



Siebel Retail Finance MCA Services Developer Guide

Version 2007

April 2007

Copyright © 2005, 2007, Oracle. All rights reserved.

The Programs (which include both the software and documentation) contain proprietary information; they are provided under a license agreement containing restrictions on use and disclosure and are also protected by copyright, patent, and other intellectual and industrial property laws. Reverse engineering, disassembly, or decompilation of the Programs, except to the extent required to obtain interoperability with other independently created software or as specified by law, is prohibited.

The information contained in this document is subject to change without notice. If you find any problems in the documentation, please report them to us in writing. This document is not warranted to be error-free. Except as may be expressly permitted in your license agreement for these Programs, no part of these Programs may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose.

PRODUCT MODULES AND OPTIONS. This guide contains descriptions of modules that are optional and for which you may not have purchased a license. Siebel's Sample Database also includes data related to these optional modules. As a result, your software implementation may differ from descriptions in this guide. To find out more about the modules your organization has purchased, see your corporate purchasing agent or your Siebel sales representative.

If the Programs are delivered to the United States Government or anyone licensing or using the Programs on behalf of the United States Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS. Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the Programs, including documentation and technical data, shall be subject to the licensing restrictions set forth in the applicable Oracle license agreement, and, to the extent applicable, the additional rights set forth in FAR 52.227-19, Commercial Computer Software--Restricted Rights (June 1987). Oracle USA, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

The Programs are not intended for use in any nuclear, aviation, mass transit, medical, or other inherently dangerous applications. It shall be the licensee's responsibility to take all appropriate fail-safe, backup, redundancy and other measures to ensure the safe use of such applications if the Programs are used for such purposes, and we disclaim liability for any damages caused by such use of the Programs.

Oracle, JD Edwards, PeopleSoft, and Siebel are registered trademarks of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

The Programs may provide links to Web sites and access to content, products, and services from third parties. Oracle is not responsible for the availability of, or any content provided on, third-party Web sites. You bear all risks associated with the use of such content. If you choose to purchase any products or services from a third party, the relationship is directly between you and the third party. Oracle is not responsible for: (a) the quality of third-party products or services; or (b) fulfilling any of the terms of the agreement with the third party, including delivery of products or services and warranty obligations related to purchased products or services. Oracle is not responsible for any loss or damage of any sort that you may incur from dealing with any third party.

Contents

1 What's New in This Release

2 MCA Services Overview

About MCA Services 13

About the Financial Component Framework 13

Transforming the DataPacket into the Protocol Format 14

Financial Component and Request_ID Mappings 14

Invoking a Financial Component 14

Sample Implementation of Client to Financial Component Communication 14

3 Channel Management

About Channel Management 17

RMI and HTTP Channel Management 18

RMI and HTTP Channel Client Class Descriptions 18

Communicating over HTTP 21

Thin clients using HTML Forms 21

Configuring Channel Management Properties 23

Developing Custom Channel Clients and Servers 25

Thin and Fat Client Examples 25

XML B2B Channel Management 32

Package: com.bankframe.ei.xml 33

Configuring XML B2B Channel Management 34

Developing Custom XML and XSL Codecs 35

DPTPCodec Transmission Format 36

XML and XSL Examples 37

Request Router Web Service 39

Request Router WSDL	40
Request Router Web Service Class Descriptions	42
Session Affinity	43
Configuring Session Affinity Properties	43
4 Financial Process Integration	
About Financial Process Integration	45
Overview of Interfacing with a Host System	45
Components of the Financial Process Integrator	46
Interaction of Financial Process Integrator Components	48
Financial Process Integrator Metadata	48
Request Transaction Fields	49
Example Transaction Request	51
Processing Host System Response	52
Response Metadata Mapping	52
Response Transaction Fields	54
Caching the Metadata (Transaction Fields)	54
TransactionField Interface	55
Response Mapping Sample Implementation	56
Support for Tiered Fields	57
Deeply Nested Cobol Copybooks	59
Mapping a Subset of Transaction Fields	61
Recurring Fields	61
Handling Error Conditions	62
Sample Error Condition Implementation	64
Transaction Field Naming	67
Mapping Entity Beans to Transactions	68
One Transaction to One Entity	68
One Transaction to Many Entities	68

Entity Bean Persistence and the FPI	69
com.bankframe.ejb.bmp	69
Writing a Persister	71
PersisterTxnMap	79
Configuring FPI Persister Settings	81
Financial Process Integrator Caching	81
Host Cache Examples	81
Deprecated FPI Caching Settings	82
Financial Process Integrator Engine	82
Financial Process Integrator Engine Interface	83
Transaction Request DataPacket	84
Transaction Request Processing Steps	84
Transaction Data-Format Class	85
TransactionHandlerUtils helper class	91
DataFormatUtils Helper Class	92
Transaction Route Entity Bean	93
Destination Entity Bean	94
Posting the Transaction Request data Object to the Host Connector	96
Configuring Financial Process Integrator Properties	96
Financial Process Integrator Testing using Test Servlet	97
EIS Connectors	101
MCA Services Connector Architecture	101
JCA Support	107
Store and Forward	110
Destination Entity Bean	111
DestinationEjbMap Entity Bean	111
Store and Forward Classes and Package Structure	112
Forcing the Host Online or Offline	120

