# HIBERSAP

The easy way to connect Java with SAP

Version 1.1.0Beta01

Copyright ©2008-2011 akquinet tech@spree GmbH

Copies of this document may be made for your own use and for distribution to others, provided that you do not charge any fee for such copies and further provided that each copy contains this Copyright Notice, whether distributed in print or electronically.

# Contents


**1 Introduction to Hibersap** **1**

**2 Quick Start** **3**

2.1 Installation . . . 3

2.2 The first Hibersap application . . . 4

2.2.1 The function module . . . 4

2.2.2 The BAPI class . . . 6

2.2.3 Structure classes . . . 8

2.2.4 Configuration . . . 11

2.2.5 Calling a function module . . . 11

**3 Installation** **15**

3.1 The Hibersap libraries . . . 15

3.2 Third party libraries . . . 15

3.2.1 Hibersap dependencies . . . 15

3.2.2 The SAP Java Connector (JCo) . . . 15

3.2.3 The SAP BAPI JCA Adapter . . . 16

3.2.4 Using other JCA compatible Resource Adapters . . . 16

3.3 Building Hibersap applications with Maven . . . 16

**4 Configuration** **19**

4.1 XML file configuration . . . 19

4.2 Programmatic configuration . . . 21

**5 Usage** **23**

5.1 Mapping Remote Function Modules to Java classes . . . 23

5.1.1 The annotations . . . 23

5.1.2 Type Conversion . . . 25

5.2 Calling Function Modules . . . 27

5.2.1 Configure Hibersap and build a SessionManager . . . 27

5.2.2 Calling a function in SAP . . . 27

5.3 Transaction Management . . . 28

5.3.1 Manual transaction handling . . . 28

5.3.2 Container Managed Transactions . . . . . 29
5.4 Authentication . . . . . 29
5.4.1 Using a central User . . . . . 29
5.4.2 Per-session authentication . . . . . 29
5.4.3 Single sign-on . . . . . 30
5.5 Java EE application server integration . . . . . 30
5.5.1 Transaction Management . . . . . 30
5.5.2 Security Management . . . . . 31
5.5.3 Configuration . . . . . 31

# Chapter 1

# Introduction to Hibersap

Hibersap is a small framework that offers an abstraction layer on top of the SAP Java Connector (JCo). It maps Java classes to SAP function modules using Java Annotations and reduces the technical code to call a function in a SAP back-end system to a minimum. Hibersap's API is very similar to Hibernate, thus offering a familiar programming interface to developers.

## Current Features

- Calling remote function modules in SAP systems
- Mapping of ABAP function modules and their parameters to Java classes and their fields using Java Annotations.
- Hibernate-like API completely hiding the JCo/JCA API.
- Custom data type conversion using @Convert annotation and Converter classes.
- Intercept execution of function modules using custom Interceptors.
- Automatic error propagation using the @ThrowExceptionOnError annotation.
- Configuration using XML file or programmatic configuration
- Using the SAP Java Connector (JCo), version 3, internally.
- Alteratively, using a JCA compliant Resource Adapter in managed environments.
- Switching between the use of JCo and JCA by means of configuration.
- Manual transaction handling using JCo.
- Local Transactions using JCA.
- Container Managed Transactions (CMT) using JCA.
- Distributed Transactions with Application Servers that support Last Resource Commit Optimization.
- Application and container managed security.

## Prospective Features

- Automated lookup of mapped classes instead of adding them manually to the configuration.
- JBoss5 deployer for annotated Hibersap classes.
- Hibernate-like current session context strategy.
- Integration of the Hibernate Validation Framework to validate mapped fields.
- Using JCo 3 stateless calls.

In chapter 2 you'll find out how easy it is to use Hibersap.

# Chapter 2

# Quick Start

## 2.1 Installation

- Make sure Maven2 is installed on your computer.
- Download SAP Java Connector 3 (http://service.sap.com/connectors) and extract the sapjco3.jar and the sapjco3 native library. (In the example, we assume you use version 3.0.1, if not you should use the correct version number.)
- Install sapjco3 jar to your local Maven repository from the commandline: "mvn install:install -file -DgroupId=com.sap -DartifactId=sap-jco -Dversion=3.0.1 -Dpackaging=jar -Dfile=sapjco3.jar".
- Create a Maven project with the repositories and dependencies elements like in listing 2.1.
- Put the JCo native library into your library path or into the folder your application will be started from (e. g. the root folder of your Eclipse project that contains your main class).
- Put the Java classes that are described in this chapter into the folder $project_home/src/main/java.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://
             maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.mycomp</groupId>
    <artifactId>hibersap-example</artifactId>
    <version>1.0</version>
    <repositories>
        <repository>
            <id>repository.hibersap</id>
            <name>Hibersap Repository for Maven</name>
            <url>
              http://hibersap.svn.sourceforge.net/viewvc/hibersap/m2repo
            </url>
        </repository>
```

```
    </repositories>
    <dependencies>
        <dependency>
            <groupId>org.hibersap</groupId>
            <artifactId>hibersap-core</artifactId>
            <version>1.1.0Beta01</version>
        </dependency>
        <dependency>
            <groupId>org.hibersap</groupId>
            <artifactId>hibersap-jco</artifactId>
            <version>1.1.0Beta01</version>
        </dependency>
        <dependency>
            <groupId>com.sap</groupId>
            <artifactId>sap-jco</artifactId>
            <version>3.0.1</version>
        </dependency>
    </dependencies>
</project>
```

Listing 2.1: The Maven pom.xml

If you don't want to use Maven to build the application, you can look up the dependencies of hibersap-core and hibersap-jco under the following link: http://hibersap.sourceforge.net/dependency-convergence.html

## 2.2 The first Hibersap application

We are going to write a small example application that will call the SAP function BAPI_SFLIGHT_GETLIST. This function is part of a demo application in SAP that offers a simplified flight-booking system. The code can be found in project `hibersap-examples`. The same rules apply for that project, i.e. the native library must be in that project's directory, too.

### 2.2.1 The function module

This is the function module's interface in SAP:

