(String address, String callId, long cSeq, List<ContactHeader> contacts)`

request `DataSourceChild SBB` to update list of available contacts for AOR. Result is passed to parent as `SBB Local Interface` invocation([Section 4.1.3, "Data Source Parent"](#)). Passed list with contacts will become current set of contact for passed AOR(this may result in removal of previously existing contacts if they are not in new set).

4.1.3. Data Source Parent

The `Data Source Parent (SIPRegistrar SBB/DataSourceParent SBB)` consumes UAC requests and triggers `DataSourceChild SBB` to store registration information. To do so it makes use of `DataSourceChildSbbLocalInterface`([Section 4.1.2, "Data Source Child"](#)). The child SBB passes outcome of invocations to `Data Source Parent` by means of another interface - `org.mobicients.slee.example.sjr.data.DataSourceParentSbbLocalInterface`. It is defined as follows:

```
public interface DataSourceParentSbbLocalInterface {

    public void getBindingsResult(int resultCode, List<RegistrationBinding> bindings);

    public void removeBindingsResult(int resultCode, List<RegistrationBinding> currentBindings, List<RegistrationBinding> removedBindings);

    public void updateBindingsResult(int resultCode, List<RegistrationBinding> currentBindings, List<RegistrationBinding> removedBindings);

}
```

Note that each method has `resultCode` argument. Its value indicates outcome of child interface invocation, that is, this value is set as `Status-Code` header in answer

Defined methods have following contracts:

`getBindingsResult(int resultCode, List<RegistrationBinding> bindings)`

invoked as result of call to `DataSourceChildSbbLocalInterface.getBindings`. Arguments contain list of current bindings.

`removeBindingsResult(int resultCode, List<RegistrationBinding> currentBindings, List<RegistrationBinding> removedBindings)`

invoked as result of call to `DataSourceChildSbbLocalInterface.removeBinding` and `DataSourceChildSbbLocalInterface.removeBindings`. Arguments contain list of current bindings and list of removed bindings.

updateBindingsResult(int resultCode, List<RegistrationBinding> currentBindings, List<RegistrationBinding> updatedBindings, List<RegistrationBinding> removedBindings);

invoked as result of call to DataSourceChildSbbLocalInterface.updateBindings.

Arguments contain list of current bindings, updated bindings and removed bindings.

4.2. SIPRegistrarSbb

SIP Registrar SBB is responsible for handling REGISTER requests and sending proper response.

SIP Registrar SBB receives REGISTER requests as initial events.

Class `org.mobicens.slee.example.sjr.sip.SIPRegistrarSbb` includes all the service logic required to perform registry tasks. That is to consume REGISTER and generate proper answer based on outcome of registry operation.

4.2.1. Local interface

SIP Registrar SBB declares SBB Local interface. This interface is used by DataSourceChild SBB to inform SIP Registrar SBB when data source operation returns result. So in short it defines contract between SIP Registrar SBB and DataSourceChild SBB. Local interface is declared in `sbb-jar.xml` in following way:

```
<sbb-jar>
  <sbb>
    <sbb-name>SIP Registrar</sbb-name>
    <sbb-vendor>org.mobicens</sbb-vendor>
    <sbb-version>1.0</sbb-version>

    ...

    <sbb-classes>
      <sbb-abstract-class
        reentrant="True">
        <sbb-abstract-class-name>org.mobicens.slee.example.sjr.sip.SIPRegistrarSbb</sbb-
abstract-class-name>
        <sbb-local-interface>
          <sbb-local-interface-name>
            org.mobicens.slee.example.sjr.data.DataSourceParentSbbLocalObject
          </sbb-local-interface-name>
        </sbb-local-interface>
      </sbb-abstract-class>

      ....

    </sbb-classes>
```

```
...  
  
</sbb>  
</sbb-jar>
```

Parent interface is declared as follows:

```
public interface DataSourceParentSbbLocalObject extends SbbLocalObject,  
    DataSourceParentSbbLocalInterface {  
  
}
```

Above methods([Section 4.1.3, “Data Source Parent”](#)) are invoked by `DataSourceChild SBB` as result of operations invoked on its local interface. According to name, methods perform certain task:

Send query result to UAC

```
@Override  
public void getBindingsResult(int resultCode,  
    List<RegistrationBinding> bindings) {  
    ServerTransaction serverTransaction = getRegisterTransactionToReply();  
    if (serverTransaction == null) {  
        tracer.warning("failed to find SIP server tx to send response");  
        return;  
    }  
    try {  
        if (resultCode < 300) {  
            sendRegisterSuccessResponse(resultCode, bindings,  
                serverTransaction);  
        } else {  
            sendErrorResponse(resultCode, serverTransaction);  
        }  
    } catch (Exception e) {  
        tracer.severe("failed to send SIP response", e);  
    }  
}
```

```
}
```