Store and Forward Processing Exceptions	120
Configuring Store and Forward Properties	121
Configuring Deployment Descriptors for Store and Forward	121
Implementing Store and Forward	121
Teller Example of Store and Forward	125
About Branch Teller Offline Transaction Processing	129
Financial Process Integrator Sample Implementations	130
Extracting the Source Code for the FPI Examples	130
Launching the FPI Examples	130
The CustomerSearch Example	130
The AccountSearch Example	150
Handling Complex Amend and Find Operations	155
Handling Create and Remove Operations	156
Sample Data Formatter Implementation	158

5 Enterprise Services

About Enterprise Services	163
Security Provider Framework	164
Security Provider Framework Classes and Package Structure	165
Configuring the Security Provider	165
DefaultBankFrameSecurityProvider	166
NullBankFrameSecurityProvider	166
Implementing a Security Provider	167
User Authentication	167
The Siebel Retail Finance Logon Process	168
The Siebel Retail Finance Logoff Process	168
com.bankframe.services.authentication package	169
Implementing a Custom Authentication Mechanism	171
Registering Authentication Mechanisms with MCA Services	175

Implementing a Client Authentication Application	175
LDAP Authentication	178
RDBMS Authentication	178
Encrypting Sensitive Data	179
Session Management	180
Components of MCA Services Session Management	180
Session Management Use Cases	180
com.bankframe.services.sessionmgmt	181
Implementing a session management aware client application	183
Implementing a custom session management implementation	183
Configuring and Administering Session Management	183
Standard Session Management Implementations	184
Access Control	185
com.bankframe.services.accesscontrol	186
Implementing a custom access control mechanism	187
LDAP Access Control Mechanism	188
Configuring LDAP Access Control	188
Configuring Access Rights	189
EJB Access Control Implementation	191
User and Group Administration Session Beans	193
Routing	204
How MCA Services Routing Works	204
The com.bankframe.services.requestrouter package	205
The com.bankframe.services.route package	206
Route Administration Session Bean	208
Request Contexts	210
Request Context Example	213
Remote Notification	214

How Siebel Retail Finance Notification Works	214
Remote Notification API	218
The com.bankframe.services.notification.targetselection Package	220
Internationalization	221
Resource Bundles	221
Localized Messages in BankframeMessages.properties	222
MCA Internationalization Framework	222
Sample Localization Implementations	226
Logging	227
Logging Package and Classes	228
Using the Logging Service	229
The Logging context	232
Techniques for problem resolution using the logging framework	233
Configuring the Logging Service	234
Integrating with other Logging Frameworks	236
Logging Deprecations	237
Audit	237
Audit Classes and Package Structure	237
Configuring the Audit Service	238
Configuring Routes to the Audit Service	239
Calling the Audit Service from within custom code	239
Exceptions in the Audit Service	240
Timing Points	240
The com.bankframe.services.trace package	240
Configuring Timing Points	245
Mail Service	248
Classes and Package Structure	248
DataPacket Structure	249
Using the Mail Service	250

Ping Utility 251

- Classes and Package Structure 251

- DataPacket Structure 251

- Using the Ping Service 252

LDAP Connectivity 253

- com.bankframe.ei.ldap 253

- Sample Bean Managed LDAP based Entity Bean 256

- LDAP Connectivity Advanced Topics 261

Data Validation 264

- Classes and Package Structure 264

- Data Validator Sample Implementations 268

Peripherals Support 274

- MCA Device Base Classes 275

- MCA Device Implementations 278

- Implementing a New MCA Peripheral Device Object 292

Printing Framework 295

- com.bankframe.services.print 295

- Generating the Service 296

- Calling the Service from another Session 296

- Accelio Architecture 297

Caching Framework 299

- com.bankframe.services.cache 301

- Cache and Cache Index Interaction 308

- Cache and CachePolicy Interaction 309

- Creating persistent caches 310

- Configuring Cache Settings 310

Cache Extensibility 312

- Configuring Cache Extensibility 312

Dynamic Configuration 313

com.bankframe.services.resource 313

Using the Dynamic Configuration Framework 319

6 Administrating MCA Services

Launching the MCA Services Administration Application 323

Administering MCA Routes 323

Administrating MCA Sessions 324

Administrating MCA Users and Groups 324

Administrating MCA Users 324

Administrating MCA Groups 325

Using the MCA Monitor Servlet 326

7 Glossary: MCA Services

1

What's New in This Release

What's New in Siebel Retail Finance MCA Services Developer Guide, Version 2007

Table 1 lists changes in this version of the document to support version 2007 of the software.

