TUHH
Technische Universität Hamburg-Harburg

Implementation of Resource Adapter for SAP R/3 and generation of java-proxy class of BAPIs

Project Work

Submitted by:
Aravind Kumar Alagia Nambi
Master of Science in Information and Media Technologies
aravind.alagia@tu-harburg.de
Matriculation Number:
23307

Supervised by:

Prof. Dr. J. W. Schmidt
STS - TUHH

M.Sc. Miguel GARCIA
STS - TUHH

Hamburg, Germany
2004-04-22

Abstract

With the introduction of every new technology to the ever-growing Internet Technology, it becomes imperative for a Corporate to be updated to stay in business. Updating comes at a cost. Either train the existing personnel or hire experts. This requires both money and time. One such scenario is where a Corporate wants to use SAP as their Enterprise Information System.

Integrating the EIS with the existing Application Server will be the first step before using the EIS for the business application development. A generic way of integration with java-based application server can be achieved with the Java 2 Connector Architecture (JCA). A Resource Adapter can be developed, independent of the specifics of different java-based application servers like JBoss, BEA Weblogic, IBM Websphere, etc. The EIS specific details could also be hidden from the application developer to speed up the process of application development. Hence the needs to develop an Object-Oriented approach in accessing the data from the EIS. For the SAP system, java-proxy class of BAPIs can be generated to access the data from SAP R/3. My project work is concerned with developing a Resource Adapter for SAP system and generating java proxy class of BAPIs.

Acknowledgements

My special thanks go to Prof. Dr. Joachim W. Schmidt, research assistant MSc.Miguel Garcia at Software Systems Department of the Technical University of Hamburg-Harburg and
Dip-Ing.Krisztiàn Szitàs of CORYX Software GmbH for providing the opportunity to work on this student project, for their essential and kind advice, encouragement and guidance throughout this student project work.

Content

Figures

Acronyms

J2EE	Java 2 Enterprise Edition
JCA	Java Connector Architecture
RA	Resource Adapter
EIS	Enterprise Information System
EAI	Enterprise Application Integration
JCo	Java Connector
CCI	Common Client Interface
BO	Business Object
API	Application Programming Interface
BAPI	Business Application Programming Interface
JAAS	Java Authentication and Authorization Service
JTA	Java Transaction API
RFC	Remote Function Call
RFM	RFC-enabled Function Module

1 Introduction

1.1 *Motivation*

With more Enterprise businesses becoming internet driven, it becomes necessary to integrate the existing Enterprise Information System (EIS) with other systems. Enterprise Application Integration (EAI) eases the integration of disparate Enterprise Information Systems. Each EAI vendor created a proprietary resource adapter interface for its own EAI product, requiring a resource adapter to be developed for each EAI vendor and EIS combination. To make the process easier, it's necessary to have a standardized resource adapter that fits into all the EAI vendors or the application server that interacts with the EIS. As a required element of the Java 2 Enterprise Edition (J2EE), the Java Connector Architecture (JCA) provides a standardized means to integrate with Enterprise Information Systems (EIS). SAP R/3 system is one such EIS which requires a generic resource adapter that will plug-in to any J2EE based application server.

Though some java-based application servers like Web AS 6.4, BEA Weblogic, IBM Websphere, etc have their own resource adapter for SAP system, other java-based application servers like Jboss, JDMK, JOnAS, etc need a resource adapter to be plugged in to access SAP system.

Interaction with SAP system is achieved through the Business Application Programming Interfaces (BAPIs). But handling BAPIs is quite difficult and complex. Hence the objective of the studienarbeit is to develop a resource adapter for SAP R/3 system and to generate BAPIs as an object-oriented classes. This gives an object-oriented view of the BAPIs, hiding its intricate details and also the underlying SAP system.

2 Java Connector Architecture

Java Connector Architecture (JCA) provides standard connector architecture for connecting application servers and Enterprise Information Systems (EIS) such as

- Enterprise Resource Planning Systems - SAP, PeopleSoft, Baan
- Transaction Monitors - CICS, Tuxedo
- Database Management Systems - Oracle, Sybase
- Flat File System

With the J2EE Connector Architecture, the scope of integration of Java-based enterprise applications with EIS problem has been greatly reduced because the architecture defines a uniform way to integrate J2EE application servers with enterprise information systems. Under the connector architecture, EIS vendors no longer have to customize their product for interaction with each compliant J2EE application server. Similarly, application server vendors do not have to make modifications whenever they need connectivity to yet another EIS. Instead, application server vendors implement the connector architecture framework only once and EIS vendors develop one standard resource adapter based on this architecture. Under these circumstances, a compliant EIS can plug into any application server that supports the connector architecture. An application server, which conforms to this standard, can connect to any EIS that provides a standard resource adapter.

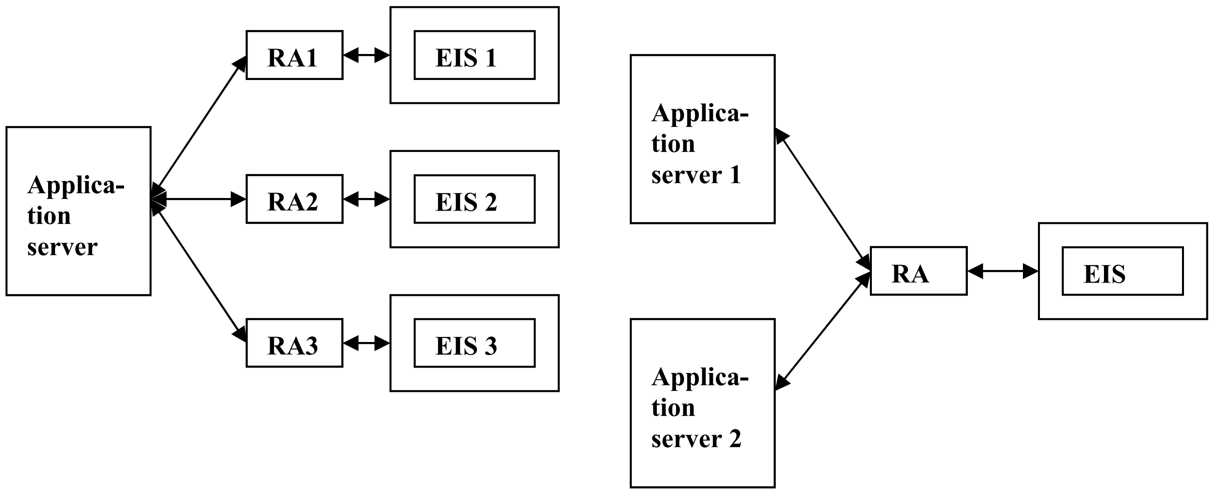


Figure 1*:* Integration between java-based application servers and EISs

There are two parts to this architecture: an EIS vendor-provided resource adapter and an application server that allows this resource adapter to be plugged in. This architecture defines a set of contracts that a resource adapter has to support to plug in to an application server. These contracts support bi-directional communication (outbound and inbound) between an application server and an EIS via a resource adapter. That is, the application server may use the resource adapter for outbound communication to the EIS, and it may also use the resource adapter for inbound communication from the EIS.

2.1 Resource Adapter Overview

The resource adapter plays a central role in the integration and connectivity between an EIS and an application server. It serves as the point of contact between application components, application servers and enterprise information systems. In a sense, a resource adaptor is similar to a JDBC driver for interfacing with a relational database. The difference is that a resource adaptor may connect to an EIS rather than a database. A resource adapter, along with the other components, must communicate with one another based on well-defined contracts that are specified by the J2EE Connector Architecture. The different components and their interactions are depicted in figure 2.