Send update binding result to UAC

```
@Override
public void updateBindingsResult(int resultCode,
    List<RegistrationBinding> currentBindings,
    List<RegistrationBinding> updatedBindings,
    List<RegistrationBinding> removedBindings) {
    ServerTransaction serverTransaction = getRegisterTransactionToReply();
    if (serverTransaction == null) {
        tracer.warning("failed to find SIP server tx to send response");
        return;
    }
    try {
        if (resultCode < 300) {
            // we have to post process, reset/add/cancel timers...
            // first update, since those are more important. This has to be
            // done, since we want timers updated first
            // so events are not fired and dont corrupt DB entries(remove
            // them,
            // after they are updated)
            updateTimers(updatedBindings);
            cancelTimers(removedBindings);
            sendRegisterSuccessResponse(resultCode, currentBindings,
                serverTransaction);
        } else {
            // we just fail :)
            sendErrorResponse(resultCode, serverTransaction);
        }
    } catch (Exception e) {
        tracer.severe("failed to send SIP response", e);
    }
}
```

Send confirmation on remove of binding

```
private static final List<RegistrationBinding> EMPTY_LIST = Collections.emptyList();
```

```
@Override
public void removeBindingsResult(int resultCode,
    List<RegistrationBinding> currentBindings,
    List<RegistrationBinding> removedBindings) {
    // same processing as update result
    updateBindingsResult(resultCode, currentBindings, EMPTY_LIST,
        removedBindings);
}
```

4.2.2. Child relation

SIP Registrar SBB defines DataSourceChild SBB as child. The DataSourceChild SBB child is used as a driver which manages persistence of user registration data. Child relation is declared in sbb-jar.xml descriptor:

```
<sbb-jar>
  <sbb>
    <sbb-name>SIP Registrar</sbb-name>
    <sbb-vendor>org.mobicens</sbb-vendor>
    <sbb-version>1.0</sbb-version>

    <sbb-ref>
      <sbb-name>DataSourceChild</sbb-name>
      <sbb-vendor>org.mobicens</sbb-vendor>
      <sbb-version>1.0</sbb-version>
      <sbb-alias>childSbb</sbb-alias>
    </sbb-ref>

    <sbb-classes>
      <sbb-abstract-class
        reentrant="True">
        <sbb-abstract-class-name>org.mobicens.slee.example.sjr.sip.SIPRegistrarSbb</sbb-
abstract-class-name>
        <get-child-relation-method>
          <sbb-alias-ref>childSbb</sbb-alias-ref>
          <get-child-relation-method-name>getChildRelation</get-child-relation-method-name>
          <default-priority>0</default-priority>
        </get-child-relation-method>
        </sbb-abstract-class>
```



```

....

</sbb-classes>

...

</sbb>
</sbb-jar>

```

Parent declaration in descriptor defines method name, in case of example its `getChildRelation`, implemented by JSLEE. This method allows parent to access `javax.slee.ChildRelation` object representing link to defined child. `ChildRelation` object gives access to `SbbLocalObject` interface. This object allows parent to:

- attach child to `ActivityContextInterface` to make it eligible to receive events.
- invoke synchronously methods defined in child `SbbLocalObject`.

Its defined in source as follows:

```

public abstract class SIPRegistrarSbb implements Sbb, DataSourceParentSbbLocalInterface{

    ...

    public abstract ChildRelationExt getChildRelation();

    ...
}

```

Refer to [Section 4.3, “DataSourceChild SBB”](#) for definition of child interface.

4.2.3. REGISTER handler

The `SIPRegistrarSbb` performs following operations in `REGISTER` event handler:

- check if request is a query, if so send response with list of contacts

```

// get configuration from MBean
final long maxExpires = config.getSipRegistrationMaxExpires();

```

```
final long minExpires = config.getSipRegistrationMinExpires();

// Process require header

// Authenticate
// Authorize
// OK we're authorized now ;-)

// extract address-of-record
String sipAddressOfRecord = getCanonicalAddress((HeaderAddress) event
    .getRequest().getHeader(ToHeader.NAME));

if (this.tracer.isFineEnabled()) {
    this.tracer
        .fine("onRegisterEvent: address-of-record from request= "
            + sipAddressOfRecord);
}

final String sipAddress = getCanonicalAddress((HeaderAddress) event
    .getRequest().getHeader(ToHeader.NAME));
final String callId = ((CallIdHeader) event.getRequest().getHeader(
    CallIdHeader.NAME)).getCallId();
final long cSeq = ((CSeqHeader) event.getRequest().getHeader(
    CSeqHeader.NAME)).getSeqNumber();
if (event.getRequest().getHeader(ContactHeader.NAME) == null) {
    // Just send OK with current bindings - this request was a
    // query.
    if (this.tracer.isFineEnabled()) {
        this.tracer.fine("query for bindings: sipAddress="
            + sipAddress);
    }
    try {
        DataSourceChildSbbLocalInterface child = (DataSourceChildSbbLocalInterface) getChildRelation()
            .create(ChildRelationExt.DEFAULT_CHILD_NAME);
        child.getBindings(sipAddress);
    } catch (Exception e) {
        tracer.severe("Exception invoking data source child sbb.",
            e);
        aci.detach(sbbContextExt.getSbbLocalObject());
        sendErrorResponse(Response.SERVER_INTERNAL_ERROR,
            event.getServerTransaction());
    }
    return;
}
```