Table 1. What's New in Siebel Retail Finance MCA Services Developer Guide, Version 2007

Topic	Description
Configuring Session Affinity Properties	Additional properties have been added to support URL rewriting and cookies.
Configuring Store and Forward Properties	Additional properties have been added for configuring the host status monitor.
Configuring Deployment Descriptors for Store and Forward	Instructions have been added for configuring application server properties to enable the host status monitor.
Configuring Cache Settings	This topic has been updated to include settings that need to be configured in BankframeResource.properties for each cache.
Cache Extensibility	Support for cache extensibility has been added to enable the plugging in of a proprietary cache implementation.

2

MCA Services Overview

This chapter provides an overview of Oracle's Siebel Retail Finance Multi-Channel Architecture Services (MCA Services), and data transmission between MCA Services and banking applications. It contains the following topics:

- About MCA Services
- About the Financial Component Framework

About MCA Services

Multi-Channel Architecture Services (MCA Services) is the framework on which all of Oracle's Siebel Retail Finance banking applications are built. MCA Services provides a mechanism for passing data between clients, financial components and host systems. MCA Services is comprised of the following functional areas:

- **Financial Component Framework.** A standardized architecture for extending and developing Siebel Retail Finance Financial Components.
- **Channel Management.** A framework for managing channel clients, servers and codecs.
- **Financial Process Integration.** A framework for communicating with legacy host systems.
- **Enterprise Services.** A set of services used by Financial Components, for example, Routing, User Authentication, Access Control and Internationalization.

About the Financial Component Framework

The Financial Component Framework provides a standard interface between Siebel Retail Finance clients and Financial Components. All data channeled between clients and Financial Components is transmitted in [DataPacket](#) format. An overview of the following Financial Component framework topics follows:

- Transforming the DataPacket into the Protocol Format
- Financial Component and Request_ID Mappings
- Invoking a Financial Component
- Sample Implementation of Client to Financial Component Communication

Transforming the DataPacket into the Protocol Format

Transforming a `DataPacket` to a protocol format, and vice versa, is achieved using a Communications Manager (CommsManager). MCA Services provides a number of CommsManagers that can transform `DataPackets` to and from different protocols, for example the `EHTTCommsManager` can transform `DataPackets` into HTTP Requests. When a client needs to send a `DataPacket` to MCA Services over HTTP it uses the `EHTTCommsManager` class to send the `DataPackets` as HTTP requests to MCA Services. MCA Services uses another CommsManager, `EHTTServletCommsManager`, to transform the HTTP requests back into `DataPackets`.

Financial Component and Request_ID Mappings

One of the `DataPacket` key values defined by MCA Services is the `REQUEST_ID` key. This key contains a five-digit number. This five-digit number is used to identify which Financial Component a `DataPacket` should be sent to. Each Financial Component has a `REQUEST_ID` associated with it. When a client wants to send a `DataPacket` to a Financial Component, it must put the `REQUEST_ID` associated with the Financial Component in the `DataPacket`.

When MCA Services receives the `DataPacket` from the client it examines the `DataPacket` to see what `REQUEST_ID` is specified. MCA Services then looks up a mapping of `REQUEST_ID`s to Financial Component names, finds the specified `REQUEST_ID`, and invokes the associated Financial Component.

Invoking a Financial Component

The Financial Component is an EJB Session bean. Every EJB has a unique JNDI (Java Naming and Directory Interface) Name. MCA Services maintains a mapping of `REQUEST_ID`s to JNDI names. When MCA Services has discovered a Financial Component's JNDI name, it asks the EJB Server to create an instance of the Financial Components. All Financial Components must have a method called `processDataPacket()`. MCA Services invokes this method, passing it the `DataPacket` received from the client.

Sample Implementation of Client to Financial Component Communication

This example illustrates how a credit transfer is implemented using MCA Services. The following assumptions are made:

- The client is a Java application and communicates with MCA Services over HTTP.
- The Financial Component that implements the credit transfer is called `CreditTransferBean`. It has the JNDI name: `eontec.bankframe.CreditTransferBean`.
- The `CreditTransferBean` expects a `DataPacket` with the following keys: `DATA_PACKET_NAME`, `FROM_ACCOUNT`, `TO_ACCOUNT` and `AMOUNT`. The `DATA_PACKET_NAME` must have a value of `CREDIT TRANSFER`.
- The user inputs values for `FROM_ACCOUNT`, `TO_ACCOUNT`, and `AMOUNT` on the application user interface.

Client Creates DataPacket

The client application creates a [DataPacket](#) with the following keys: NAME, REQUEST_ID, FROM_ACCOUNT, TO_ACCOUNT and AMOUNT.

Client Sends DataPacket to MCA Services

The client must use the [EHTTPCommsManager](#) class to send the [DataPacket](#) to MCA Services via a HTTP request.