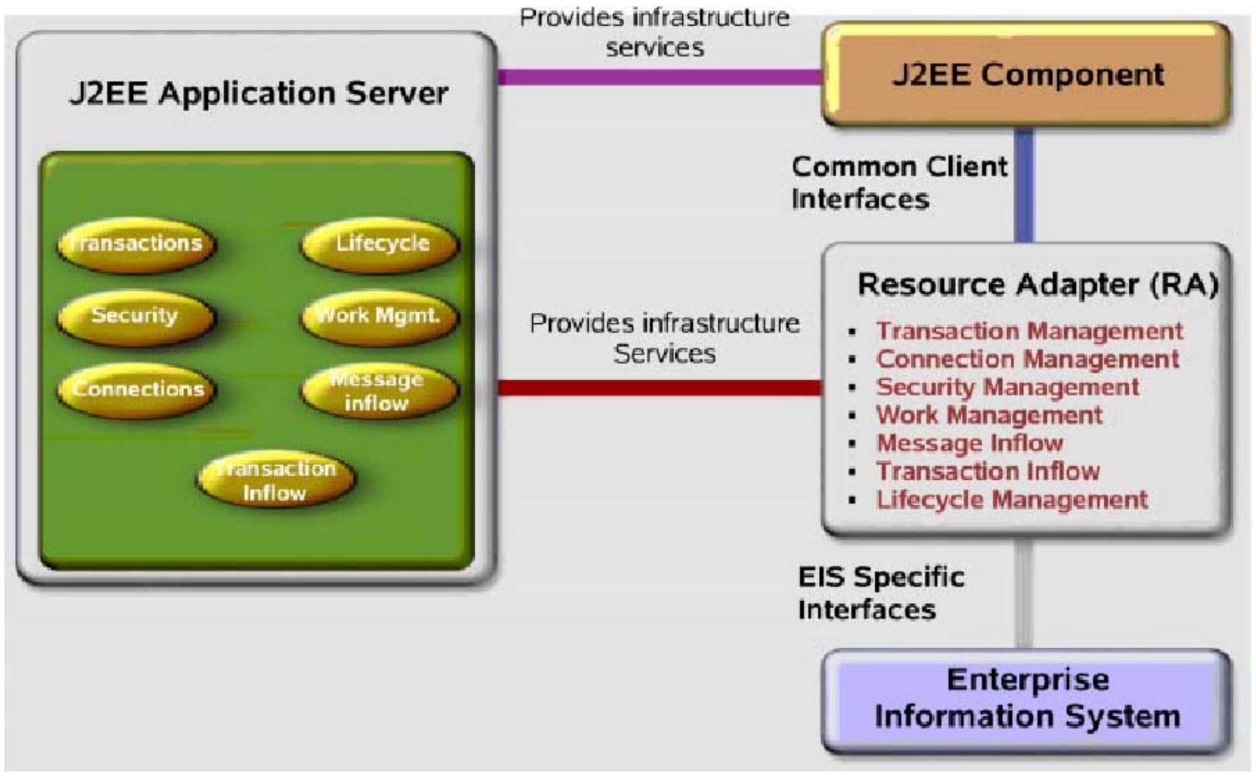


Figure 2 : Integration of Resource adapter with Application server, EIS and Application componenet

Programmatically, a resource adaptor must provide three sets of interfaces so that an application server or an adaptor client can interact with it:

- System-level Interface - The interface between the application server and resource adaptor.

- Application Interface - The interface for other applications on the application server (e.g., Web Applications, EJB components) to interact with the resource adaptor.

- Common Client Interface (CCI) - An general-purpose Application Interface defined in the JCA specification for interaction with the resource adaptor. It is defined for general data transfer, rather than for invoking specific functions in an EIS.

System contracts can be extended with the implemention of *ManagedConnection* and *LocalTransaction* or *XATransaction* interfaces. The application and CC interfaces are the handle for the application component to interact with EIS.

2.2 System-level Interface

The Java Connector Architecture defines the system-level interface that allows interactions between a J2EE application server and the resource adaptor. Application server provides system-level services to resource adaptors via the interface.

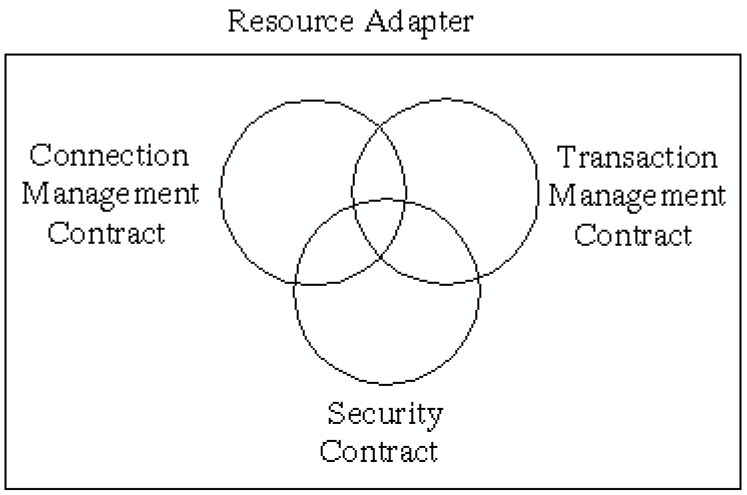

Resource Adapter
Connection Management Contract
Transaction Management Contract
Security Contract

Figure 3: Resource adapter and contracts

Apart from the three basic system-level contracts, as shown in the figure 3, that come with JCA 1.0 version, the JCA 1.5 has Work, Lifecycle, Message inflow and Transaction inflow contracts.

2.2.1 Connection Management

The connection management contract specifies an architected contract between an application server and a resource adapter. The goal of the connector architecture is to enable efficient, scalable, and extensible connection pooling mechanisms, not to specify a mechanism or implementation for connection pooling. The goal is accomplished by defining a standard contract for connection management with the providers of connections—that is, resource adapters. An application server should use the connection management contract to implement a connection pooling mechanism in its own implementation-specific way.

The connection management contract has been designed with the following goals:

- To provide a consistent application programming model for connection acquisition for both managed and non-managed (two-tier) applications.
- To enable a resource adapter to provide a connection factory and connection interfaces based on the CCI specific to the type of resource adapter and EIS.
- To provide a generic mechanism by which an application server can provide different services—transactions, security, advanced pooling, error tracing/logging—for its configured set of resource adapters.
- To provide support for connection pooling.

The application component that initiates a connection to the underlying EIS through the resource adapter could be managed or non-managed by the application server. In the managed scenario, the application server manages the connection through the connection contract with the resource adapter. In non-managed scenario, the application component uses the resource adapter directly.

2.2.2 Transaction Management

The transaction management contract controls transactions in two different ways.

First, the application server uses a transaction manager to support a transaction management infrastructure that enables an application component to perform transactional access across multiple EIS resource managers. The transaction manager manages transactions across multiple resource managers and supports propagation of the transaction context across distributed systems. The transaction manager supports a JTA XAResource-based transaction management contract with a resource adapter and its underlying resource manager. The ERP system supports JTA transactions by implementing a XAResource interface through its resource adapter. The TP system also implements a XAResource interface. This interface enables the two resource managers to participate in transactions that are coordinated by an external transaction manager. The transaction manager uses the XAResource interface to manage transactions across the two underlying resource managers.

Second, the transaction management contract can control transactions by creating *local transactions*. Local transactions are local in the sense that they exist only on a particular EIS resource. The transaction contract allows these transactions to be controlled, but they are related to any transaction that exists on the application server where the JCA resource adapter is running.

Also the resource adapter need not implement the transaction management contract. Making this optional allows for resource adapters in non-transaction resources.

2.2.3 Security Management

The security contract enables the application server to connect to an EIS system using security properties. The application server authenticates with the EIS system by using security properties composed of a principal (a user id) and credentials (a password, a certificate, and so on). An application server can employ container-managed sign-on method to authenticate to an EIS system (via a resource adapter).

With container-managed sign-on, the security credentials configure when the resource adapter is deployed on the application server. You can choose from several ways to configure security properties when using container-managed sign-on. First, with *Configured Identity,* all resource adapter connections use the same identity when connecting to the EIS system. Second, with *Principal Mapping,* the principal used when connecting to the EIS system is based on a combination of the current principal in the application server and the mapping which maps how the principal in the application server will map to a principal in the EIS system. The third is *Caller Impersonation,* where the principal used in the EIS system exactly matches the principal in the application server. The fourth is *Credentials Mapping,* which is similar to Caller Impersonation, except the type of credentials must be mapped from application server credentials to EIS credentials.

2.2.4 Lifecycle Management

The lifecycle management contract provides a way for an application server to manage the lifecycle of the resource adapter instance, through starting and stopping the instance. During deployment of the resource adapter or during application server startup, the application server starts the resource adapter by bootstrapping the adapter in its address space. Upon undeployment or application server shutdown, the application server shuts down the resource adapter by notifying it. This contract's functionality is provided through lifecycle management interfaces for both the application server and the resource adapter.

2.2.5 Work Management

The work management contract allows the resource adapter to do work by submitting it to an application server for execution. The resource adapter submits work to the application server, which dispatches a thread to run the work. Although this is not a required contract, there are a variety of reasons why it is advantageous for a resource adapter to allow the application server to handle its work. An application server handles thread management efficiently, and can even forbid a resource adapter from creating its own threads. Also, when the application server handles thread management, the resource adapter becomes more portable.