- check if request is remove action, if so remove bindings from data base

```

// else its update/add/remove op
// do some prechecks before sending task to JDBC RA. JDBC exeutes
// batch in another set of threads. To avoid
// latency, lets check if we can deny REGISTER without pushing job
// to JDBC RA threads.
List<ContactHeader> newContacts = getContactHeaderList(event
    .getRequest().getHeaders(ContactHeader.NAME));

final ExpiresHeader expiresHeader = event.getRequest().getExpires();

if (hasWildCard(newContacts)) {
    if (this.tracer.isFineEnabled()) {
        this.tracer.fine("Wildcard remove");
    }
    // This is a "Contact: *" "remove all bindings" request
    if ((expiresHeader == null)
        || (expiresHeader.getExpires() != 0)
        || (newContacts.size() > 1)) {
        // malformed request in RFC3261 ch10.3 step 6
        aci.detach(sbbContextExt.getSbbLocalObject());
        sendErrorResponse(Response.BAD_REQUEST,
            event.getServerTransaction());
        return;
    }

    // now do the work. in jdbc task
    try {
        DataSourceChildSbbLocalInterface child = (DataSourceChildSbbLocalInterface) getChildRelation()
            .create(ChildRelationExt.DEFAULT_CHILD_NAME);
        child.removeBindings(sipAddress, callId, cSeq);
    } catch (Exception e) {
        tracer.severe("Exception invoking data source child sbb.",
            e);
        aci.detach(sbbContextExt.getSbbLocalObject());
        sendErrorResponse(Response.SERVER_INTERNAL_ERROR,
            event.getServerTransaction());
        return;
    }
}

```

```
} else { ... }
```

- check condition for update, update bindings, add,remove and send response

```
if(){
    // ...
} else {
    // Update bindings
    if (this.tracer.isFineEnabled()) {
        this.tracer.fine("Updating bindings");
    }
    ListIterator<ContactHeader> li = newContacts.listIterator();

    while (li.hasNext()) {
        final ContactHeader contact = (ContactHeader) li.next();

        // get expires value, either in header or default
        // do min-expires etc
        long requestedExpires = 0;

        if (contact.getExpires() >= 0) {
            requestedExpires = contact.getExpires();
        } else if ((expiresHeader != null)
            && (expiresHeader.getExpires() >= 0)) {
            requestedExpires = expiresHeader.getExpires();
        } else {
            requestedExpires = 3600; // default
        }

        // If expires too large, reset to our local max
        if (requestedExpires > maxExpires) {
            requestedExpires = maxExpires;
        }

        } else if ((requestedExpires > 0)
            && (requestedExpires < minExpires)) {
            // requested expiry too short, send response with
            // min-expires
            //
            aci.detach(sbbContextExt.getSbbLocalObject());
            sendErrorResponse(Response.INTERVAL_TOO_BRIEF,
```

```

        event.getServerTransaction());
    return;
}

try {
    contact.setExpires((int) requestedExpires);
} catch (InvalidArgumentException e) {
    tracer.severe("failed to set expires?!?!", e);
    aci.detach(sbbContextExt.getSbbLocalObject());
    sendErrorResponse(Response.SERVER_INTERNAL_ERROR,
        event.getServerTransaction());
    return;
}

// Get the q-value (preference) for this binding - default
// to 0.0 (min)
float q = 0;
if (contact.getQValue() != -1)
    q = contact.getQValue();
if ((q > 1) || (q < 0)) {
    aci.detach(sbbContextExt.getSbbLocalObject());
    sendErrorResponse(Response.BAD_REQUEST,
        event.getServerTransaction());
    return;
}

}

// play with DB :)
try {
    DataSourceChildSbbLocalInterface child = (DataSourceChildSbbLocalInterface) getChildRelation()
        .create(ChildRelationExt.DEFAULT_CHILD_NAME);
    child.updateBindings(sipAddress, callId, cSeq, newContacts);
} catch (Exception e) {
    tracer.severe("Exception invoking data source child sbb.",
        e);
    aci.detach(sbbContextExt.getSbbLocalObject());
    sendErrorResponse(Response.SERVER_INTERNAL_ERROR,
        event.getServerTransaction());
    return;
}
}

```

4.2.4. TIMER handler

The `SIPRegistrarSbb` listens for timer events. Receiving one, is equal to expiration of certain contact. The AOR and expired contact are determined by value of ACI variable of custom activity context interface. The interface is defined as follows:

```
public interface SbbActivityContextInterface extends
    ActivityContextInterfaceExt {

    public RegistrationBindingData getData();

    public void setData(RegistrationBindingData value);

}
```