```
FUNCTION BAPI_SFLIGHT_GETLIST.
  IMPORTING
     VALUE(FROMCOUNTRYKEY) LIKE   BAPISFDETA-COUNTRYFR
     VALUE(FROMCITY)       LIKE   BAPISFDETA-CITYFROM
     VALUE(TOCOUNTRYKEY)   LIKE   BAPISFDETA-COUNTRYTO
     VALUE(TOCITY)         LIKE   BAPISFDETA-CITYTO
     VALUE(AIRLINECARRIER) LIKE   BAPISFDETA-CARRID DEFAULT SPACE
     VALUE(AFTERNOON)      LIKE   BAPI_AUX-AFTERNOON DEFAULT SPACE
     VALUE(MAXREAD)        LIKE   BAPI_AUX-MAXREAD DEFAULT 0
  EXPORTING
     VALUE(RETURN)         LIKE   BAPIRET2 STRUCTURE   BAPIRET2
  TABLES
```

```
FLIGHTLIST                STRUCTURE  BAPISFLIST
```

Listing 2.2: The ABAP Function

This function module takes some parameters that represent search criteria to look up flights in SAP's database. The matching flights are returned in the FLIGHTLIST table, which contains information such as the airline carrier id, a flight connection code and departure/destination data. The RETURN structure may be filled by SAP with extra messages like errors, warnings, etc.

The import parameters are simple types, whereas the export and table parameter are complex data types (ABAP structures). The RETURN parameter is of type BAPIRET2, which is a standard structure that can be found in many function modules' interfaces and that is not specific to this BAPI. See table 2.1.

| **Component name** | **Type** | **Description** |
|---|---|---|
| TYPE | Character | Message type: S Success, E Error, W Warning, I Info, A Abort |
| ID | Character | Messages, message class |
| NUMBER | Numeric character | Messages, message number |
| MESSAGE | Character | Message text |
| LOG_NO | Character | Application log: log number |
| LOG_MSG_NO | Numeric character | Application log: Internal message serial number |
| MESSAGE_V1 | Character | Messages, message variables |
| MESSAGE_V2 | Character | Messages, message variables |
| MESSAGE_V3 | Character | Messages, message variables |
| MESSAGE_V4 | Character | Messages, message variables |
| PARAMETER | Character | Parameter name |
| ROW | 4-byte integer | Lines in parameter |
| FIELD | Character | Field in parameter |
| SYSTEM | Character | Logical system from which message originates |

Table 2.1: Structure BAPIRET2

The FLIGHTLIST table's lines are of type BAPISFLIST which contains the following fields, as shown in table 2.2.

| **Component name** | **Type** | **Description** |
|---|---|---|
| CARRID | Character | Airline carrier ID |
| CONNID | Numerical character | Flight connection code |
| FLDATE | Date | Flight date |
| AIRPFROM | Character | Airport of departure |
| AIRPTO | Character | Destination airport |
| DEPTIME | Time | Departure time |
| SEATSMAX | 4-byte integer | Maximum capacity |
| SEATSOCC | 4-byte integer | Occupied seats |

Table 2.2: Structure BAPISFLIST

### 2.2.2 The BAPI class

To call a function module from a Java application using Hibersap, we have to write a *BAPI class* that acts as an adapter to JCo. The BAPI class is a simple Java class with a number of fields representing the BAPI's import, export and table parameters. In case the BAPI parameter being a scalar parameter, the Java field itself is of a simple Java type. In the case of a structure parameter, the Java field's type has to be a complex type, too. A table parameter maps to a Collection of a complex type.

All setup related to the function module's interface is done via Java annotations. A BAPI class is defined using the Hibersap class annotation @Bapi, which has a name argument specifying the name of the SAP function module we want to call. (All Hibersap annotations can be found in the package org.hibersap.annotations.)

```
package org.hibersap.examples.flightlist;

import java.util.List;
import org.hibersap.*;

@Bapi("BAPI_SFLIGHT_GETLIST")
public class FlightListBapi
{
  ...
}
```

Listing 2.3: The BAPI class

The fields that are mapped to the function module's parameters are annotated with the @Import, @Export or @Table annotations. Additionally, we have to specify the function module's field name to which it relates, using the @Parameter annotation. The @Parameter's second argument, type, tells Hibersap if the parameter has a simple or a complex structure type. The enumeration ParameterType defines possible values, the default type for element type being SIMPLE. In most cases we have to specify a parameter's name only. Since table parameters always represent a table of complex types, any parameter type argument will be ignored by Hibersap.

```
@Import
@Parameter("FROMCOUNTRYKEY")
private final String fromCountryKey;

@Import
@Parameter("FROMCITY")
private final String fromCity;

@Import
@Parameter("TOCOUNTRYKEY")
private final String toCountryKey;

@Import
@Parameter("TOCITY")
private final String toCity;

@Import
```

```
@Parameter("AIRLINECARRIER")
private final String airlineCarrier;

@Import
@Parameter("AFTERNOON")
@Convert(converter = BooleanConverter.class)
private final boolean afternoon;

@Import
@Parameter("MAXREAD")
private final int maxRead;

@Export
@Parameter(value="RETURN", type = ParameterType.STRUCTURE)
private BapiRet2 returnData;

@Table
@Parameter("FLIGHTLIST")
private List<Flight> flightList;
```

Listing 2.4: The BAPI fields

The Java type of each simple field is related to the SAP field's data type. Hibersap relies on the Java Connector's conversion scheme as shown in table 5.2.

The @Convert annotation on the field afternoon in the listing above tells Hibersap to use a Converter of type BooleanConverter to convert the parameter AFTERNOON which is a character field of length 1 to a Java boolean value. See section 5.1 on how to implement custom Converters.

To conclude the example, we write a constructor which takes all the import parameters as arguments, initializing the corresponding fields.

```
public FlightListBapi( String fromCountryKey,
                       String fromCity,
                       String toCountryKey,
                       String toCity,
                       String airlineCarrier,
                       boolean afternoon,
                       int maxRead )
{
    this.fromCountryKey = fromCountryKey;
    this.fromCity = fromCity;
    this.toCountryKey = toCountryKey;
    this.toCity = toCity;
    this.airlineCarrier = airlineCarrier;
    this.afternoon = afternoon;
    this.maxRead = maxRead;
}
```

Listing 2.5: The constructor

Finally, we should add a getter method for each field. Hibersap does not need setter methods, because all fields are set using reflection. We could of course add additional fields and methods if needed.