2.2.6 Message Inflow Management

The Message Inflow contract specifies a standard contract between an application server and a resource adapter; this allows the resource adapter to deliver messages to endpoints deployed within the application server, in a standardized way. The resource adapter delivers messages to these endpoints, either synchronously or asynchronously, without caring about messaging style, semantics, or infrastructure. The end result is that these endpoints, which are typically message-driven bean applications, are no longer restricted to receiving only JMS messages. Instead, through a Connector 1.5-compatible resource adapter, EIS vendors and messaging providers can communicate with these endpoints using any type of message, which includes but is not limited to their own proprietary brand.

2.2.7 Transaction Inflow Management

The Transaction Inflow Contract introduced in Connector 1.5 expands the support of transactions within the standard. Prior to Connector 1.5, J2EE applications could propagate transactions to an EIS, but an EIS could not pass transactions to the application server using a resource adapter. With the latest version of this architecture, however, transactions can now flow both ways. Due to this contract, a compliant resource adapter can now import transactions from an EIS and then propagate them to an application server. In addition to providing this transaction flow functionality, this contract also defines a mechanism for transaction completion and crash recovery flows from the EIS. The contract also guarantees that all ACID properties of the imported transaction are maintained. For a resource adapter within an n-tier environment to meet the requirements of this contract, it must implement methods for the *BootstrapContext* and *XATerminator* interfaces

2.3. Application Interface

Application interfaces of a resource adaptor is a set of interfaces for applications running on the application server (e.g. Web Applications, EJB Components, or Enterprise Applications) to access the resource adaptor. By using these interfaces, application can make function calls to the underlying EIS, or retrieve information from the EIS. The application interface of a resource adaptor is specific to the resource adaptor.

2.4 Common Client Interface (CCI)

The CCI defines a standard client API for application components. The CCI enables application components and Enterprise Application Integration (EAI) frameworks to drive interactions across heterogeneous EISs using a common client API. CCI is designed for general data-transfer, instead of accessing specific functions from a particular EIS. A resource adaptor is free to provide its own native application interfaces, or the Common Client Interface.

The CCI is set of low-level API. The API mainly defines a remote function-call interface that focuses on executing functions on an EIS, and retrieving the result thereafter. The CCI is designed to be independent of any EIS, but is capable of retrieving metadata about the EIS from a repository. The CCI is designed with the following goals:

- It defines a remote function-call interface that focuses on executing functions on an EIS and retrieving the results. The CCI can form a base level API for EIS access on which higher level functionality can be built.
- It is targeted primarily towards application development tools and EAI frameworks.
- Although it is simple, it has sufficient functionality and an extensible application programming model.
- It provides an API that both leverages and is consistent with various facilities defined by the J2SE and J2EE platforms.
- It is independent of a specific EIS. For example, it does not use data types specific to an EIS. However, the CCI can be capable of being driven by EIS-specific metadata from a repository.

The CCI APIs can be divided into four sections: First, the APIs related to establishing a connection to an EIS, also referred to as the *Connection Interfaces.* The second area of the CCI APIs covers command execution on an EIS, referred to as the *Interaction Interfaces.* Third is the *Record/ResultSet Interfaces,* which encapsulate the query results to an EIS. The fourth area, referred to as the *Metadata Interfaces,* allows EIS's metadata (the type of data) to be queried.

3 Implementation of Resource Adapter

The J2EE Connector Architecture specification defines interfaces, which implement the three contracts summarized in the previous chapter. Most of these interfaces are mandatory, in that they must be implemented by the adapter, while others do not need to be implemented. These non-mandatory interfaces are provided for the developers so they can maintain a consistent programming model if they choose. Vendors can define and implement their own interfaces, with no effect, as long as methods, which are required by the architecture, are provided in an EIS-specific manner. In the end, these vendor-defined interfaces will be very similar to the provided interfaces. All of the interfaces deliver a useful and standard framework that an EIS vendor can follow to develop a resource adapter. Descriptions of each interface, in relation to the contracts, are included below to provide a foundation for the design and implementation of a specific resource adapter within an n-tier environment.

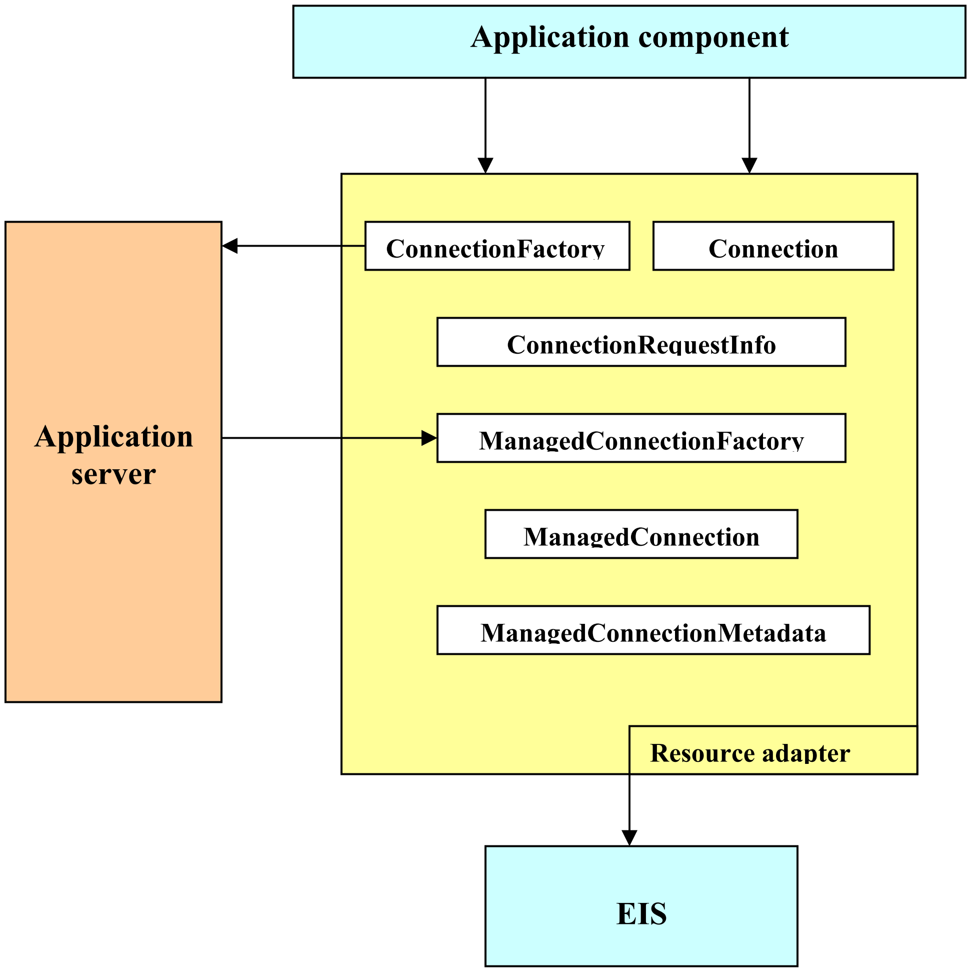


Figure 4: Connection management Interfaces

3.1 ConnectionFactory

ConnectionFactory is an interface that allows an application component to get a connection to an EIS instance. An application establishes a connection through the *getConnection* method. Then, this method must ask the application server to allocate a connection through the server's *ConnectionManager.allocateConnection* method. The resource adapter relinquishes this responsibility to the application server since the server is in charge of pooling connections and providing other services. Any of the resource adapter's specific request information must be passed to the *ConnectionManager.allocateConnection* method through the *ConnectionRequestInfo* parameter. The method *getConnection* can be overloaded if the EIS requires additional functionality.

The *ConnectionFactory* interface does not need to be implemented. An EIS vendor can choose to support the functionality of this interface in another way as long as the *getConnection* method is provided by the resource adapter. The *getConnection* method is required because it is how an application acquires a connection to the EIS.

3.2 Connection

The *Connection* interface provides an application with connectivity to an EIS. A *close* method must be provided so the application component can terminate the connection to the EIS.

This interface is not required by the connector architecture. The resource adapter just has to provide a *Connection* object with a corresponding close method to abide by this portion of the contract. The *Connection* object is necessary because that is how an application represents a connection to the EIS. A *close* method is also required because the application must have the ability to terminate the connection with the EIS.

3.3 ConnectionRequestInfo

ConnectionRequestInfo represents a resource adapter's request-specific data. It is passed to the application server's *ConnectionManager.allocateConnection*. The value null can be used if there is no data to pass. If a resource adapter chooses to implement this interface, then it must provide the *equals* and *hashcode* methods to aide the application server in connection pooling.