Since SLEE 1.1 allows custom ACI in event handler method signature, timer event handler looks as follows:

```
public void onTimerEvent(TimerEvent timer, SbbActivityContextInterface aci) {
    // detach from this activity, we don't want to handle any other event on
    // it
    aci.detach(this.sbbContextExt.getSbbLocalObject());
    // get data needed to remove binding from aci
    RegistrationBindingData data = aci.getData();
    if (data == null) {
        // another service's timer event, ignore
        return;
    }
    try {
        DataSourceChildSbbLocalInterface child = (DataSourceChildSbbLocalInterface) getChildRelation()
            .create(ChildRelationExt.DEFAULT_CHILD_NAME);
        child.removeBinding(data.getContact(), data.getAddress());
    } catch (Exception e) {
        tracer.severe("Exception invoking data source child sbb.", e);
        return;
    }
    // end the activity
}
```

```
try {
    ((NullActivity) aci.getActivity()).endActivity();
} catch (Exception e) {
    tracer.warning("failed to end binding aci", e);
}
}
```

4.3. DataSourceChild SBB

`DataSourceChild SBB` is responsible for two tasks. First one is to persist registration data in some sort of a storage. Second is to callback `SIP Registrar SBB` with result of operation invoked on `DataSourceChild SBB` local interface.

This example uses `DataSourceChild SBB` implementation which interacts with database by means of JDBC Resource Adaptor. This following sections contain not only general description of `DataSourceChild SBB`, they contain description of specific implementation of said SBB.

4.3.1. Local interface

`DataSourceChild SBB` declares `SBB Local` interface. This interface is used by `SIP Registrar SBB` to trigger `DataSourceChild SBB` to perform certain operation on storage in which registration data is being held. So in short it defines contract between `SIP Registrar SBB` and `DataSourceChild SBB`. Local interface is declared in `sbb-jar.xml` in following way:

```
<sbb-jar>

...

<sbb>
  <sbb-name>DataSourceChild</sbb-name>
  <sbb-vendor>org.mobicens</sbb-vendor>
  <sbb-version>1.0</sbb-version>

  <sbb-classes>
    <sbb-abstract-class
      reentrant="True">
      <sbb-abstract-class-name>
        org.mobicens.slee.example.sjr.data.jdbc.DataSourceChildSbb
      </sbb-abstract-class-name>
    </sbb-abstract-class>
    <sbb-local-interface>
      <sbb-local-interface-name>
```

```
        org.mobicens.slee.example.sjr.data.DataSourceChildSbbLocalObject
    </sbb-local-interface-name>
</sbb-local-interface>
</sbb-classes>

...

</sbb>
</sbb-jar>
```

Child interface is declared as follows:

```
public interface DataSourceChildSbbLocalObject extends SbbLocalObject,
    DataSourceChildSbbLocalInterface {

}
```

Above methods([Section 4.1.2, “Data Source Child”](#)) are invoked by SIP Registrar SBB as result of registrar routines. According to name, methods perform certain task:

Initiate data storage

```
@Override
public void init() {
    // create db schema if needed
    Connection connection = null;
    try {
        connection = jdbcRA.getConnection();
        connection.createStatement().execute(
            DataSourceSchemaInfo._QUERY_CREATE);
    } catch (SQLException e) {
        tracer.warning("failed to create db schema", e);
    } finally {
        try {
            connection.close();
        } catch (SQLException e) {
            tracer.severe("failed to close db connection", e);
        }
    }
}
```



```
    }  
  }  
}
```

Fetch bindings for AOR

```
@Override  
public void getBindings(String address) {  
    executeTask(new GetBindingsJdbcTask(address, tracer));  
}
```

Remove single binding

```
@Override  
public void removeBinding(String contact, String address) {  
    executeTask(new RemoveBindingJdbcTask(address, contact, tracer));  
}
```

Remove all bindings

```
@Override  
public void removeBindings(String address, String callId, long cSeq) {  
    executeTask(new RemoveBindingsJdbcTask(address, callId, cSeq, tracer));  
}
```

Update/Add bindings

```
@Override
public void updateBindings(String address, String callId, long cSeq,
    List<ContactHeader> contacts) {
    executeTask(new UpdateBindingsJdbcTask(address, callId, cSeq, contacts, tracer));
}
```

4.3.2. Parent relation

The `DataSourceChild SBB` in this example has a parent. In this example, the parent is `SIP Registrar SBB`. The `DataSourceChild SBB` accesses its parent by using JBoss Communications SLEE extension API:

```
final DataSourceParentSbbLocalInterface parent = (DataSourceParentSbbLocalInterface) sbbContextExt
    .getSbbLocalObject().getParent();
```

4.3.3. JDBC RA Tasks

Default implementation of `DataSourceChild SBB` uses JDBC RA to persist user contact information. It performs its task using set of JDBC RA tasks.