```
public boolean getAfternoon()
{
    return this.afternoon;
}
...
```

Listing 2.6: The getter methods

---

**Note** — There is one constraint in the current version of Hibersap you should take into account: The mapping between SAP parameters and Java classes works as expected only if the SAP function module follows the BAPI standard, this means:

- Deep tables (i. e. tables in tables) are not supported.
- Tables in import and export parameters are not supported.
- Changing parameters can not be mapped.
- ABAP exceptions are simply converted to HibersapException.

---

### 2.2.3 Structure classes

There are two more classes we have to write: One for the complex export parameter RETURN, which is named BapiRet2, after the SAP data type. It is another annotated simple Java class with fields related to some of the function module's parameter. To keep the example simple, we do not map all the fields of the RETURN parameter.

```
package org.hibersap.bapi;

import org.hibersap.annotations.*;

@BapiStructure
public class BapiRet2
{
    @Parameter("TYPE")
    @Convert(converter = CharConverter.class)
    private char type;

    @Parameter("ID")
    private String id;

    @Parameter("NUMBER")
    private String number;

    @Parameter("MESSAGE")
    private String message;


    public char getType()
```

```
    {
        return this.type;
    }

    public String getId()
    {
        return this.id;
    }

    public String getNumber()
    {
        return this.number;
    }

    public String getMessage()
    {
        return this.message;
    }
}
```

Listing 2.7: The BapiRet2 class

The class is annotated with @BapiStructure to tell Hibersap that it maps to a complex parameter on the SAP side. Each particular field is annotated with the already known @Parameter annotation that defines the name of the corresponding structure field. The BapiRet2 class is already part of Hibersap, since this structure is used by a lot of SAP function modules. This means, you don't have to implement it.

Second, there is a Java class that Hibersap will map to the table parameter FLIGHTLIST, which in our example is simply called Flight. The table FLIGHTLIST will be filled by SAP with the flight information matching our request.

```
package org.hibersap.examples.flightlist;

import java.util.Date;
import org.hibersap.*;

@BapiStructure
public class Flight
{
    @Parameter("CARRID")
    private String carrierId;

    @Parameter("CONNID")
    private String connectionId;

    @Parameter("AIRPFROM")
    private String airportFrom;

    @Parameter("AIRPTO")
    private String airportTo;

    @Parameter("FLDATE")
```

```
    private Date flightDate;

    @Parameter("DEPTIME")
    private Date departureTime;

    @Parameter("SEATSMAX")
    private int seatsMax;

    @Parameter("SEATSOCC")
    private int seatsOccupied;

    public String getAirportFrom()
    {
        return this.airportFrom;
    }

    public String getAirportTo()
    {
        return this.airportTo;
    }

    public String getCarrierId()
    {
        return this.carrierId;
    }

    public String getConnectionId()
    {
        return this.connectionId;
    }

    public Date getDepartureTime()
    {
        return DateUtil.joinDateAndTime( flightDate, departureTime );
    }

    public Date getFlightDate()
    {
        return flightDate;
    }

    public int getSeatsMax()
    {
        return this.seatsMax;
    }

    public int getSeatsOccupied()
    {
        return this.seatsOccupied;
    }
```

```
}
```

Listing 2.8: The Flight class

Please note that the method getDepartureTime() does not simply return the field departureTime but calls a utility method DateUtil.joinDateAndTime(). This is done here because ABAP does not know a data type that contains date and time, rather a timestamp is separated into two fields, one of type Date, the other of type Time. Therefore the Java Connector returns a java.util.Date for the SAP date field containing the date fraction (date at 00:00:00,000) and another java.util.Date for the time field containing the time fraction (i.e. Jan. 1st, 1970 plus time).

### 2.2.4 Configuration

To configure Hibersap, we need to specify a minimum set of information for the Hibersap framework, plus properties for the Java Connector. To accomplish this, we create an XML file named hibersap.xml in $project_home/src/main/resources/META-INF. Hibersap will look for this file in the classpath under "/META-INF/hibersap.xml".

In the example we use a minimum set of JCo properties to be able to connect to the back-end SAP system. All valid JCo properties are specified in the JCo library interface com.sap.conn.jco.ext.DestinationDataProvider (see javadoc provided with JCo).

```
<?xml version="1.0" encoding="UTF-8"?>
<hibersap xmlns="urn:hibersap:hibersap-configuration:1.0">
  <session-manager name="A12">
    <context>org.hibersap.execution.jco.JCoContext</context>

    <properties>
      <property name="jco.client.client" value="800" />
      <property name="jco.client.user" value="sapuser" />
      <property name="jco.client.passwd" value="password" />
      <property name="jco.client.lang" value="en" />
      <property name="jco.client.ashost" value="10.20.80.76" />
      <property name="jco.client.sysnr" value="00" />
      <property name="jco.destination.pool_capacity" value="1" />
    </properties>

    <annotatedClasses>
      <class>org.hibersap.examples.flightlist.FlightListBapi</class>
    </annotatedClasses>
  </session-manager>
</hibersap>
```

Listing 2.9: hibersap.xml

### 2.2.5 Calling a function module

Hibersap's API is very close to Hibernate. There is the notion of a SessionManager which should be created only once in an application, because it is rather expensive to create. One SessionManager is needed for each SAP system which is used by the application.

The SessionManager is responsible for creating Sessions. A Session represents a connection to the SAP system. The first time we call a function module on a Session, Hibersap gets a connection from the underlying connection pool. When closing a session, the connection is returned to the pool. Therefore you have to take care always to close the session, preferably in a finally block, else the connection pool may get exhausted sooner or later.

The following function configures a Hibersap SessionManager. First, an instance of type AnnotationConfiguration has to be created for the named SessionManager, as specified in hibersap.xml. Finally, we build the SessionManager. In a real application this should be done once, reusing the SessionManager throughout the application's lifetime.

```
public SessionManager createSessionManager()
{
    AnnotationConfiguration configuration
        = new AnnotationConfiguration("A12");
    return configuration.buildSessionManager();
}
```

Listing 2.10: Creating the SessionManager

Now it is time to call the function module in SAP. After creating the SessionManager and opening a new Session, we create an instance of our BAPI Class, passing all parameters needed to execute the function as Constructor arguments. Then we simply call the execute() method on the Session, passing the BAPI class, which actually performs the call to SAP. Now the flightListBapi object is enriched with all the values which the function module returned and which we have mapped in our BAPI Class.