3.4 ManagedConnectionFactory

The interface *ManagedConnectionFactory* either matches an existing connection to the EIS with the incoming request or creates a new physical connection to the EIS. When the application server needs to allocate a connection to the EIS, it asks the resource adapter's *ManagedConnectionFactory* instance to get either an existing or a new connection. The configuration of the instance is facilitated by request-specific data. This interface, along with *ManagedConnection*, supports connection pooling. A resource adapter must provide an implementation of this interface. The code fragments below detail how this interface can be implemented. The implementation of this interface should have the following methods:

- **createConnectionFactory (**ConnectionManager connectionManager**)**

  This method should create a *ConnectionFactory* instance based on the *ConnectionManager* instance from the application server. This method should return the *ConnectionFactory* instance.

- **createManagedConnection (**javax.security.auth.Subject subject, ConnectionRequestInfo cxRequestInfo**)**

  This method should create a new physical connection to the EIS in an EIS specific way EIS using the security information in the parameter subject. The resource adapter must decipher how the security information is passed from the application server so that it can retrieve the necessary data. If the security information is null, then the resource manager must find it elsewhere. *ConnectionRequestInfo* object holds the request-specific data. The *subject* parameter is related to the security contract.

- **matchManagedConnections (**java.util.Set connectionSet, javax.security.auth.Subject subject, ConnecionRequestInfo cxRequestInfo**)**

  This method determines if there is an existing connection, from the parameter *ConnectionSet,* that can be used as the connection to the EIS. The check is based on criteria that are specific to the EIS. The method should return *null* if no match found. The *subject* parameter is related to the security contract.

3.5 ManagedConnection

The *ManagedConnection* interface provides an application-level connection handle from the EIS to the resource adapter's *ManagedConnection* instance. Communication between the two occurs through listeners and event notifications. Support for error logging and tracing must be present. Metadata about this instance and the EIS can be retrieved by invoking *getMetaData*, which returns information encapsulated in a *ManagedConnectionMetaData* instance. The interface also provides methods, like cleanup, to reinitialize the instance and free resources after communication ceases. The instance does not close the connection, however. This is handled by the application server so connection pooling can be utilized. An implementation of this interface is required by the specification. The implementation of this interface should have the following methods:

- **getConnection (**javax.security.auth.Subject subject, ConnectionRequestInfo cxRequestInfo**)**

  This method creates an application-level handle to EIS in an EIS specific way. This method also reauthenticates the connection to EIS using the security information provided.

- **destroy()**
  This method destroys a physical connection to the underlying EIS. This method is typically called by the application server.

- **addConnectionEventListener(**ConnectionEventListener listener**)**

  This method registers a connection event listener with the instance to get connection-related event notifications. An application server uses these event notifications to do its pool management, transaction management, and connection cleanup.

- **removeConnectionEventListener(**ConnectionEventListener listener**)**

  This method removes a connection event listener from the instance.

- **cleanup()**

  This method reinitializes the handles created by the instance before the handle is put back in the pool.

3.5.1 Transaction Management Interfaces

In addition to providing a connection handle to an EIS, a *ManagedConnection* instance gives access to two interfaces: *javax.transaction.xa.XAResource* and *javax.resource.spi.LocalTransaction*. The *XAResource* interface facilitates the transaction between a transaction manager and a particular EIS. The *LocalTransaction* interface manages local transactions.

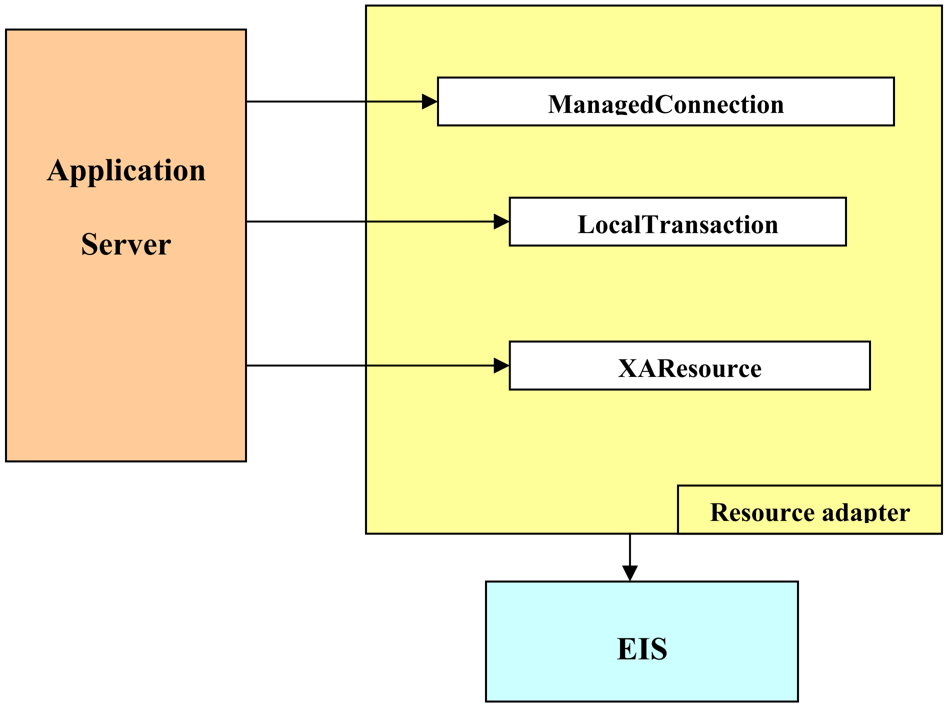


Figure 5: Transaction contract between Adapter and the Application Server

Transactions are setup before the *getConnection* method is invoked. Connection sharing is also enabled through the *associateConnection* method. The transaction management components of this interface must be implemented regardless of the level of transactional support provided by the resource adapter. The implementation, however, should throw appropriate exceptions if the type of transaction is not allowed by the resource adapter.

- **getXAResource ()**
  This method returns the *XAResource* instance associated with the *ManagedConnection* instance. If XA transactions are not supported, this method should throw an exception. Implementation is specific to EIS.

- **getLocalTransaction ()**

  This method returns the *LocalTransaction* instance associated with the *ManagedConnection* instance. If local transactions are not supported, this method should throw an exception. Implementation is specific to EIS.

3.5.1.1 XAResource

The *XAResource* interface is a Java mapping of the XA interface, which defines the contract between any resource adapter and a transaction manager in a distributed transaction processing environment. In reference to an enterprise information system's resource adapter, this interface enables the resource manager to participate in distributed transactions that are controlled and coordinated by an external transaction manager. The transaction manager uses the *XAResource* instance to manage the transaction. The *XAResource* interface must be implemented if XA transactions are supported by the resource adapter. In this case, certain requirements, like maintaining a 1-1 relationship between the *ManagedConnection* and *XAResource* instances and implementing one-phase commit, are mandated by the specification.

3.5.1.2 LocalTransaction

The LocalTransaction interface provides support for local transactions, which are managed and performed by the EIS. Information regarding the transaction is achieved through listeners and event notifications. If either local or both transactions are supported by the resource adapter, then it must supply this interface.

3.6 ManagedConnectionMetaData

ManagedConnectionMetaData facilitates the retrieval of metadata about a ManagedConnection instance and the particular EIS. The metadata must provide the enterprise information system's product name and version, the maximum number of concurrent connections that it can support, and the username associated with the connection. To comply with the connector architecture, an implementation of this interface must be included with the resource adapter.