MCA Services Converts the HTTP Request Back to a DataPacket

MCA Services uses the [EHTTPServletCommsManager](#) class to convert the HTTP request back to a [DataPacket](#).

MCA Services Determines which Financial Component to Invoke

MCA Services checks the [REQUEST_ID](#) key in the [DataPacket](#). It looks up the mapping of [REQUEST_IDS](#) to JNDI names, and determines that the [DataPacket](#) should be sent to the EJB named `'eontec.bankframe.CreditTransfer'`.

MCA Services Passes the DataPacket to the Financial Component

MCA Services asks the EJB Container to create an instance of the bean named `'eontec.bankframe.CreditTransfer'`, that is, [CreditTransferBean](#). When the instance is created MCA Services invokes [CreditTransferBean](#)'s `processDataPacket()` method, passing it the [DataPacket](#) from the client.

CreditTransferBean Processes the DataPacket and Returns its Response Data

[CreditTransferBean](#) parses the information in the [DataPacket](#) and carries out the credit transfer. It returns a response [DataPacket](#) confirming the transaction was carried out and containing the new balance on the account the money was transferred from.

MCA Services Passes the Response Data back to the Client

MCA Services uses the [EHTTPServletCommsManager](#) to send the response back to the client as a HTTP response.

The Client Converts the HTTP Response back into DataPackets

The client uses [EHTTPCommsManager](#) to convert the HTTP Response into a Vector of [DataPackets](#). In this case the Vector contains a single [DataPacket](#) with the information returned from the Financial Component.

3 Channel Management

This chapter covers the MCA Channel Management framework and includes the following topics:

- RMI and HTTP Channel Management
- XML B2B Channel Management
- Request Router Web Service
- Session Affinity

About Channel Management

MCA provides a variety of channel clients that communicate over a variety of protocols. Requests can be comprised of multiple [DataPackets](#). Most Siebel Retail Finance clients use channel clients, which in turn communicate with channel servers that act as gateways to Financial Components.

Not all Siebel Retail Finance clients need to use a Channel client to communicate with MCA, for example, browser clients send request data in a HTTP Post or Get request.

The Channel Management Framework is composed of the following constituent parts:

DataPackets

Data is passed to, from, and within MCA Services in [Datapackets](#). [DataPackets](#) are hashtables that use a simple key-object mapping. There are a number of standard key names, such as [REQUEST_ID](#) and [DATA PACKET NAME](#), that must be included in all [DataPackets](#) in order for them to be processed by MCA.

All data that is passed between channel clients and MCA is encoded as a [Vector](#) of [DataPackets](#). This provides a standard format for all data used within MCA. All responses from MCA are also encoded as a [Vector](#) of [DataPackets](#). This helps provide a standard view of MCA to all Siebel Retail Finance clients regardless of their type.

Channel Clients

A channel client is a class provided by MCA that is used by any fat client wishing to send data to, and receive data from MCA. It deals with all communication issues involved in sending a request to MCA and receiving the corresponding response. This ensures that the view provided by all channel clients to Siebel Retail Finance clients is consistent. However the data sent by each channel client to MCA will depend entirely on the network and network protocol over which the data is being sent. Therefore each channel client must be able to accept requests in a standard format ([DataPackets](#)) and convert this to a channel (network) specific format for transmission.

Channel Servers

The main function of a channel server is to accept requests from a channel client, convert this request to a `DataPacket` and pass the `DataPacket` to the `RequestRouter`. The channel server will also appropriately encode the response from the Financial Component and return this to the calling channel client. This means that for most channel clients there will be a corresponding specific channel server that will understand the network specific format of the request and build a standard `DataPacket` request from this.

Codecs

Codecs are used to encode data that is sent between some channel clients and channel servers. Siebel Retail Finance client requests consist of one or more `DataPacket` objects. However `DataPacket` objects usually need to be converted to a specific form before they can be sent over a network connection. This is the job of the codec. It will convert a `DataPacket` representation of a request to a format that can be sent over the network. codecs must also be able to rebuild the original `DataPacket` request from the encoded request to allow the channel server to process it.

RMI and HTTP Channel Management

MCA Services provides support for RMI and HTTP clients. The `com.bankframe.ei.channel.client` package provides two mechanisms for passing `DataPackets` over http connections, one to be used with thin clients, the other with fat clients.

RMI and HTTP Channel Client Class Descriptions

Package: `com.bankframe.ei.channel.client`

This package contains the classes that are used by Siebel Retail Finance clients to communicate with MCA.

ChannelClient Interface

All channel clients must implement this interface. All implementing classes must override the `send(Vector)` method.

ChannelClientFactory Class

This class uses the factory pattern to generate `com.bankframe.channel.ChannelClient` instances based on properties set in the `BankframeResource.properties` file. The purpose of this is to remove the need for code changes should a Siebel Retail Finance client wish to change the way (protocol) by which it transmits data. By using this factory pattern all the Siebel Retail Finance client needs to do is change the values within the properties file and the `ChannelClientFactory` will supply the appropriate class for the new transmission protocol. The factory can also be configured to return the same instance of a `ChannelClient`, or a new instance each time, by setting the `enforce.singleton` property in `BankframeResource.properties`. By default the `getChannelClient()` method will lookup `channel.client` property key. However, another property key can be specified through `getChannelClient(String clientName)`.