```
public void showFlightList()
{
    SessionManager sessionManager = createSessionManager();

    Session session = sessionManager.openSession();
    try
    {
        FlightListBapi flightList = new FlightListBapi( "DE", "Frankfurt",
                                                        "DE", "Berlin",
                                                        null, false, 10 );
        session.execute( flightList );
        showResult( flightList );
    }
    finally
    {
        session.close();
    }
}
```

Listing 2.11: Executing the function

To see the result of the function call, we simply print the BAPI class' fields to the console in the showResult() method:

```
private void showResult( FlightListBapi flightList )
{
    System.out.println( "AirlineId: " + flightList.getFromCountryKey() );
```

```
    System.out.println( "FromCity: " + flightList.getFromCity() );
    System.out.println( "ToCountryKey: " + flightList.getToCountryKey() );
    System.out.println( "ToCity: " + flightList.getToCity() );
    System.out.println( "AirlineCarrier: " + flightList
                                            .getAirlineCarrier() );
    System.out.println( "Afternoon: " + flightList.getAfternoon() );
    System.out.println( "MaxRead: " + flightList.getMaxRead() );

    System.out.println( "\nFlightData" );
    List<Flight> flights = flightList.getFlightList();
    for ( Flight flight : flights )
    {
        System.out.print( "\t" + flight.getAirportFrom() );
        System.out.print( "\t" + flight.getAirportTo() );
        System.out.print( "\t" + flight.getCarrierId() );
        System.out.print( "\t" + flight.getConnectionId() );
        System.out.print( "\t" + flight.getSeatsMax() );
        System.out.print( "\t" + flight.getSeatsOccupied() );
        System.out.println( "\t" + flight.getDepartureTime() );
    }

    System.out.println( "\nReturn" );
    BapiRet2 returnStruct = flightList.getReturnData();
    System.out.println( "\tMessage: " + returnStruct.getMessage() );
    System.out.println( "\tNumber: " + returnStruct.getNumber() );
    System.out.println( "\tType: " + returnStruct.getType() );
    System.out.println( "\tId: " + returnStruct.getId() );
}
```

Listing 2.12: Printing the results

Finally, create a main method that calls the showFlightList() method. Build the project with maven on the command-line using “mvn compile” and run the main class, or run it directly from your IDE.

In the example, we are looking for all flights from Frankfurt to Berlin. The result should look like follows, in this example, there were two flights found.

```
AirlineId: DE
FromCity: Frankfurt
ToCountryKey: DE
ToCity: Berlin
AirlineCarrier:
Afternoon: false
MaxRead: 10

FlightData
        FRA  SXF  LH  2402  220  191  Thu Dec 30 10:30:00 CET 2010
        FRA  SXF  LH  2402  220  207  Fri Dec 31 10:30:00 CET 2010

Return
        Message:
        Number: 000
```

```
        Type: S
        Id:
```

If there were no flights found, the return field will contain the following data returned by SAP:

```
Return
        Message: No corresponding flights found
        Number: 150
        Type: E
        Id: BC_BOR
```

To see further examples, you may check out the complete hibersap project from the subversion source code repository. See the information provided at the Hibersap project site http://hibersap.sourceforge.net. There you will also find examples of using Hibersap in a Java EE application server with a JCA resource adapter.

# Chapter 3

# Installation

## 3.1 The Hibersap libraries

The Hibersap framework is divided into the following libraries:

- hibersap-core-1.1.0Beta01
- hibersap-jco-1.1.0Beta01
- hibersap-jca-1.1.0Beta01

When using Hibersap in conjunction with a JCA compatible resource adapter, both the hibersap-core and the hibersap-jca jars must be on the application's classpath. When using Hibersap with the SAP Java Connector (JCo), the hibersap-core and the hibersap-jco libraries are needed.

## 3.2 Third party libraries

Hibersap has a few dependencies on third-party libraries, such as apache-commons or JAXB, which you have to add to your application's classpath. The easiest way to achieve this is to build your projects with Maven, see section 3.3. Depending on which interfacing technology is actually used, you also need the SAP Java Connector (JCo) or a JCA compatible resource adapter.

### 3.2.1 Hibersap dependencies

Please see the project's dependencies page under the following URL to get a list of libraries needed and add these to your application's classpath: http://hibersap.sourceforge.net/dependency-convergence.html

### 3.2.2 The SAP Java Connector (JCo)

The SAP JCo library is provided by SAP to their customers and comes with its own license. Please check the following URL for more information and downloads: http://service.sap.com/connectors. You need to have a user and password for this URL, which is free for SAP

customers. Valuable help regarding SAP technologies can also be found on the SAP Developer Network SDN: http://sdn.sap.com.

After downloading SAP Java Connector 3.0, extract the sapjco3.jar and the sapjco3 native library. The native library must be on your application's library path. You may copy it to a global lib path, to a special directory such as C:/WINDOWS/SYSTEM32 on Windows systems or to the execution directory of the application (e.g. the bin folder of your application server or the folder from which you run your application). Additionally, the JCo jar file must be on the application's classpath.

### 3.2.3 The SAP BAPI JCA Adapter

If you are using Hibersap with a resource adapter in a managed environment, the preferred choice is the SAP BAPI JCA Adapter. The project's web site can be found under http://sourceforge.net/projects/sapbapijcaadapt/. By the way, the SAP BAPI JCA Adapter seems to be the only Open Source adapter for SAP R/3.

For your application, you need to deploy the resource adapter to your application server. Since it uses the SAP Java Connector internally, the previous section also applies to this scenario, i.e. the SAP JCo has to be installed.

### 3.2.4 Using other JCA compatible Resource Adapters

The Hibersap JCA part was written using the SAP BAPI JCA Adapter and Hibersap was not yet tested with other resource adapters. Nevertheless, you are welcome to test Hibersap using other resource adapters and we would be glad to get your feedback.

Integrating a different Resource Adapter in combination with application-managed authentication includes writing a Resource Adapter specific implementation of the interface org.hibersap.execution.jca.cci.ConnectionSpecFactory. This is due to a limitation in the JCA specification which specifies the javax.resource.cci.ConnectionSpec as an empty interface, letting it up to the vendor-specific implementation to define which properties it contains. For a framework like Hibersap whose aim is to support different implementations this means that there must be a layer on top of the ConnectionSpec interface which makes it possible to adapt to different JCA implementations without having explicit dependencies to each of them. The ConnectionSpecFactory implementation for the SAP BAPI JCA Adapter uses reflection for setting and getting the ConnectionSpec's parameters in order to reduce (compile-time) dependencies to the resource adapter.

However, an implementation of the ConnectionSpecFactory is not needed if the application uses container-managed authentication, because in this case there is no need to use a ConnectionSpec.

## 3.3 Building Hibersap applications with Maven

If you are building your application with Maven, you need to specify the Hibersap repository for your module. Adding the following part to the repositories section of your pom.xml gives your project access to the libraries you need when using Hibersap.