3.7 Explanation at the Code level

3.7.1 System-level interface - Connection contract

The implementation of *ManagedConnectionFactory* interface should take care of the connection contract between the application server and the resource adapter. The application server passes the related set of connection in the pool to the resource adapter. If no relevant connection exists in the pool, a new connection is initiated by the reource adapter. This can be demontrated by the *createManagedConnection* and *matchManagedConnection* methods of *ManagedConnectionFactory* interface.

```
public ManagedConnection createManagedConnection (
    javax.security.auth.Subject subject,
    ConnectionRequestInfo cxRequestInfo)
    throws javax.resource.ResourceException {

        // Creates a new physical connection to the EIS in an EIS specific way.
}


public ManagedConnection matchManagedConnections (
    java.util.Set connectionSet,
    javax.security.auth.Subject subject,
    ConnecionRequestInfo cxRequestInfo)
    throws javax.resource.ResourceException {

        // Determines if there is an existing connection, from the parameter ConnectionSet, that  can be used as the
        //connection to the EIS.  The check is based on criteria that is specific to the EIS.

      // Must return null if a match does not exist.
}
```

3.7.2 System-level interface - Transaction contract

The implementation of *ManagedConnection* interface should take care of the transaction contract between the application server and the resource adapter. The transaction can be distributed or local. For distributed scenario, an instance of the implementation of the *XAResource* interface is handed to the application server. For local scenario, an instance of the implementation of the *LocalInterface* is handed to the application server.

```
public XAResource getXAResource ()
    throws javax.resource.ResourceException {

      // Returns the XAResource instance associated with the ManagedConnection instance.  If XA transactions are
      //not supported, this method should throw an exception.

    // Implementation of XAResource is specific to the EIS.
}
```

```
public LocalTransaction getLocalTransaction ()
    throws javax.resource.ResourceException {

    // Returns the LocalTransaction instance associated with the ManagedConnection instance. If local
    //transactions are not supported, this method should throw an exception.

    // Implementation is specific to the EIS.
}
```

3.7.3 System-level interface - Security contract

The security contract is adhered to by incorporating the J2EE Java Authentication and Authorization Service (JAAS) into the connection management interfaces. The three connection interfaces that must be modified to implement the security contract are *ConnectionFactory*, *ManagedConnectionFactory* and *ManagedConnection*.

- **ConnectionFactory**
  Security services from the application server are enabled by the resource adapter when the *ConnectionFactory* instance asks the application server to allocate a connection through the *ConnectionManager.allocateConnection* method. Depending on whether the application server or the application component manages the logon to the EIS, the resource adapter calls this method differently.
  - If the application server is in charge of logging on to the EIS, then the application component does not pass any security information to the resource adapter in the *getConnection* method. Since the resource adapter does not have this info, it does not forward any security information to the server.
  - If the application component is in charge of the logon to the EIS, however, it provides the necessary security information to the resource adapter through the *getConnection* method. Then, the resource adapter will handle the logon process through the *ManagedConnectionFactory* instance.

- **ManagedConnectionFactory**
  The *ManagedConnectionFactory* interface of the connection contract must be modified to support the security contract via the *createManagedConnection* method. The necessary modifications depend on whether the application server or the resource manager facilitates the logon to the EIS.
  - If the application server is in charge of this process, then it invokes *createManagedConnection* with the appropriate security information. This security data can be passed in a variety of ways established by JAAS. The instance retrieves the security information and then uses it to logon to the EIS.
  - If the resource adapter is in charge of the logon process, then the application server passes a null security parameter to the *createManagedConnection* method. In this case, the resource adapter looks for security information in the *ContextRequestInfo* instance. If it finds what it is looking for, then this information is used to connect to the EIS. Otherwise, the resource adapter uses the default security of the *ManagedConnectionFactory* instance.

3.8 Implementation for Non-managed Environment

The connection management contract enables a resource adapter to be used in a two-tier application directly from an application client. In a non-managed application scenario, the *ConnectionManager* implementation class may be provided either by a resource adapter as a default *ConnectionManager* implementation or by application developers. A default implementation of the *ConnectionManager* should be defined for a resource adapter only at development time. The default *ConnectionManager* instance interposes on the connection request and delegates the request to the *ManagedConnectionFactory* instance. The *ManagedConnectionFactory* creates a physical connection represented by a *ManagedConnection* instance to the underlying EIS. The *ConnectionManager* gets a connection handle from the ManagedConnection and returns it to the connection factory. The connection factory returns the connection handle to the application. A resource adapter supports interactions between its internal objects in an implementation-specific way. For example, a resource adapter can use the connection event listening mechanism as part of its *ManagedConnection* implementation for connection management. However, the resource adapter is not required to use the connection event mechanism to drive its internal interactions.

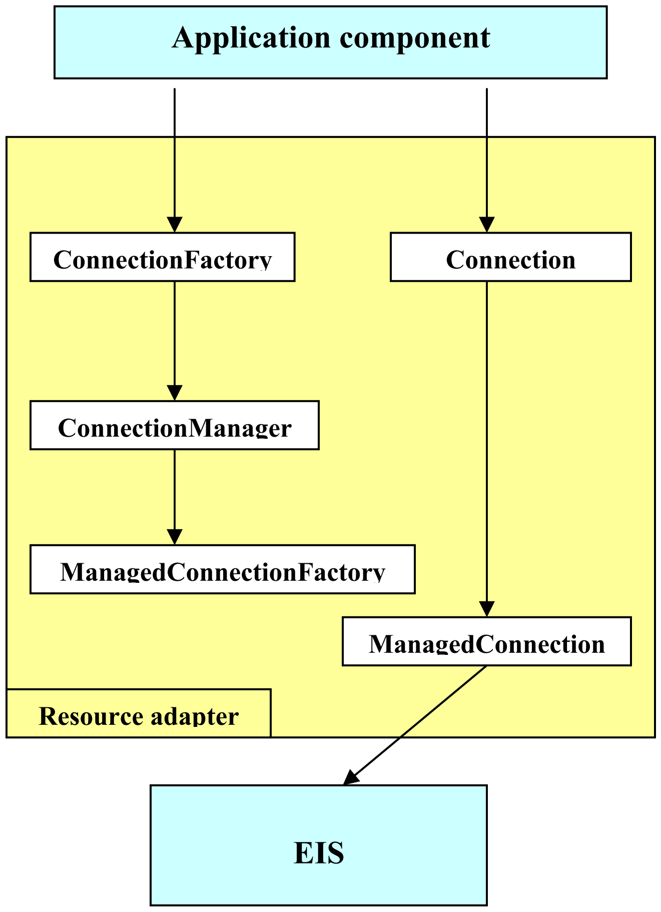


Figure 6: Non-managed scenario

4 Resource Adapter for SAP R/3 System

SAP R/3 system, one of the Enterprise Information Systems, can be seamlessly integrated with any J2EE based application servers through JCA Adapter. Adapter that adheres to the details that are described in the previous chapters will be sufficient to interface with the SAP system with the SAP's Java middleware, the SAP Java Connector (JCo). The other option of integrating could be through web services.

4.1 Web services vs Resource adapter for SAP

Boiled down to the simplest level, JCA and Web services are about connecting systems together. At that level, the two technologies can be seen as competing.

EISs such as SAP hold information that is critical to a business. Because of this, JCA needs to provide support for security and transactions. Additionally, a lot of data goes into and out of EISs, so performance is important. JCA is scalable to large numbers of EIS clients, and it provides connection management, which makes it possible for large numbers of clients to connect to the EIS efficiently.

Web services are about linking together arbitrary, heterogeneous systems that may be widely dispersed over the Internet. So, the first big difference is that Web services aren’t targeted specifically at EISs; any kind of software could be exposed as a Web service. Most services aren't going to be transactional or require the kind of security that EISs do, so initial Web services standards had no support for either security or transactions.

The second big difference between JCA and Web services is the desire for Web services to work with heterogeneous systems. To make this possible, Web services are very independent of technology. All that is required is HTTP over TCP/IP and XML. Thus, it doesn't matter what language the service is written in, or what platform it runs on. So, theoretically, a Java Web service running on UNIX will have no trouble working with a C# Web service on Windows 2003.

Considering the above pros and cons, JCA will be the right choice to make an EIS work with J2EE application and Web services will be the right choice to deal with a simpler piece of software out on the Internet.

4.2 Architecture of SAP JCo

SAP developed and released SAP Java Connector for enabling the java based applications to use the functionality or extend the functionality of SAP system. The SAP Java Connector (SAP JCo) is a toolkit that allows a Java application to communicate with any SAP System**.** JCo provides a set of Application Programming Interface that hides SAP related intricacies, data type conversion, connection pooling, etc and supports both inbound (Java to ABAP) and outbound (ABAP to Java) calls.