Each task use queries defined in `org.mobicens.slee.example.sjr.data.jdbc.DataSourceSchemaInfo`. Example of such query, looks as follows:

```
public class DataSourceSchemaInfo {

    // lets define some statics for table and queries in one place.
    public static final String _TABLE_NAME = "SIP_REGISTRAR";
    public static final String _COLUMN_SIP_ADDRESS = "SIP_ADDRESS";
    public static final String _COLUMN_CONTACT = "CONTACT";
    public static final String _COLUMN_EXPIRES = "EXPIRES";
    public static final String _COLUMN_REGISTER_DATE = "REGISTER_DATE";
    public static final String _COLUMN_Q = "Q";
    public static final String _COLUMN_CALL_ID = "CALL_ID";
    public static final String _COLUMN_C_SEQ = "C_SEQ";
```

```

// SQL queries.
// drop table
public static final String _QUERY_DROP = "DROP TABLE IF EXISTS "
    + _TABLE_NAME + ";";
// create table, use contact as PK, since it will be unique ?
// | CONTACT (PK) | SIP_ADDRESS (PK) | Q | EXPIRES | REGISTER_DATE | CALL_ID
// | C_SEQ |
public static final String _QUERY_CREATE = "CREATE TABLE " + _TABLE_NAME
    + " (" + _COLUMN_CONTACT + " VARCHAR(255) NOT NULL, "
    + _COLUMN_SIP_ADDRESS + " VARCHAR(255) NOT NULL, " + _COLUMN_Q
    + " FLOAT NOT NULL, " + _COLUMN_EXPIRES + " BIGINT NOT NULL, "
    + _COLUMN_REGISTER_DATE + " BIGINT NOT NULL, " + _COLUMN_C_SEQ
    + " BIGINT NOT NULL, " + _COLUMN_CALL_ID
    + " VARCHAR(255) NOT NULL, " + "PRIMARY KEY(" + _COLUMN_CONTACT
    + "," + _COLUMN_SIP_ADDRESS + ")" + ");";

...

// update row for AOR and contact
public static final String _QUERY_UPDATE = "UPDATE " + _TABLE_NAME + " "
    + "SET " + _COLUMN_CALL_ID + "=?," + _COLUMN_C_SEQ + "=?,"
    + _COLUMN_EXPIRES + "=?," + _COLUMN_Q + "=?,"
    + _COLUMN_REGISTER_DATE + "=? " + "WHERE " + _COLUMN_SIP_ADDRESS
    + "=? AND " + _COLUMN_CONTACT + "=?";

...

}

```

Following tasks are defined to perform operations on persistent storage:

GetBindingsJdbcTask

```

public class GetBindingsJdbcTask extends DataSourceJdbcTask {

    private List<RegistrationBinding> bindings = null;

    private final String address;
    private final Tracer tracer;

```

```
public GetBindingsJdbcTask(String address, Tracer tracer) {
    this.address = address;
    this.tracer = tracer;
}

@Override
public Object executeSimple(JdbcTaskContext taskContext) {
    try {
        PreparedStatement preparedStatement = taskContext.getConnection()
            .prepareStatement(DataSourceSchemaInfo._QUERY_SELECT);
        preparedStatement.setString(1, address);
        preparedStatement.execute();
        ResultSet resultSet = preparedStatement.getResultSet();
        bindings = DataSourceSchemaInfo.getBindingsAsList(address, resultSet);
    } catch (Exception e) {
        tracer.severe("failed to execute task to get bindings of "+address,e);
    }
    return this;
}

@Override
public void callBackParentOnException(
    DataSourceParentSbbLocalInterface parent) {
    parent.getBindingsResult(Response.SERVER_INTERNAL_ERROR,
        EMPTY_BINDINGS_LIST);
}

@Override
public void callBackParentOnResult(DataSourceParentSbbLocalInterface parent) {
    if (bindings == null) {
        parent.getBindingsResult(Response.SERVER_INTERNAL_ERROR, EMPTY_BINDINGS_LIST);
    }
    else {
        parent.getBindingsResult(Response.OK, bindings);
    }
}
}
```

RemoveBindingJdbcTask

```

public class RemoveBindingJdbcTask extends DataSourceJdbcTask {

    private final String address;
    private final String contact;
    private final Tracer tracer;

    public RemoveBindingJdbcTask(String address, String contact, Tracer tracer) {
        this.address = address;
        this.contact = contact;
        this.tracer = tracer;
    }

    @Override
    public Object executeSimple(JdbcTaskContext taskContext) {
        try {
            PreparedStatement preparedStatement = taskContext.getConnection()
                .prepareStatement(DataSourceSchemaInfo._QUERY_DELETE);
            preparedStatement.setString(1, address);
            preparedStatement.setString(2, contact);
            preparedStatement.execute();
            if (this.tracer.isInfoEnabled()) {
                this.tracer.info("Removed binding: " + address
                    + " -> " + contact);
            }
        } catch (Exception e) {
            tracer.severe("failed to remove binding", e);
        }
        return this;
    }

    @Override
    public void callBackParentOnException(
        DataSourceParentSbbLocalInterface parent) {
        // nothing to call back
    }

    @Override
    public void callBackParentOnResult(DataSourceParentSbbLocalInterface parent) {
        // nothing to call back
    }
}