```
<repositories>
```

```
    <repository>
        <id>repository.hibersap</id>
        <name>Hibersap Repository for Maven</name>
        <url>
            http://hibersap.svn.sourceforge.net/viewvc/hibersap/m2repo
        </url>
    </repository>
    ...
</repositories>
```

Listing 3.1: Maven repository for Hibersap

This is how you can specify the dependencies to the Hibersap libraries in your pom.xml. Depending on which interfacing technology you use, you generally need to depend from either hibersap-core and hibersap-jco or from hibersap-core and hibersap-jca.

```
<dependency>
    <groupId>org.hibersap</groupId>
    <artifactId>hibersap-core</artifactId>
    <version>1.1.0Beta01</version>
    <type>jar</type>
</dependency>
```

Listing 3.2: hibersap-core

```
<dependency>
    <groupId>org.hibersap</groupId>
    <artifactId>hibersap-jco</artifactId>
    <version>1.1.0Beta01</version>
    <type>jar</type>
</dependency>
```

Listing 3.3: hibersap-jco

```
<dependency>
    <groupId>org.hibersap</groupId>
    <artifactId>hibersap-jca</artifactId>
    <version>1.1.0Beta01</version>
    <type>jar</type>
</dependency>
```

Listing 3.4: hibersap-jca

Due to Maven's dependency management capabilities, all transient dependencies are automatically resolved so that there is no need to specify any of the above mentioned third-party libraries. However, since the SAP JCo and the SAP BAPI JCA Adapter can not be found in a (public) Maven repository due to its commercial license model, you have to manually add these libraries to your local repository.

The latest version of the SAP Java Connector can be downloaded from the SAP Service Marketplace at http://service.sap.com/connectors. There you will find different distribution packages for the supported platforms. To install the JCo jar to your local Maven repository, unpack the downloaded zip file (e.g. version 3.0.1) and issue the following command from your console.

```
mvn install:install-file -DgroupId=com.sap -DartifactId=sap-jco -Dversion
   =3.0.1 -Dpackaging=jar -Dfile=sapjco.jar
```

Listing 3.5: Installation of sap-jco.jar to local Maven repository

After installing the JCo library to your local repository or alternatively deploying it to your project's or company's repository, you can include the JCo lib in your project by declaring the following dependency.

```
<dependency>
    <groupId>com.sap</groupId>
    <artifactId>sap-jco</artifactId>
    <version>3.0.1</version>
</dependency>
```

Listing 3.6: Installation of sap-jco.jar to local Maven repository

The other important file from the JCo distribution is the JCo native library which you must put into your application's library path.

# Chapter 4

# Configuration

Hibersap configuration consists of a set of properties. There are two possibilities to configure Hibersap:

- XML file configuration
- Programmatic configuration

While bootstrapping, Hibersap first tries to configure itself looking for the /META-INF/hibersap.xml file in the classpath. When creating a SessionManager, configuration can be set or overwritten using programmatic configuration.

Using an XML file is the preferred way to configure Hibersap.

## 4.1 XML file configuration

The format of the XML configuration file is heavily inspired by JPA's persistence.xml. If you have to access different SAP systems, you can define more than one SessionManager by multiplying the session-manager XML element.

```
<?xml version="1.0" encoding="UTF-8"?>
<hibersap xmlns="urn:hibersap:hibersap-configuration:1.0">
  <session-manager name="A12">
    <context>org.hibersap.execution.jco.JCAContext</context>

    <jca-connection-factory>java:/eis/sap/A12</jca-connection-factory>
    <jca-connectionspec-factory>
      org.hibersap.execution.jca.cci.
        SapBapiJcaAdapterConnectionSpecFactory
    </jca-connectionspec-factory>

    <properties>
      <property name="jco.client.client" value="800" />
      <property name="jco.client.user" value="sapuser" />
      <property name="jco.client.passwd" value="password" />
      <property name="jco.client.lang" value="en" />
      <property name="jco.client.ashost" value="10.20.80.76" />
      <property name="jco.client.sysnr" value="00" />
```

```
    </properties>

    <annotatedClasses>
      <class>org.hibersap.examples.flightlist.FlightListBapi</class>
      <class>org.hibersap.examples.flightdetail.FlightDetailBapi</class>
    </annotatedClasses>

    <interceptorClasses>
      <class>com.mypackage.MyHibersapInterceptor</class>
    <interceptorClasses>
  </session-manager>
</hibersap>
```

Listing 4.1: hibersap.xml

Table 4.1 shows a list of the configuration parameters.

| Parameter | Description |
|---|---|
| context | The fully qualified class name of the Context class. This class must implement org.hibersap.session.Context and acts as a facade to the actually used interfacing technology. Existing implementations are org.hibersap.execution.jco.JCoContext for the SAP Java Connector (JCo) or org.hibersap.execution.jca.JCAContext for a JCA compatible Resource Adapter. (Default: JCoContext.) |
| jca-connection-factory | The JNDI name of the JCA Connection Factory. This parameter has to be specified if the application uses a JCA compatible resource adapter. The resource adapter has to be deployed on the application server independently from Hibersap and the application, defining a JNDI name. Hibersap will use this name to look up the resource adapter's ConnectionFactory. |
| jca-connectionspec-factory | The fully qualified class name of the ConnectionSpecFactory implementation used to get a ConnectionSpec object. A ConnectionSpec is is used to provide data such as a user name and password for the current session. An existing implementation is SapBapiJcaAdapterConnectionSpecFactory. |
| properties | Zero or more additional properties. These depend on the interfacing technology in use. For the SAP JCo, all the JCo-specific properties must be defined here. |
| annotatedClasses | All annotated BAPI classes which are used with the SessionManager must be listed here. |
| interceptorClasses | A list of the interceptor classes. These must implement org.hibersap.session.ExecutionInterceptor and are called before and after a SAP function module gets executed. |

Table 4.1: Hibersap configuration parameters

To build a SessionManager using the hibersap.xml file, you simply have to create an object of class org.hibersap.configuration.AnnotationConfiguration, specifying the SessionManager name as an argument. Note that there is also a default constructor for AnnotationConfiguration which can be used if there is only one configured SessionManager. Hibersap will issue a warning when there are more than one SessionManagers configured, but the no-args constructor is used.