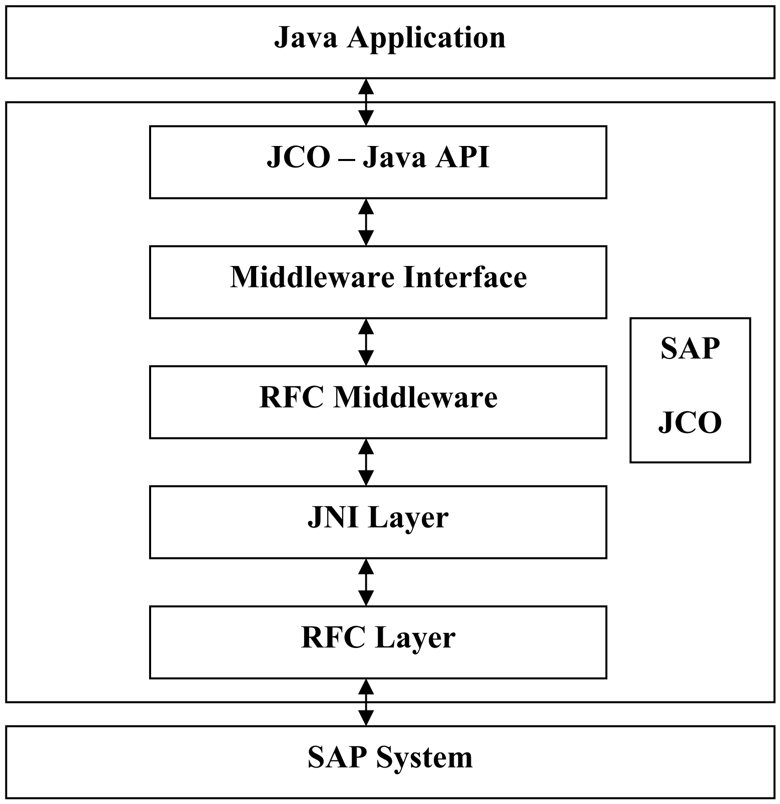


Figure 7: SAP Java Connector Architecture

The above figure shows the technical schema of data conversion in the SAP JCo. Starting from a Java application, a Java method is forwarded via the *JCo Java API* and an additional *Middleware Interface* to *RFC Middleware*, where it is converted to an RFC (ABAP) call using the *JNI* (Java Native Interface) layer, and sent to the SAP system. Using the same method in the other direction, an *RFC Call* is converted to Java and forwarded to the Java application:

The basis for all communication between SAP and external components (as well as for most communication between SAP components) is the Remote Function Call (RFC) protocol. RFC comes in three flavors. Most client programs want to use regular, Synchronous RFC (sRFC), but SAP also supports Transactional RFC (tRFC) and Queued RFC (qRFC). tRFC is used mainly to transfer ALE Intermediate Documents (IDocs). JCo supports all the three flavours of RFCs.

4.3 Resource Adapter and JCo

The resource adapter will communicate with the SAP system with the standard Java Connector APIs. The connection to the SAP system should be made at the *ManagedConnectionFactory.* The ManagedConnectionFactory is a singleton, only one object will be instantiated by the application server in case of the managed scenario. The connection could be pooled at the application server through the resource adapter.

4.3.1 Connecting to SAP

JCO.Client class provides methods to set up a communication channel with the SAP system. The burden of placing the connection pooling could be placed at the application server through the resource adapter or to the SAP System itself. The *JCO.createClient* static method makes a connection to the SAP.

```
JCO.Client con = JCO.createClient("001",           // SAP client
                                  "<userid>",      // userid
                                  "****",          // password
                                  "EN",            // language
                                  "<hostname>",    // application server host //name
                                  "00");           // system number
```

The connection specific parameters can be specified at the resource adapter descriptor file. This requires modification of the descriptor file for connections to different SAP and redeployment in the application server. To overcome this situation, *ConnectionRequestInfo* object can be used to get the user-specific information to make the connection to the SAP system. This object comes as an argument for the *createConnection* method of the *ManagedConnectionFactory*. The parameters could be retrieved from the *ConnectionRequestInfo* object.

4.3.2 Connection pooling

The other way of connecting to the SAP system will be by getting a connection from the connection pool maintained by the SAP. This moves the connection pooling responsibility from the J2EE based application server to the SAP system.

```
JCO.addClientPool (“<POOL_NAME>”,   // pool name
                   5,               // maximum number of connections
                   logonProperties); // properties

con =  JCO.getClient (“<POOL_NAME>”);
```

Each pool which has the connections with the same userid have unique name assigned by the user. The maximum number of connection in a pool could also be specified. *Properties* object of Java holds the connection parameters.

There are some issues to be considered when using connection pooling, be it maintained at the application server or the SAP system. Pooling is based on the userid of the connection parameter. When an open connection in a pool is reused, the previous owner of that connection is unknown. Some SAP functions maintain state. In order to isolate applications from one another, JCo calls the reset function in R/3 starting with release 4.0. This reset is performed automatically by JCo whenever a connection is returned to the pool. But R/3 3.1 does not have the reset function. So when run against a 3.1 system, either the existing connection with its state have to be reused or disconnect and reconnect again, which is more costly in terms of performance. JCo offers a variant of the *getClient()* method with a *boolean* parameter that allows to specify whether to reset (disconnect/reconnect for a 3.1 system) or not. The *getClient()* method that does not have any parameters will not disconnect/reconnect. This should only be used if no application using this pool ever calls any function in SAP that maintains state.

The pooling of connections could also be implemented at the resource adapter which has the connection contract with the application server. In effect, the application server pools the connections. This can be achieved by the singleton *ManagedConnectionFactory. matchManagedConnection* method gets set of connections pooled by the application. The set holds the similar connections that are pooled by the application server. The connections in the set could be checked against the userid and if it matches the connection could be returned, else a new connection is made. Here also the same limitation discussed above holds.

The connection handle thus received can be passed on to the application component as such through the *ConnectionFactory* object. In case of SAP, the *Common Client Interface* will not be helpful. Hence the connection in this case will be EIS specific. There is another possibility of defining the *Connection* interface other than just passing on the *connection* object. Replication of the BAPI could be implemented which accepts the input parameters and returns the result. In this case, the SAP system is completely hidden from the application.

4.3.3 Transaction

Having detailed about the connection, transaction is another issue that complements the connection. Most RFMs, including most BAPIs, are stateless. SAP does not remember anything between calls in the same session. Most updating BAPIs require an additional external commit call to actually cause any change on the SAP database to happen. All multiple update BAPI calls in the same session are combined into one Logical Unit of Work (LUW). A commit call at the end of the session which had one or more updating BAPIS. The is done via a call to *BapiService.TransactionCommit*, RFM name BAPI_TRANSACTION_COMMIT.

Till the recent release of SAP, two-phase commit is not supported. Hence the transaction can be maintained by the *LocalTransaction* interface, provided by the JCA specification. Implementation of the interface should hold the *begin*, *commit* and *rollback* methods. All the methods must fire the connection handle event to provide connection related event notification to the application server. The commit method should have the BAPI call BAPI_TRANSACTION_COMMIT. The rollback method should have the BAPI call BAPI_TRANSACTION_ROLLBACK. As the resource adapter maintains transaction contract with the application server, the transaction requirements of the application component can be extended by the application server and finally executed by the resource adapter.

4.3.4 Transaction in Non-managed environment

In this scenario, the application server is not involved. The connection pooling in this case should be transferred to the SAP system. The *LocalTransaction* handle should be given to the application which becomes fully responsible for maintaining the transactions. This is used by applications like applet, servlet, etc.

4.4 Business Application Programming Interface (BAPI)

BAPIs (Business Application Programming Interfaces) are the standardized methods that can be used to access the SAP Business Objects. BAPIs can be used from different programming environments and from all development platforms outside of the R/3 system. BAPIs are the handles for the underlying SAP Business Objects.

ABAP Function Modules can only be called from an external client if they are marked as Remote Function Call (RFC)-enabled. R/3 contains several thousands of such RFC-enabled Function Modules (RFMs). Amongst them are the BAPIs. BAPIs are RFMs that follow additional rules and are defined as object type methods in SAP's Business Object Repository (BOR).

4.4.1 BAPIS and JCo

The metadata of all RFMs must be available to JCo. This is accomplished by creating a *JCO.Repository* object. The actual metadata for the BAPIs could be retrieved dynamically from SAP at runtime.

```
JCO.Repository repo = new JCO.Repository("name", con);
```

The repository takes in two parameters: one being an arbitrary name and the other being a connection or connection pool to the SAP system. The userid used for the repository has to have sufficient authorizations in SAP for the metadata access to be possible.

```
IFunctionTemplate ft = repo.getFunctionTemplate(“BAPI_NAME”);
 Function fn = ft.getFunction();
```

Creating a *JCO.Function* object is a two-step process. First, a function template (interface *IFunctionTemplate*) should be created. A function template contains all the metadata (parameters and exceptions) for an RFM. JCo retrieves the metadata only once and caches it to optimize performance. The *getFunctionTemplate()* method of *JCO.Repository* is used to create the template. If null is returned, the RFM could not be found in SAP. From the template, a *JCO.Function* object (method *getFunction()*) can be created. A function object not only has metadata, but also the actual parameters for the execution of the RFM. The relationship between a function template and a function in JCo is similar to the one between a class and an object in Java. The code shown above encapsulates the creation of a function object. It is a good idea to create a fresh function object for each individual execution. This way, it can be ensured that the parameters do not contain any leftovers from the previous call, like table rows that should not be really sent to SAP.