HttpClient

This is a client for transmitting [DataPackets](#) over any HTTP connection. Fat clients communicating over HTTP should use this client. This client has a number of properties that must be set in the [BankframeResource.properties](#) file. Settings include what codec class to use to encode and decode a vector of DataPackets. The HttpClient can also add values from the first DataPacket as request properties to the http connection. This is all configurable in the properties file.

HttpsClient

This is a client for transmitting [DataPackets](#) over a secure HTTPS connection using SSL. Any application, that requires the transfer of information over a secure connection to a server should use this client. Before a secure connection can be made the client and server must have a truststore and also a keystore created. The truststore contains trusted certificates and the keystore holds the public-private keys used in SSL. This client has a number of properties that must be set in the [BankframeResource.properties](#) file.

RmiIiopClient

This class is used to call the [RequestRouter](#) directly using an RMI call. [RequestRouter](#) stub classes are needed by the Siebel Retail Finance client when using this class.

Package: com.bankframe.ei.channel.server.

This package contains all the classes required to listen for and process incoming Siebel Retail Finance client requests. Each class will deal with a single combination of transmission protocol, for example, HTTP or RMI, and data format.

HttpServer

This is a servlet that listens for HTTP requests from any [HttpClient](#). The server will decode the incoming requests to [DataPackets](#) and pass them onto the relevant Financial Components. It will then take the response and appropriately encode this response for transmission back to the [HttpClient](#). Again it uses settings in the [BankframeResource.properties](#) file to deduce the format of the request.

JspHttpServer

This is the class that processes requests that originate from JSPs. JSPs are generally used when user input is from HTML forms. MCA provides a mechanism by which Siebel Retail Finance client developers can encode multiple [DataPackets](#) within a single HTTP post request. The [JSPHttpServer](#) will process requests from the JSP bean class. It will interpret the form field names and data to produce request [DataPacket](#)(s). The response received from the Financial Component is returned to the JSP bean code, where it is handled in the [handleResponse\(\)](#) method.

HttpBoomerangServer

This is a test servlet that extends HttpServer. Rather than routing the vector of [DataPackets](#) found in the request, it returns the vector as a response. It is useful for testing channel client and codec configuration. The vector sent and the vector received by the client should be the same. The servlet can be used by setting the [channel.http.client.url](#) property to the URL of the deployed servlet.

Package: com.bankframe.ei.channel.codec

This package contains classes that implement codecs (coders/decoders).

Codec

All codecs implement this interface. This defines the method signatures for sending and receiving [DataPackets](#). All codecs will turn a vector of [DataPackets](#) into a string representation.

DPTPCodec

DPTP stands for 'DataPacket Transmission Protocol'. It is used for encoding character data. This codec converts a [Vector](#) of [DataPackets](#) into a string representation. This representation uses an XML format, however this is not a fully qualified XML representation as it doesn't specify a DTD. It is however valid XML. This XML therefore is only used between [HttpClient](#)s and [HttpServer](#)s.

JOTPCodec

JOTP stands for 'Java Object Transmission Protocol' and is used for encoding binary data. This codec turns a [DataPacket](#) into a hexadecimal string representation. The advantage of using this codec is that it can encode any java object as a string representation because it can represent any literal in a string format. For example the [DPTPCodec](#) could not encode [DataPackets](#) that contain binary data, such as, for example, integers or classes. In this instance the [JOTPCodec](#) should be used.

DPTPPaddingCodec

The DPTPPaddingCodec extends the DPTPCodec and it is used to wrap or pad out the special characters used by DPTPCodec in encoding and decoding. The special characters are `<`, `>` and their corresponding XML entity reference values `<` and `>`. If using [DataPackets](#) with XML elements as values, it may be appropriate to use the DPTPPaddingCodec to ensure data integrity. This codec uses a padding string defined by `channel.codec.paddingstring` property. If none is defined, it will default to `^`.

com.bankframe.fe.jsp.BankframePage

All JSPs consist of two components: a java bean and a `.jsp` file. The java bean is used to store the information that is either input by the user or displayed on the HTML page. The `.jsp` file transforms this information into HTML.

The `com.bankframe.fe.jsp.BankframePage` class is the super-class that all java beans used with JSPs are derived from.

The `BankframePage` class has the following methods:

- **executeRequest()**. A JSP sends a request to a Financial Component by invoking the java bean's `executeRequest()` method.
- **handleResponse()**. Each java bean overrides this method in order to process the response data returned from the Financial Component.
- **isError()**. This method can be invoked to check if the Financial Component returned an error.
- **getErrorMessage()**. This method will return the error message if the Financial Component returned an error.