```
AnnotationConfiguration configuration= new AnnotationConfiguration("A12");
SessionManager sessionManager = configuration.buildSessionManager();
```

## 4.2 Programmatic configuration

After creating a Configuration object which will be used to build the SessionManager, configuration can be set or overwritten programmatically.

The information from the XML file is internally converted into a Java data structure reflecting the structure of the XML document. All configuration classes have Java Bean style accessor methods (get.. and set..) for their fields.

For programmatic configuration you have to change or create SessionManagerConfig for each SessionManager you want to use. You can use method chaining to build the object. The following example creates a SessionManagerConfig object that is equal to the one created internally by the hibersap.xml example in section 4.1.

```
SessionManagerConfig cfg = new SessionManagerConfig( "A12" )
    .setContext( JCoContext.class.getName() )
    .setJcaConnectionFactory( "java:/eis/sap/A12" )
    .setJcaConnectionSpecFactory( "org.hibersap.execution.jca.cci."
                                  + "SapBapiJcaAdapterConnectionSpecFactory" )
    .setProperty( "jco.client.client", "800" )
    .setProperty( "jco.client.user", "sapuser" )
    .setProperty( "jco.client.passwd", "password" )
    .setProperty( "jco.client.lang", "en" )
    .setProperty( "jco.client.ashost", "10.20.80.76" )
    .setProperty( "jco.client.sysnr", "00" )
    .addAnnotatedClass( FlightListBapi.class )
    .addAnnotatedClass( FlightDetailBapi.class )
    .addInterceptor( MyHibersapInterceptor.class );

AnnotationConfiguration configuration = new AnnotationConfiguration(cfg);
SessionManager sessionManager = configuration.buildSessionManager();
```

Listing 4.2: Programmatic configuration

# Chapter 5

# Usage

## 5.1 Mapping Remote Function Modules to Java classes

The main task of Hibersap is to map Remote Function Modules (RFMs) which are defined in SAP R/3 to Java classes. To accomplish this, Hibersap provides a set of Java annotations that map Java classes and their fields to the elements of the function module's interface. This section provides detailed information on the Hibersap annotations and on how data types are converted.

For a complete example of how to define Hibersap classes and mappings, see chapter 2 or checkout the Hibersap examples code from the source code repository.

### 5.1.1 The annotations

Since Hibersap maps SAP Remote Function Modules to plain Java classes, an application must define one class for each function module that acts as a kind of proxy for the function module. This class is called a BAPI class and has to be annotated with the @Bapi[1] annotation[2]. The annotation's value property defines the name of the corresponding function module:

```
@Bapi("BAPI_SFLIGHT_GETLIST")
public class FlightListBapi {...}
```

A function module's interface usually has a set of parameters which are either of a scalar data type (such as a number, a text or a date) or of a complex data type (in ABAP called structure, a set of parameters of scalar data types). Scalar parameters can be mapped directly to a field of the BAPI class, whereas structures are mapped to complex Java types, i.e. classes with a set of fields. In Hibersap, these classes representing an ABAP structure are annotated with the @BapiStructure annotation. Since Structure class instances need to be created by Hibersap, they must have a default constructor.

```
@BapiStructure
public class BapiRet2 {...}
```

To map parameters of a function module to fields of Java classes, those fields have to be annotated with the @Parameter annotation. Here, Hibersap needs to know the name of the function module's parameter as well as its type using the annotation's value and type properties

[1] All Hibersap annotations and enumerations used by annotations are declared in the package org.hibersap.annotations.

[2] Actually, a BAPI is a special kind of a Remote Function Module but gives a more handy term.

respectively. The Enumeration ParameterType defines the types STRUCTURE and SIMPLE for parameters of a complex or scalar type:

```
@Export
@Parameter(value = "RETURN", type = ParameterType.STRUCTURE)
private BapiRet2 returnData;
```

Since the interface parameters of ABAP functions are divided into import, export and table parameters, Hibersap has to know where to find the corresponding parameters. The @Import, @Export and @Table annotations provide this information. However, since members of complex parameters should always be scalar parameters, we only have to use these annotations on "top-level" parameters, i.e. parameters of the BAPI class. Parameters of Structure classes just need the @Parameter annotation:

```
@BapiStructure
public class BapiRet2 {
    @Parameter("MESSAGE")
    private String message;
    ...
}
```

ABAP tables are lists of complex parameters. As such they are mapped to a java.util.Collection field. Hibersap again needs to know the name of the ABAP parameter as provided by the @Parameter annotation and its type as provided by the @Table annotation. A special characteristic of table parameters is that they can be used as import parameters as well as export parameters. Tables that import data to the function module are filled by the Java application, Hibersap will use the application-provided Collection to populate the table. Tables that export data from the function module will be read by Hibersap and copied to the corresponding Collection. Hibersap detects the type of the Collection's element by identifying its generic type, which means that it is crucial to define the Collection with a generic type (the class Flight in the example):