BAPI has IMPORT/EXPORT parameters. The parameters could be simple types, structures or tables.

To access the import parameter list, *getImportParameterList()* can be used. The value of the scalar parameter is set using the *setValue()* method, passing the value as the first, and the name as the second argument. Many overloaded version of *setValue()* exist in JCo, in order to support all the data types. Again, JCo will do its best to convert any value that is passed to the data type appropriate for the field, and an exception is thrown if the conversion fails. The *setValue()* method is also available for *JCO.Structure* and *JCO.Table* so that the values of simple types, structure fields and fields in a table row can be set.

For simple types:

```
fn.getImportParameterList().setValue(“BAPI_TYPE_NAME”, “VALUE”);
```

For structures:

```
Structure st = fn.getImportParameterList().getStructure(“BAPI_STRUCTURE_NAME”);
st.setValue(“BAPI_TYPE_NAME”, “VALUE”);
```

For tables:

```
Table table =  fn.getTableParameterList().getTable(“TABLE_NAME”);
table.setValue(“BAPI_TYPE_NAME”, “VALUE”);
```

After setting the IMPORT parameters of the BAPI, the function should be executed.

```
fn.execute();
```

After the execution of the function, the EXPORT parameters can be retrieved by the calling the method *getExportParameterList()* of the Function *fn*. The simple types, structures and tables can be retrieved the same way as setting the IMPORT parameters with *get* instead of *set* in the methods.

4.5 Handling outbound call from SAP

To allow the SAP system to issue remote function calls (RFCs) or BAPIs to the adapter, an RFC destination on the SAP system must be created.

JCO.Server class of the JCo library encapsulates the basic JCO server functionality. Application programmers will extend this class and override the handleRequest() method to implement a specific server.

```
public Listener extends JCO.Server{
        // Constructor takes gateway host, service, program ID, and repository
        publicListener(String gwhost, String gwserv, String program_id, Irepository repos) {
                super(gwhost, gwserv,  program_id, repos);

                //gwhost - the gateway host
                //gwserv - the gateway service number
                //program_id - the program id
                //repos - the repository used by the server to lookup the definitions of an incoming function call
        }
        // Handles incoming requests
         protected void handleRequest(JCO.Function function) {
             try {
                  // Application specific processing goes here
                }catch(JCO.AbapException ex)
                    throw new JCO.AbapException("SYSTEM_FAILURE", ex.getMessage());
                }

        }
}
```

A server thread could be started with the *JCO.Server.start()* method which would listen for any outbound call from the SAP. With respect to the Message Inflow contract of the resource adapter, the *ResourceAdapter* interface supports the activation and deactivation of message endpoints that reside within the application server. The *endpointActivation* method is called by the application server to activate a message endpoint. This method accepts two parameters: a *MessageEndpointFactory* instance and a configured *ActivationSpec* instance.

The *MessageEndpointFactory* instance is used by the resource adapter to create message endpoint instances within the application server using the *createEndpoint* method, once the adapter has a message to propagate. Once the endpoint instances are created, the resource adapter can deliver messages to the instances, either serially or concurrently. Although the *MessageEndpointFactory* instance could be used to create an unlimited number of endpoint instances, the application server may restrict the number of endpoint creations.

In addition to a *MessageEndpointFactory* instance, a configured *ActivationSpec* instance is passed to the resource adapter by the application server and is used for configuration during endpoint activation. This instance stores crucial information regarding the type of endpoint listener that the resource adapter will send messages to. The *ActivationSpec* instance is configured by an endpoint deployer during endpoint deployment.

Once an endpoint is activated, it can receive messages from the message provider through the resource adapter. When the message provider sends a message, the resource adapter determines which message listener type the endpoint supports; it then forwards the appropriate message to the endpoint.

5 Generation of Java proxy classes of BAPIs

As seen in the preceding chapter, handling the BAPI calls are complicate. The complexity can be listed as follows:

- Handling the IMPORT/EXPORT properties are complex.
- Need to convert the data types from SAP system to Java standard. Though JCo helps in the conversion but need to map the data types.
- The structures could have nested structures which make the process further complex.
- The exception handling.
- Hand typing all the parameters as seen in the SAP system is quite cumbersome.

All those listed above make the business application developer's work more cumbersome. This brings up the necessity to develop Object-Oriented view of the Business Objects and its BAPIs in the SAP system to let the developer concentrate more on the business logic than on the intricacies involved in calling BAPIs. The java proxy class generation is done with the help of the Jakarte Velocity, a java-based template engine.

5.1 Velocity

Velocity is a Java-based template engine, a simple and powerful development tool that allows one to easily create and render documents that format and present the data. In our case, the data is the java proxy class. Velocity can be used either as a standalone utility for generating source code and reports, or as an integrated component of other systems. The Velocity Template Language (VTL) is used to define the structure of the proxy classes to be generated. Velocity templates could reference objects in Java code and could make method calls to retrieve data and eventually generate the classes.

The sequence of steps are as follows:

- Initialize Velocity.
- Create a Context object.
- Add the data objects to the Context.
- Choose a template.
- Merge the template and the data to produce the output.

5.2 Structure of the generated classes

A class that holds the list of all the Business Objects that is generated. The connection to the SAP system through the Resource Adapter is initiated from this class. The connections are expensive and to prevent an application from making many connections, this class called the *BapiRegistry* is made as singleton. The general structure of this class is as follows:

```
class BapiRegistry {
                    public static BapiRegistry getInstance() {
                    // provides the instance of this class
                     }

                    public BO1name getBO1Name() {
                    //create an instance of a Business Object (BO)
                              return new BO1(this);
                    }

                    public BO2name getBO2Name() {
                    //create an instance of a Business Object (BO)
                              return new BO2(this);
                    }

                    ....
                    ....

                    public SapConnection getConnection() {
                    //gives an existing connection within the
                    //context of this object or create a new connection
                    //SapConnection is the client view of the connection
                    //instance from the SAP Resource Adapter
                    }
}
```

The structure of the generated Business Object is as follows:

```
/**
 * Documentation of the Business Object as found in the SAP system
 */
Class BO {
          /**
           * Documentation of the BAPI as found in the SAP system
           */
          class BAPI {
                    private static final BAPI_NAME = „BAPI_NAME“; //Description with length restriction of the
                    //field and the optionality
                    class Argument {
                              private type SIMPLE_TYPES;
                              ...
                              ..
                              //Linked list to hold the records of the table
                              private LinkedList listTABLE_NAME;
                              // getter and setter method for the SIMPLE_TYPES
                              public type getSIMPLE_TYPE()  {
                                        return SIMPLE_TYPE;
                              }

                              public void setSIMPLE_TYPE(type SIMPLE_TYPE) {
                                        this.SIMPLE_TYPE = SIMPLE_TYPE;
                              }

                              class Structure {
                                        private type SIMPLE_TYPES;
                                        //getter and setter methods of the SIMPLE_TYPES;
                              }
                              class Table {
                                        private type SIMPLE_TYPES;
                              }
                    }
                    class Result {
                              //same as Argument;
                    }
          }

          public BO(BapiRegistry br) {
          // Constructor of the Business Object
          }

          public BO.BAPI.Result bAPI(BO.BAPI.Argument argument) {
          //open a connection to SAP system with the instance of BapiRegistry
          //all the necessary SAP JCo calls;
          //fill Result inner class of the BAPI with the EXPORT parameters
                    return result;
          }
}
```