Communicating over HTTP

MCA Services provides a channel client and channel server to send data over HTTP connections. It is recommended that all data sent over HTTP connections should use these classes.

Currently the majority of requests that are made to Siebel Retail Finance are over HTTP connections. Channel management provides a customizable method of sending data over HTTP connections known as DPTP ([DataPacket](#) Transmission protocol). It sends a serializable string representation of [DataPackets](#) over the HTTP connection. This is the most common way that fat Siebel Retail Finance clients will use to send and receive data to and from Financial Components.

When using DPTP, a codec is specified to encode/decode the data over the wire. For each codec there is an associated MIME type. For instance the MIME type [application/x-eontec-datapacket-hex](#) corresponds to the [JOTPCodec](#) class. All MIME types to codec mappings are specified in the [BankframeResource.properties](#) file, while there is a client setting to specify which MIME type the HTTP channel client should use (and thus which codec to use).

The HTTP server is an instance of the [javax.servlet.http.HttpServlet](#) class that listens for HTTP requests on a given port. It also uses the [BankframeResource.properties](#) file to determine all the codecs that it should support. It reads the MIME type to codec mappings and creates [Codec](#) objects for each specified mapping. Upon receiving a HTTP request it will read the [content-type](#) field from the HTTP header information and use the mapping information to select the codec to decode the request data. It will also use this mapping to encode the response to send back over the HTTP connection.

Thin clients using HTML Forms

A common way whereby clients send and receive data from MCA is through a web browser using HTML forms. In this case there is no Siebel Retail Finance channel client, instead the web browser is the client. This is because the browser will indirectly send the data to MCA.

This topic describes how HTML forms should be written to allow data to be sent to MCA Services. Communication with MCA is handled through JSPs, which encapsulate the request data (HTTP post request) as a [ServletRequest](#) object. This object contains the data entered in the form along with the name of each field of the form. When this object gets passed to MCA, MCA must retrieve all the field names along with the data entered for those fields and convert this data into one or more [DataPackets](#). It is this requirement that multiple [DataPacket](#) requests must be constructed from a single HTTP post request which has led to the following HTML form syntax.

HTML Form Syntax

In order to send data from HTML forms to MCA Services, the names given to each field in the form must be valid. This allows form designers to name fields in such a way to allow requests to be encoded as either single or multiple [DataPackets](#).

If data from a HTML form is to be converted to a single [DataPacket](#) request, then all form names must not contain the square brackets ('[' or ']'). Other than this convention any other previously valid names are still valid.

However if the data from a form should be converted into a multiple [DataPacket](#) request there are a number of rules that must be adhered to. Failure to adhere to these rules will cause the request to fail and the server to report an exception. The convention is that each field in the form must contain a number identifying which request [DataPacket](#) the data from that field should be part of.

Syntax Rules

All form fields, including hidden fields, must have a valid request packet number in their name if they are to form a multiple [DataPacket](#) request. If the request is to be a single [DataPacket](#) request then no packet numbers are needed.

- This number must be immediately preceded with '[' and immediate followed by ']'.
- No additional characters may follow the ']' character.
- All characters between '[' and ']' must be numeric.
- The 'REQUEST_ID' and 'DATA PACKET NAME' fields must be followed with [0], that is, they must be contained in the first [DataPacket](#) of the request.
- There must be a sequential order for the packets numbers. If a field exists that has a packet number 5, then there must exist a field with packet number 4.

It should be noted that all fields in the form will get encoded as HTTP parameters and the Server processing them will process these HTTP parameters. However HTTP requests can also contain attributes. These can be set in Java code, and may be set in some classes that extend the [BankFramePage](#) class. These attributes should be named according to the above syntax. Failure to do this will result in these attributes being ignored. However an exception will not be thrown for an incorrectly formatted attribute as happens for incorrectly formatted parameters. This is because many application servers will introduce their own attributes. This means that when processing attributes there is no way of distinguishing between a Siebel Retail Finance attribute and an application server attribute, so incorrectly formatted attributes will be ignored to ensure that an exception is not raised for a server attribute.

HTML Form Field Examples

Valid fields include

[REQUEST_ID](#)[0]

[DATA PACKET NAME](#)[0]

[ADDRESS1](#)[3] Provided there exists a field with packet number 2.

Invalid fields include

[REQUEST_ID](#)[3] [REQUEST_ID](#) must be in packet number 0.

[DATA PACKET NAME](#)[56] Must be in packet number 0.

[ADDRE](#)[1][SS1](#) – '[' Is not at the end of the string.

[1][ADDRESS1](#) – '[' Is not at the end of the string.

[ADDRESS1](#)[3] If there does not exist a field with packet number 2.

[ADDRESS1](#)[c3] Packet number is not numeric.