```
@Bapi("BAPI_SFLIGHT_GETLIST")
public class FlightListBapi {
    @Table
    @Parameter("FLIGHTLIST")
    private List<Flight> flightList;
}
```

The Java field representing a table need not be defined as a concrete implementation. When Hibersap creates a Collection, it creates a java.util.HashSet for fields defined as java.util.Set and a java.util.ArrayList for fields defined as java.util.Collection or java.util.List.

Table 5.1 shows an overview of the Hibersap annotations.

| Annotation | Description / Parameters |
|---|---|
| @Bapi | Maps a Java class to a Remote Function Module in SAP. **value** The name of the RFM, e.g. BAPI_FLIGHT_GETLIST |
| @BapiStructure | Maps the Java class to a structure in an RFM's interface |
| @Parameter | Maps a field to a parameter of the RFM's interface. **value** The parameter's name as defined by the RFM. **type** The parameter's type, either ParamType.SIMPLE for a scalar type or ParamType.STRUCTURE for complex types. Default: SIMPLE. |
| @Import | Defines a field annotated with @Parameter to be an import parameter. |
| @Export | Defines a field annotated with @Parameter to be an export parameter. |
| @Table | Defines a field annotated with @Parameter to be a table parameter. |
| @Convert | The parameter's data type will be converted by the given Converter, i.e. the Java field's data type may differ from the JCo data type conversion pattern. **converter** An implementation of org.hibersap.conversion. Converter. |
| @ThrowExceptionOnError | Generate a SapException when the function module returns errors in a RETURN structure or table. **returnStructure** Contains the path to the BAPI's return structure or table. The first element should be 'EXPORT' or 'TABLE' to indicate if the return structure is defined as an export or table parameter. The last element is the name of the return structure, usually 'RETURN'. **errorMessageTypes** The message types which Hibersap shall interpret as an error. In these cases an Exception will be thrown. The RETURN structure's field TYPE is compared to the message types. |

Table 5.1: Hibersap annotations: Overview

### 5.1.2 Type Conversion

The Java type of each simple field is related to the ABAP field's data type. Hibersap relies on the Java Connector's conversion scheme as shown in table 5.2.

| ABAP type | Description | Java type |
|---|---|---|
| C | Character | java.lang.String |
| N | Numerical character | java.lang.String |
| D | Date | java.lang.Date |
| T | Time | java.lang.Date |
| X | Byte field | byte[] |
| P | Packed number | java.lang.BigDecimal |
| I | 4-byte integer | int |
| F | Floating point number | double |
| STRING | Variable-length character | java.lang.String |
| XSTRING | Variable-length byte field | byte[] |

Table 5.2: SAP Java Connector type conversion

Hibersap allows to convert a parameter's data type to any Java type and vice versa. For example, ABAP does not have a boolean data type. Usually a boolean in ABAP is represented by a character field of length 1. A parameter is true if it equals to 'X', false if it is empty. With a Hibersap Converter it is possible to map an ABAP "boolean" parameter to a boolean field in Java.

You can use Hibersap Converters to do any kind of data type conversion. There are a few Converters defined in the package org.hibersap.conversion. However, it is easy to write your own converter by implementing the org.hibersap.conversion.Converter interface:

```
public interface Converter
{
    /**
     * Convert the SAP value, as it is returned by the underlying
     * interfacing technology (e.g. the SAP Java Connector, JCo)
     * to the Java data type of the corresponding BAPI class field.
     * Hibersap will call this method after calling the SAP function
     * and before setting the field in the Java class.
     *
     * @param sapValue The object which is returned by the SAP interface
     * @return The converted value
     * @throws ConversionException - if the value can not be converted
     */
    Object convertToJava( Object sapValue ) throws ConversionException;

    /**
     * Convert the Java value of the corresponding BAPI class field to the
     * data type as it is expected by the underlying interfacing
     * technology (e.g. the SAP Java Connector, JCo). Hibersap will call
     * this method before calling the SAP function.
     *
     * @param javaValue The value of the BAPI class field
     * @return The converted value
     * @throws ConversionException - if the value can not be converted
     */
    Object convertToSap( Object javaValue ) throws ConversionException;
```

```
}
```

Listing 5.1: The Converter interface

To use a converter, you simply have to annotate the filed in the BAPI or Structure class with Hibersap's @Convert annotation, specifying the Converter that should be called:

```
@Import
@Parameter ( "SHOW_DETAILS" )
@Convert(converter = BooleanConverter.class)
private final boolean showDetails;
```

Listing 5.2: Using a converter

## 5.2 Calling Function Modules

### 5.2.1 Configure Hibersap and build a SessionManager

Chapter 4 explains in detail how to configure Hibersap. The following code snippet assumes that there is the /META-INF/hibersap.xml configuration file defining a SessionManager named "A12" in the application's classpath and shows how to build a SessionManager:

```
AnnotationConfiguration configuration= new AnnotationConfiguration("A12");
SessionManager sessionManager = configuration.buildSessionManager();
```

Listing 5.3: Building the SessionManager

The SessionManager should be created only once in an application's lifetime. It depends on the kind of application how to store the SessionManager instances. E.g. in a web application it may be created and shut down in a ServletContextListener when starting the web application and put into the servlet context, in a stand-alone application it may be created and managed by a utility class with static accessor methods or – more advisable – by a dependency injection container such as Pico Container, Guice or Spring.

### 5.2.2 Calling a function in SAP

Calling a remote function module in SAP is as easy as opening a new Session via the SessionManager, creating the BAPI class with the required parameters, and calling the Session's execute() method with the Bapi class as an argument.

```
Session session = sessionManager.openSession();
try
{
    FlightListBapi flightList = new FlightListBapi( "DE", "Frankfurt",
                                                    "DE", "Berlin",
                                                    null, false, 10 );
    session.execute( flightList );
    showResult( flightList );
}
finally
{
    session.close();
```

```
}
```

Listing 5.4: Executing the BAPI function

After the call to session.execute(), the Bapi class is populated with the SAP function's return parameters as it is defined in the parameter mappings.

A physical connection to the SAP system is created during the first call to the Session's execute() method. It is crucial to close the Session after the last call to the execute() method. If Sessions are not closed, the connection pool may get exhausted. Therefore it is strongly recommended to close the Session in a finally block like in the above example. This way, even if one of the method calls in the try block throws a RuntimeException, the Session will still be closed.

If the application keeps sessions open for a long time, connections may also get shut down by the SAP system after some timeout period or the may get broken due to network related reasons. This means that Sessions should have a short lifetime (just like database sessions), usually a Session should be used for a single request or unit of work.

Hibersap Sessions are inexpensive to create, the Java Connector's connection pool takes care of an efficient connection management (if properly configured), and Hibersap will not akquire a SAP-connection before it is needed. Thus, Sessions should be used once and then discarded.

## 5.3 Transaction Management

SAP R/3 allows external callers of its function modules to control logical units of work (transactions). Calling applications may call a number of function modules and commit or rollback all changes made during the function calls. This can be done by calling special BAPIs named "BAPI_TRANSACTION_COMMIT" and "BAPI_TRANSACTION_ROLLBACK".[3]

Using Hibersap, it is very easy to start and end transactions, you do not have to map and call these BAPIs yourself. Depending on the chosen interfacing technology, you can either handle your transactions manually (JCo and JCA) or use Container Managed Transactions (JCA only).

### 5.3.1 Manual transaction handling

The Hibersap Session has a method beginTransaction() which must be called when starting the transaction. beginTransaction() returns a Transaction object on which you can later call the methods commit() or rollback() to tell SAP to commit or rollback all changes made during the transaction.