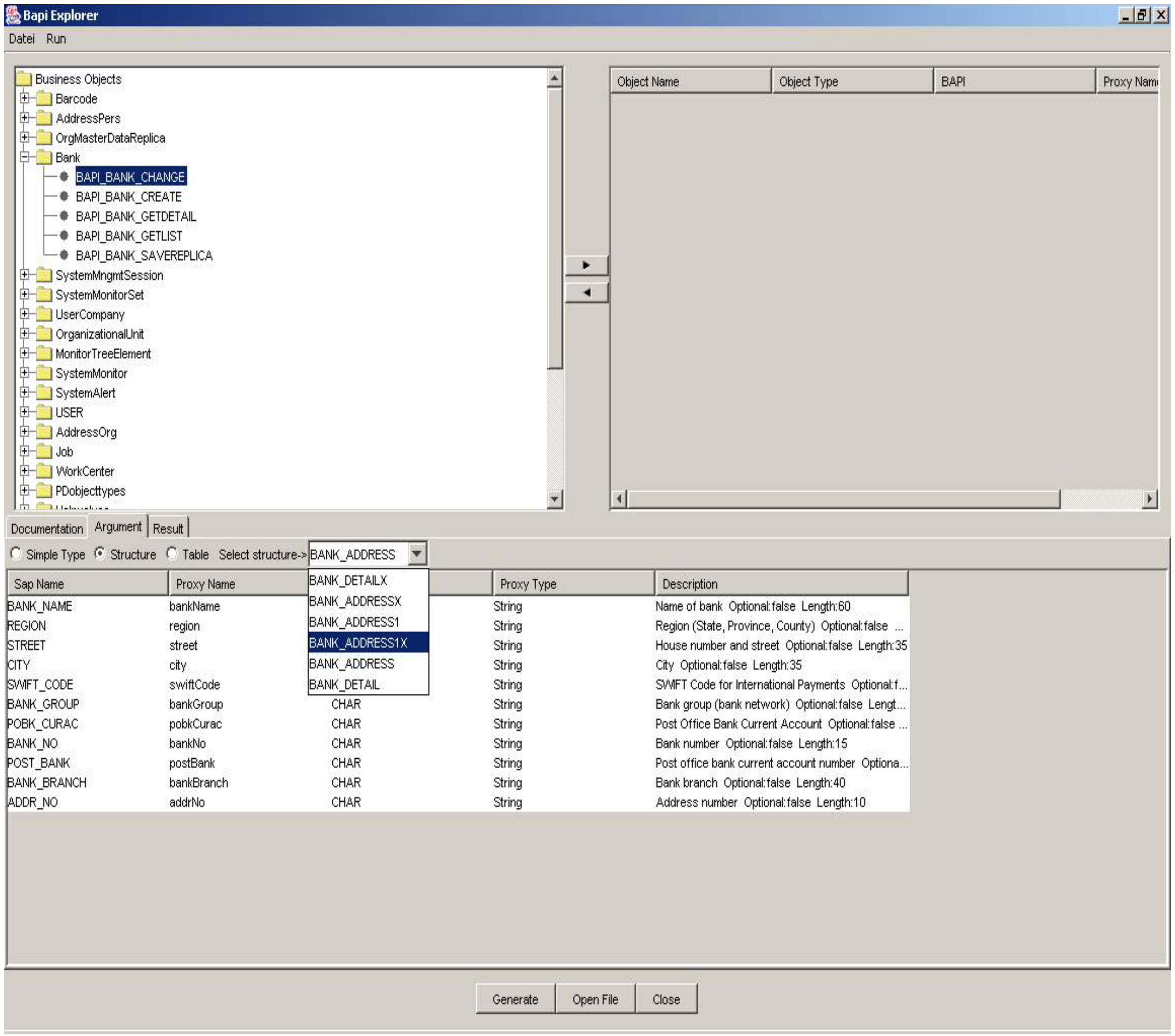
The generated Business Object has its BAPI as its inner classes. The BAPI inner class in turn has *Argument* and *Result* as its inner classes. This gives a complete Object-Oriented view of the Business Objects in the SAP system. The ease of use of these generated classes can be demonstrated with an example as follows:

```
BapiRegistry br = new BapiRegistry();
Bank bk = br.getBank();
Bank.GetDetail.Argument arg = bk.newGetDetailArgument();
arg.setBankCtry("DE");
arg.setBankKey("12345678");
Bank.GetDetail.Result result = bk.getDetail(arg);
```

The above example makes "BAPI_BANK_GETDETAIL" call with "BANKCTRY" and "BANKKEY" as its IMPORT parameter. The structure IMPORT also works the same by creating an instance of the *Arguments* inner class *Structure*. The BAPI call is executed by *getDetail* method which returns an instance of its inner class *result.* The parameters of the EXPORT parameters can be retrieved from the *Result* instance.

5.3 User Interface

A Swing based User Interface is created to allow the user to view all the BOs and BAPIs in the SAP system. The user interface is simple and easy to use. The main purpose of this UI is for the application developers who don't have much knowledge about the SAP system. The UI gives the documentation of the Business Objects and the BAPIs. It also lists the IMPORT/EXPORT parameters and its corresponding java-proxy names.



Simple Type [✓ Structure] Table Select structure-> BANK_ADDRESS

Sap Name	Proxy Name		Proxy Type	Description
BANK_NAME	bankName		String	Name of bank Optional:false Length:60
REGION	region		String	Region (State, Province, County) Optional:false ...
STREET	street		String	House number and street Optional:false Length:35
CITY	city		String	City Optional:false Length:35
SWIFT_CODE	swiftCode		String	SWIFT Code for International Payments Optional:f...
BANK_GROUP	bankGroup	CHAR	String	Bank group (bank network) Optional:false Lengt...
POBK_CURAC	pobkCurac	CHAR	String	Post Office Bank Current Account Optional:false ...
BANK_NO	bankNo	CHAR	String	Bank number Optional:false Length:15
POST_BANK	postBank	CHAR	String	Post office bank current account number Optiona...
BANK_BRANCH	bankBranch	CHAR	String	Bank branch Optional:false Length:40
ADDR_NO	addrNo	CHAR	String	Address number Optional:false Length:10

Figure 8: User Interface – BAPI Explorer

6 Conclusion

The integration of SAP R/3 system with java-based application server is achieved through the Resource Adapter that complies with the Java 2 Connection Architecture (JCA) specification. The Resource Adapter for the SAP R/3 system could be plugged into any java-based application with no modification. The Resource Adapter maintains the transaction and security issues related with the SAP system.

The resource adapter is the best choice when it comes to integration of SAP with J2EE application server. Application servers like Web AS 6.4 of SAP, BEA Weblogic 8.1 and IBM Websphere have their own resource adapter for SAP system. Resource adapters are to be written for other application servers like JBoss, JDMK, JonAS, etc. The following scenario A corporate’s enterprise application runs on Jboss and there is a necessity to integrate with a remote SAP. The remote machine on which SAP runs has Web AS 6.4 that already has SAP integrated with it through a resource adapter. Now the corporate has two options to interate its system with SAP. One: extend the contract the contract of the resource adapter on Web AS to Jboss (considering the corporate has the rights on Web AS) and Two: to implements its own resource adapter. The latter will be the best option as the former requires another resource adapter that acts as a link between WebAS and Jboss which is a performance overhead.

The generated Java proxy classes of the BAPI give an Object-Oriented view of the Business Objects of the SAP system. The generated classes hide the details of the BAPIs, JCo and the Resource Adapter. This allows the application developer to concentrate more on the business logic than on the details of the SAP system.

Bibliography

[JCA03]	*J2EE Connector Architecture Specification*, Version 1.5, Sun Microsystems, Inc., November 2003.
[J2EE03]	*Java 2 Platform Enterprise Edition Specification*, Version 1.4, Sun Microsystems, Inc., November 2003.
[JBOSS02]	*JBoss Administration and Development,* Scott Stark, Marc Fleury, The JBoss Group, Sams Publication, 2002.
[RA01]	*The J2EE Connector Architecture's Resource Adapter*, Jennifer Rodoni, Sun Microsystems, Inc., December 2001.
[RA03]	*What's New in the J2EE Connector Architecture 1.5,* Jennifer Rodoni, Sun Microsystems, Inc., March 2003.
[JCO02]	*Developing Applications with the SAP Java Connector (JCo)*, Thomas G. Schuessler, Arasoft GmbH, 2002.
[SAP02]	*Enterprise Java for SAP*, Austin Sincock, Apress Publications, 2002.
[JCA03]	*JCA – An Emerging Integration Technology*, Vijay, Mark, Steven, Avion, Inc., 2003.
[VELOCITY03]	*The Apache Jakarta Project: Velocity*, Version 1.4, http://jakarta.apache.org/velocity/, October 2003.
[SAP03]	*Komponenten für SAP mit Java,* Daniel Basler, Software and Support Verlag GmbH, 2003.
[JAAS01]	*Java Security,* Scott Oaks, O'Reilly and Associates, Inc., 2001.
[EAI01]	*Solutions for Enterprise Application Integration using Java and Web Services,* Rima Patel Sriganesh, Sun Microsystems Inc., 2002.
[PATTERN01]	Core J2EE Patterns, Deepak Alur, John Crupi, Dan Malks, Sun Microsystems press, A Prentice Hall Title, 2001.