```

RemoveBindingsJdbcTask

```
public class RemoveBindingsJdbcTask extends DataSourceJdbcTask {

    private int resultCode = Response.OK;
    private List<RegistrationBinding> currentBindings = null;
    private List<RegistrationBinding> removedBindings = null;

    private final String address;
    private final String callId;
    private final long cSeq;

    private final Tracer tracer;

    public RemoveBindingsJdbcTask(String address, String callId, long cSeq,
        Tracer tracer) {
        this.address = address;
        this.callId = callId;
        this.cSeq = cSeq;
        this.tracer = tracer;
    }

    @Override
    public Object executeSimple(JdbcTaskContext taskContext) {

        SleeTransaction tx = null;
        try {
            tx = taskContext.getSleeTransactionManager().beginSleeTransaction();

            Connection connection = taskContext.getConnection();
            // static value of query string, since its widely used :)
            PreparedStatement preparedStatement = connection
                .prepareStatement(DataSourceSchemaInfo._QUERY_SELECT);
            preparedStatement.setString(1, address);

            preparedStatement.execute();
            ResultSet resultSet = preparedStatement.getResultSet();
            // IMPORTANT: we need both - currently present bindings and removed
            // ones
            // so SBB can update timers
            currentBindings = DataSourceSchemaInfo.getBindingsAsList(address, resultSet);
            List<RegistrationBinding> removedBindings = new ArrayList<RegistrationBinding>();
            Iterator<RegistrationBinding> it = currentBindings.iterator();
```

```

while (it.hasNext()) {
    RegistrationBinding binding = it.next();
    if (callId.equals(binding.getCallId())) {
        if (cSeq > binding.getCSeq()) {
            it.remove();
            removedBindings.add(binding);
            preparedStatement = connection
                .prepareStatement(DataSourceSchemaInfo._QUERY_DELETE);
            preparedStatement.setString(1, address);
            preparedStatement.setString(2,
                binding.getContactAddress());
            preparedStatement.execute();
            if (this.tracer.isInfoEnabled()) {
                this.tracer.info("Removed binding: " + address
                    + " -> " + binding.getContactAddress());
            }
        }
        else {
            resultCode = Response.BAD_REQUEST;
            return this;
        }
    }
    else {
        removedBindings.add(binding);
        preparedStatement = connection
            .prepareStatement(DataSourceSchemaInfo._QUERY_DELETE);
        preparedStatement.setString(1, address);
        preparedStatement.setString(2, binding.getContactAddress());
        preparedStatement.execute();
        if (this.tracer.isInfoEnabled()) {
            this.tracer.info("Removed binding: " + address
                + " -> " + binding.getContactAddress());
        }
    }
}

tx.commit();
tx = null;

} catch (Exception e) {
    tracer.severe("Failed to execute task", e);
    resultCode = Response.SERVER_INTERNAL_ERROR;
} finally {

```

```
        if (tx != null) {
            try {
                tx.rollback();
            } catch (Exception f) {
                tracer.severe("failed to rollback tx", f);
            }
        }
    }

    return this;
}

@Override
public void callBackParentOnException(
    DataSourceParentSbbLocalInterface parent) {
    parent.removeBindingsResult(Response.SERVER_INTERNAL_ERROR,
        EMPTY_BINDINGS_LIST, EMPTY_BINDINGS_LIST);
}

@Override
public void callBackParentOnResult(DataSourceParentSbbLocalInterface parent) {
    if (resultCode > 299) {
        parent.removeBindingsResult(resultCode, EMPTY_BINDINGS_LIST,
            EMPTY_BINDINGS_LIST);
    } else {
        parent.removeBindingsResult(resultCode, currentBindings,
            removedBindings);
    }
}
}
```