```
Session session = sessionManager.openSession();
Transaction transaction = null;
try
{
    transaction = session.beginTransaction();
    session.execute( bapiClass1 );
    ...
```

[3]Please note that not all Remote Function Modules and even not all BAPIs in SAP R/3 support this feature. Special care has to be taken in the ABAP code which should not call a COMMIT WORK itself. See SAP note 0131838, "Collective Note for BAPIs with the Commit Work Command" for a list of BAPIs that are not transaction enabled.

```
    session.execute( bapiClassN );
        transaction.commit();
}
catch (Exception e)
{
    if (transaction != null)
        transaction.rollback();
}
finally
{
    session.close();
}
```

Listing 5.5: Manual transaction handling

### 5.3.2 Container Managed Transactions

Applications running in an application server that use EJBs and a Java EE compatible Resource Adapter, may profit of the app server's capability of automatically handle transactions. In this case, the application code does not have to take care of starting, committing and rolling back transactions.

See section 5.5 for a detailed discussion on Container Managed Transactions.

## 5.4 Authentication

### 5.4.1 Using a central User

The most common way to interact with a SAP system in enterprise applications is to use a central SAP user that is shared by all users of the application. Using this kind of authentication is a matter of configuration. When using Hibersap with JCo, the user credentials are defined in the Hibersap configuration. When using Hibersap with JCA, the credentials are usually defined in the resource adapter.

### 5.4.2 Per-session authentication

If the application has to provide user credentials, e.g. when each application user log on to the SAP system using his own SAP account, you can specify logon credentials for each Hibersap session you create.

When obtaining a Session from the SessionManager, the method openSession(Credentials) must be used. The authentication information is specified in the Credentials object.

```
Credentials credentials = new Credentials()
    .setUser("atuttle")
    .setPassword("myPassw0rd")
    .setLanguage("EN");
Session session = sessionManager.openSession( credentials );
session.execute( bapiClass );
```

```
...
```

Listing 5.6: Component-managed authentication

The attributes defined in the Credentials object are simply passed on to SAP. Table 5.3 lists the possible fields including the corresponding JCo parameter as defined in interface DestinationDataProvider of the SAP Java Connector version 3.

| Credentials field | Description | JCo parameter |
|---|---|---|
| user | SAP logon user | JCO_USER |
| aliasUser | SAP logon user alias, can be used instead of logon user | JCO_ALIAS_USER |
| client | SAP client | JCO_CLIENT |
| language | Logon language ISO code | JCO_LANG |
| password | Logon password | JCO_PASSWD |
| ssoTicket | SAP Cookie Version 2 as logon ticket for SSO based authentication | JCO_MYSAPSSO2 |
| x509Certificate | X509 certificate for certificate based authentication | JCO_X509CERT |

Table 5.3: SAP Java Connector type conversion

### 5.4.3 Single sign-on

A special case of per-session authentication is the use of single sign-on. In a SAP environment it is common to use SAP Logon Tickets issued by SAP systems. In a web application / portal scenario the ticket is stored in a cookie named MYSAPSSO2 in the HTTP user session. After obtaining this ticket it has to be passed to the Hibersap session by setting the ssoTicket field of the Credentials object like described in the previous section 5.4.2.

## 5.5 Java EE application server integration

If developing applications which run inside a Java EE application server and make use of Enterprise Java Beans (EJB), it is strongly recommended using Hibersap in combination with a JCA compatible resource adapter. A resource adapter offers some important features which will be treated in this section.

### 5.5.1 Transaction Management

The application server's transaction manager is used to handle transactions. In combination with EJBs and Container Managed Transactions (CMT), the code does not need to care about transaction demarcation, because the container sets the transaction boundaries implicitly. This simplifies development and reduces the probability of errors. If using CMT, all the function calls to SAP which take place during an EJB method call are running in one transaction. If the

EJB method returns normally, the transaction manager commits the transaction. In case of an error, the transaction is automatically rolled back. Error here means that a RuntimeException is thrown or an application exception configured with the rollback=true attribute. (Note that Hibersap only throws runtime exceptions.)

Unfortunately, SAP R/3 does not provide the two-phase commit protocol (2PC) which is necessary for a resource to participate in a distributed transaction. The effect is that resource adapters for SAP R/3 support only local transactions, which is no problem if there is only one resource (the SAP R/3 system) enlisting in a transaction. If there are multiple resources participating in one distributed transaction (e.g. a SAP R/3 system, a database and/or a JMS system), the transaction manager starts a distributed transaction which requires 2PC aware resources as its participants. The good news is that some application servers (e.g. JBoss Application Server, Bea Weblogic or SAP WebAS) support a strategy called Last Resource Commit Optimization which makes it possible for a single non-2PC-aware resource to enlist in a distributed transaction. This is transparent for the programmer.

### 5.5.2 Security Management

With a resource adapter there are two methods to authenticate with the SAP R/3 system, container-managed and component-managed authentication. Using the former, the credentials (user name and password) are configured centrally in the resource adapter and the container uses these to automatically sign in whenever connecting to the SAP system. With component-managed authentication, an application passes credentials each time a connection is obtained from the resource adapter. This method is used when each user of the application signs in using its own SAP user, e.g. by entering the SAP user name and password in the application or when using single sign-on with logon tickets. Hibersap supports both authentication methods.

Container-managed authentication is simply a matter of configuration. The Resource Adapter will use the provided username/password combination (e.g. a central, technical SAP user).

Using component-managed authentication, the application has to provide authentication information itself. This may be e.g. a username/password combination entered by the application user, or a single sign-on login ticket obtained from the HTTP session. Section 5.4.2 shows how to use different authentication information on a per-session basis, section 5.4.3 shows how to use single sign-on tickets.

### 5.5.3 Configuration

Another advantage of using a resource adapter over using JCo directly is that a resource adapter can be deployed and configured independent of the applications that use it. Thus it is not necessary to reconfigure and redeploy all applications that connect to a SAP system whenever connection parameters (e.g. the SAP system's host name) change. Instead, only the resource adapter has to be reconfigured. Since the configuration of the resource adapter specifies all the necessary properties to access the SAP system, the Hibersap configuration only needs to specify the resource adapter's JNDI name that is used to look up the adapter's connection factory.