Sample valid form

```
<form method="post" action="jspservertest.jsp">
```

```

<table>

<tr>

<td>Field 1:</td>

<td><input type="text" name="FIELD1[1]"></td>

</tr>

<tr>

<td>Field 2:</td>

<td><input type="text" name="FIELD2[2]"></td>

</tr>

<tr>

<td>Field 3:</td>

<td><input type="text" name="FIELD3[1]"></td>

</tr>

</table>

<input type="hidden" name="REQUEST_ID[0]" value="MC999">

<input type="hidden" name="DATA PACKET NAME[0]" value="TEST">

<input type="submit" value="Submit">

</form>

```

Configuring Channel Management Properties

The channel client is configured in the `BankframeResource.properties` file. `ChannelManagerFactory` class will pick up these properties and supply an appropriate channel client class based on these properties.

Some properties are generic to all channel clients, while some are specific to a given channel client. All the generic properties are prefixed with the keyword `channel` only, while all specific properties are prefixed with a prefix specific to that client. The default constructor of all clients should accept no parameters and read all information needed to construct from the `BankframeResource.properties` file.

Codec Mapping Properties

These properties map MIME types to codec class names and are used by the HTTP client and server classes. All mappings are prefixed with `channel.http.codec.mapping` and followed with the actual mapping.

`channel.http.codec.mapping.application/x-eontec-datapacket-xml=com.bankframe.ei.channel.codec.DPTPCodec` maps the codec class

`DPTPCodec` to the MIME type `application/x-eontec-datapacket-xml`.

By using these mappings, all the valid codecs that a Http server can support are not hard coded into MCA. It is important to be aware that a HTTP channel property (`channel.http.client.contentType`) must match one of the mappings specified in the `BankframeResource.properties` file.

Valid Properties

`channel.client` – The fully qualified class name of the channel client to be used. This allows the client channel factory to supply instances of this class.

`channel.http.client.url`. This specifies the URL of the Http Server (Servlet URL) that the HTTP client will connect to.

`channel.http.client.contentType`. A property specific to the Http client manager. This specifies the MIME type of the encoding that the client will use.

- `channel.http.codec.mapping.application/x-eontec-datapacket-xml=com.bankframe.ei.channel.codec.DPTPCodec`

- `channel.http.codec.mapping.application/x-eontec-datapacket-hex=com.bankframe.ei.channel.codec.JOTPCodec`

Configuring HttpClient

The `BankframeResource.properties` file requires the following changes in order to configure the HTTPS client settings:

```
channel.client=com.bankframe.ei.channel.client.HttpsClient
```

```
channel.http.client.url=https://<URL of the HTTP server>
```

```
channel.https.truststore=<path to truststore>
```

```
channel.https.keystore=<path to identity keystore>
```

```
channel.https.keystorePassword=<keystore password>
```

```
channel.https.ssl.protocol=<class name of the SSL protocol>. Possible values are:
```

```
com.sun.net.ssl.internal.www.protocol for WebLogic and
```

```
com.ibm.net.ssl.internal.www.protocol for WebSphere.
```

```
channel.https.ssl.provider=<class name of the SSL provider>. Possible values are
```

```
com.sun.net.ssl.internal.ssl.Provider for WebLogic and com.ibm.jsse.JSSEProvider for WebSphere.
```


Developing Custom Channel Clients and Servers

If there is a channel that a Siebel Retail Finance client wishes to communicate over, but channel clients do not exist then developers can write their own channel client and server classes to handle that particular channel.

If this new channel uses HTTP then the developer need only write a custom codec class that adheres to the codec interface and edit the `BankframeResource.properties` file to include the new codec in the MIME type to codec class name mappings to use this new codec.

If however the channel is not over HTTP then the developer should write a server class (possibly a servlet) that can process incoming requests in the channel specific format. This means that the server will accept requests in the channel specific format and convert this to a `Vector` of `DataPackets` that is forwarded to the `RequestRouter`. The server must then read the response (in `DataPackets`) from the `RequestRouter` and return this over the channel in the channel specific format.

The developer must also develop a channel client class that implements the `com.bankframe.ei.channel.client.ChannelClient` interface that mandates that there must be a `send(Vector)` method. The developer should write this method to take a `Vector` of `DataPackets` and send it to the server encoded in the channel specific format, handling any channel specific communication issues that may arise on the way. The aim is to make sending and receiving `DataPackets` transparent to the Siebel Retail Finance client. This method should always return a `Vector` of `DataPackets` to the Siebel Retail Finance client even if communication errors occur.