UpdateBindingsJdbcTask

```
public class UpdateBindingsJdbcTask extends DataSourceJdbcTask {

    private int resultCode = Response.OK;
    private List<RegistrationBinding> currentBindings = null;
    private List<RegistrationBinding> updatedBindings = null;
    private List<RegistrationBinding> removedBindings = null;
```



```

private final String address;
private final String callId;
private final long cSeq;
private final List<ContactHeader> contacts;

private final Tracer tracer;

public UpdateBindingsJdbcTask(String address, String callId, long cSeq,
    List<ContactHeader> contacts, Tracer tracer) {
    this.address = address;
    this.callId = callId;
    this.cSeq = cSeq;
    this.contacts = contacts;
    this.tracer = tracer;
}

@Override
public Object executeSimple(JdbcTaskContext taskContext) {

    // fetch those values, yes, we do some things twice, its done to
    // avoid pushing everythin into JDBC execute.

    // final ExpiresHeader expiresHeader =
    // super.event.getRequest().getExpires();
    ListIterator<ContactHeader> li = this.contacts.listIterator();
    SleeTransaction tx = null;
    try {
        tx = taskContext.getSleeTransactionManager().beginSleeTransaction();
        Connection connection = taskContext.getConnection();
        // static value of query string, since its widely used :)
        PreparedStatement preparedStatement = connection
            .prepareStatement(DataSourceSchemaInfo._QUERY_SELECT);
        preparedStatement.setString(1, address);

        preparedStatement.execute();
        ResultSet resultSet = preparedStatement.getResultSet();

        // lets have it as map, it will be easier to manipulate in this
        // case.
        Map<String, RegistrationBinding> bindings = DataSourceSchemaInfo
            .getBindingsAsMap(address, resultSet);
        removedBindings = new ArrayList<RegistrationBinding>();
        updatedBindings = new ArrayList<RegistrationBinding>();
    }
}

```

```
while (li.hasNext()) {
    ContactHeader contact = li.next();

    //
    // get expires value, either in header or default
    // do min-expires etc
    long requestedExpires = contact.getExpires();

    float q = 0;
    if (contact.getQValue() != -1)
        q = contact.getQValue();

    // Find existing binding
    String contactAddress = contact.getAddress().getURI()
        .toString();

    RegistrationBinding binding = (RegistrationBinding) bindings
        .get(contactAddress);

    if (binding != null) { // Update this binding

        if (this.callId.equals(binding.getCallId())) {
            if (this.cSeq <= binding.getCSeq()) {
                resultCode = Response.BAD_REQUEST;
                return this;
            }
        }

        if (requestedExpires == 0) {
            bindings.remove(contactAddress);
            removedBindings.add(binding);
            preparedStatement = connection
                .prepareStatement(DataSourceSchemaInfo._QUERY_DELETE);
            preparedStatement.setString(1, address);
            preparedStatement.setString(2,
                binding.getContactAddress());
            preparedStatement.execute();
            if (this.tracer.isInfoEnabled()) {
                this.tracer.info("Removed binding: " + address
                    + " -> " + contactAddress);
            }
        } else {
            // update binding in map, it will be sent back
            binding.setCallId(callId);
        }
    }
}
```

```

binding.setExpires(requestedExpires);
binding.setRegistrationDate(System.currentTimeMillis());
binding.setCSeq(this.cSeq);
binding.setQValue(q);
updatedBindings.add(binding);
// update DB
preparedStatement = connection
    .prepareStatement(DataSourceSchemaInfo._QUERY_UPDATE);
preparedStatement.setString(1, binding.getCallId());
preparedStatement.setLong(2, binding.getCSeq());
preparedStatement.setLong(3, binding.getExpires());
preparedStatement.setFloat(4, binding.getQValue());
preparedStatement.setLong(5,
    binding.getRegistrationDate());

preparedStatement.setString(6, address);
preparedStatement.setString(7,
    binding.getContactAddress());
preparedStatement.execute();
if (this.tracer.isInfoEnabled()) {
    this.tracer.info("Updated binding: " + address
        + " -< " + contactAddress);
}
}

} else {

    // Create new binding
    if (requestedExpires != 0) {
        RegistrationBinding newRegistrationBinding = new RegistrationBinding(
            address, contactAddress, requestedExpires,
            System.currentTimeMillis(), q, callId,
            this.cSeq);
        // put in bindings
        bindings.put(
            newRegistrationBinding.getContactAddress(),
            newRegistrationBinding);
        updatedBindings.add(newRegistrationBinding);
        // update DB
        preparedStatement = connection
            .prepareStatement(DataSourceSchemaInfo._QUERY_INSERT);
        preparedStatement.setString(1,
            newRegistrationBinding.getCallId());
        preparedStatement.setLong(2,

```

```

        newRegistrationBinding.getCSeq());
    preparedStatement.setLong(3,
        newRegistrationBinding.getExpires());
    preparedStatement.setFloat(4,
        newRegistrationBinding.getQValue());
    preparedStatement.setLong(5,
        newRegistrationBinding.getRegistrationDate());

    preparedStatement.setString(6, address);
    preparedStatement.setString(7,
        newRegistrationBinding.getContactAddress());
    preparedStatement.execute();
    if (this.tracer.isInfoEnabled()) {
        this.tracer.info("Added new binding: " + address
            + " -< " + contactAddress);
    }
}
}
}
// now lets push current bindings
currentBindings = new ArrayList<RegistrationBinding>(
    bindings.values());
tx.commit();
tx = null;

} catch (Exception e) {
    tracer.severe("Failed to execute jdbc task.", e);
    resultCode = Response.SERVER_INTERNAL_ERROR;
} finally {

    if (tx != null) {
        try {
            tx.rollback();
        } catch (Exception f) {
            tracer.severe("failed to rollback tx", f);
        }
    }
}

return this;
}

@Override
public void callBackParentOnException(

```

```

        DataSourceParentSbbLocalInterface parent) {
    parent.updateBindingsResult(Response.SERVER_INTERNAL_ERROR,
        EMPTY_BINDINGS_LIST, EMPTY_BINDINGS_LIST, EMPTY_BINDINGS_LIST);
}

@Override
public void callBackParentOnResult(DataSourceParentSbbLocalInterface parent) {
    if (resultCode > 299) {
        parent.updateBindingsResult(resultCode, EMPTY_BINDINGS_LIST,
            EMPTY_BINDINGS_LIST, EMPTY_BINDINGS_LIST);
    } else {
        parent.updateBindingsResult(resultCode, currentBindings,
            updatedBindings, removedBindings);
    }
}
}
}

```

4.3.4. JDBC Task Exception handler

The `JdbcTaskExecutionThrowableEvent` handler is fairly simple as it only invokes parent:

```

public void onJdbcTaskExecutionThrowableEvent(
    JdbcTaskExecutionThrowableEvent event, ActivityContextInterface aci) {
    if (tracer.isWarningEnabled()) {
        tracer.warning(
            "Received a JdbcTaskExecutionThrowableEvent, as result of executed task "
            + event.getTask(), event.getThrowable());
    }
    // end jdbc activity
    final JdbcActivity activity = (JdbcActivity) aci.getActivity();
    activity.endActivity();
    // call back parent
    final DataSourceParentSbbLocalInterface parent = (DataSourceParentSbbLocalInterface) sbbContextExt
        .getSbbLocalObject().getParent();
    final DataSourceJdbcTask jdbcTask = (DataSourceJdbcTask) event
        .getTask();
    jdbcTask.callBackParentOnException(parent);
}

```

4.3.5. JDBC Task Result handler

Similar to [Section 4.3.4, “JDBC Task Exception handler”](#), the `SimpleJdbcTaskResultEvent` is also very simple:

```
public void onSimpleJdbcTaskResultEvent(SimpleJdbcTaskResultEvent event,
    ActivityContextInterface aci) {
    if (tracer.isFineEnabled()) {
        tracer.fine("Received a SimpleJdbcTaskResultEvent, as result of executed task "
            + event.getTask());
    }
    // end jdbc activity
    final JdbcActivity activity = (JdbcActivity) aci.getActivity();
    activity.endActivity();
    // call back parent
    final DataSourceParentSbbLocalInterface parent = (DataSourceParentSbbLocalInterface) sbbContextExt
        .getSbbLocalObject().getParent();
    final DataSourceJdbcTask jdbcTask = (DataSourceJdbcTask) event
        .getTask();
    jdbcTask.callBackParentOnResult(parent);
}
```

Running the Example

The easiest way to test example is to deploy example and start container, and run `SIPP` scripts from `sipp` directory.

5.1. Configuration

There are only two parameter which can be configured. Both are exposed as JMX properties. The JMX bean can be accessed under `slee:sipregistrarconfigurator=v2RegistrarConf` name.

It exposes following properties:

`minExpires`

sets lower boundry acceptable for `Expires` header. Its expressed in seconds and has default value of 120.

`maxExpires`

sets upper boundry for `Expires` header. Its expressed in seconds and has default value of 3600.

Traces and Alarms

6.1. Tracers

The example Application uses multiple JAIN SLEE 1.1 Tracer facility instances. Below is full list:

Table 6.1. SIP JDBC Registrar Tracer and Log Categories

Sbb	Tracer name	LOG4J category
SIP Registrar	SipRegistrar	javax.slee.SbbNotification[service=ServiceID[name=Sip Registrar,vendor=org.mobicients,version=1.0],sbb=SbbID[name=SIP Registrar,vendor=org.mobicients,version=1.0]] .SipRegistrar
DataSourceChild	DataSourceChildSbb	javax.slee.SbbNotification[service=ServiceID[name=Sip Registrar,vendor=org.mobicients,version=1.0],sbb=SbbID[name=DataSourceChild,vendor=org.mobicients,version=1.0]] .DataSourceChildSbb



Important

Spaces were introduced in LOG4J category column values, to correctly render the table. Please remove them when using copy/paste.

6.2. Alarms

The example Application does not sets JAIN SLEE Alarms.

Appendix A. Revision History

Revision History

Revision 1.0

Thu Dec 30 2011

BartoszBaranowski

Creation of the JBoss Communications JAIN SLEE SIP JDBC Registrar Example User Guide.

Index

F

feedback, viii