Thin and Fat Client Examples

The following examples illustrate how both thin client and fat client solutions can communicate with MCA over HTTP. For the sake of simplicity the following assumptions are made:

- There exists a Financial Component called `eontec.bankframe.examples.CreditTransfer`.
- The Financial Component is an implementation of a Credit Transfer.
- The Financial Component is deployed on Route: EX330.
- The Financial Component expects a request `DataPacket` with the following format as input:

Key	Value
<code>DATA_PACKET_NAME</code>	CREDIT_TRANSFER
<code>REQUEST_ID</code>	EX330
<code>FROM_ACCOUNT</code>	A/C Number money comes from
<code>TO_ACCOUNT</code>	A/C Number money goes to
<code>AMOUNT</code>	Amount to transfer

- The Financial Component returns a `Vector` of `DataPackets` containing a single `DataPacket` with the following format:

Key	Value
DATA_PACKET_NAME	CREDIT_TRANSFER_RESPONSE
REQUEST_ID	00000
FROM_ACCOUNT	A/C Number money came from
TO_ACCOUNT	A/C Number money went to
AMOUNT	Amount transferred
NEW_BALANCE	New balance of a/c money was transferred from

The following example illustrates how a HTML based solution communicates with MCA. Siebel Retail Finance HTML solutions are built using Java Server Pages (JSPs) The following example illustrates a simple JSP that submits some information to a Financial Component.

Thin client example

credittransfer.html

```
<html>

<head><title>Credit Transfer</title></head>

<body bgcolor="#ffffff">

<form method="post" action="credittransfer.jsp">

<table border="0" width="50%">

<tr><td>To Account:</td><td><input type="text" name="TO_ACCOUNT"
size="25"></td></tr>

<tr><td>From Account:</td><td><input type="text" name="FROM_ACCOUNT"
size="25"></td></tr>

<tr><td>Amount:</td><td><input type="text" name="AMOUNT" size="25"></td></tr>

</table>

<input type="hidden" name="REQUEST_ID" value="EX330">

<input type="hidden" name="DATA_PACKET_NAME" value="CREDIT_TRANSFER">

<input type="submit" value="Submit">

</form>

</body>
```

```
</html>
```

credittransfer.html explanation

This is the HTML form used to submit the credit transfer information:

Note that the name of the input fields must match the name of the corresponding entry in the `DataPacket`. This is a single `DataPacket` request so '[' or ']' is not used.

The first hidden input field contains the `REQUEST_ID` value to put in the `DataPacket`.

The second hidden input field contains the name to give the `DataPacket`.

When the Submit button on the HTML form is pressed the form data will be submitted to a JSP called `credittransfer.jsp`.

credittransfer.jsp

```
<%@ page import="com.BankFrame.examples.credittransfer.jsp.CreditTransferPage" %>

<jsp:useBean id="creditTransferPage" scope="page"
class="com.BankFrame.examples.credittransfer.jsp.CreditTransferPage" />

<%= creditTransferPage.executeRequest(config,request,response) %>

<html>

<head><title>Credit Transfer Completed</title></head>

<body bgcolor="#ffffff">

<table border="0" width="50%">

<tr><td>To Account:</td><td><jsp:getProperty name="creditTransferPage"
property="toAccount" /></td></tr>

<tr><td>From Account:</td><td><jsp:getProperty name="creditTransferPage"
property="fromAccount" /></td></tr>

<tr><td>Amount:</td><td><jsp:getProperty name="creditTransferPage"
property="amount" /></td></tr>

<tr><td>New Balance:</td><td><jsp:getProperty name="creditTransferPage"
property="newBalance" /></td></tr>

</table>

</body>

</html>
```

credittransfer.jsp Code Explanation

`credittransfer.jsp` carries out the following steps:

Imports a java bean called `com.BankFrame.examples.credittransfer.jsp.CreditTransferPage`.

Creates an instance of this java bean called `creditTransferPage`.

Invokes the `creditTransferPage.executeRequest()` method to send the data from the HTML form to MCA.

When the `executeRequest()` method is invoked, the HTML Form data is translated into a `DataPacket` and the `DataPacket` is passed to the Financial Component specified by the `REQUEST_ID` in the `DataPacket`. The response data from the Financial Component is returned to the `CreditTransferPage` java bean. The `CreditTransferPage` java bean parses and caches the response data.

The JSP uses the `<jsp:getProperty/>` tags to retrieve the response data cached in the `CreditTransferPage` java bean.

`credittransfer.jsp` is parsed by the JSP Engine to produce the HTML output.

CreditTransferPage

```
package com.BankFrame.examples.credittransfer.jsp;

import java.util.Vector;

import com.BankFrame.bo.DataPacket;

import com.BankFrame.fe.jsp.BankFramePage;

public class CreditTransferPage extends BankFramePage {

    private String fromAccount = null;

    private String toAccount = null;

    private String amount = null ;

    private String newBalance = null;

    public CreditTransferPage() {}

    public String getFromAccount() { return this.fromAccount; }

    public String getToAccount() { return this.toAccount; }

    public String getAmount() { return this.amount;}

    public String getNewBalance() { return this.newBalance; }

    public void setFromAccount(String fromAccount) { this.fromAccount =
fromAccount; }

    public void setToAccount(String toAccount) {this.toAccount = toAccount; }
```

```

public void setAmount(String amount) {this.amount = amount; }

public void setNewBalance(String newBalance) {this.newBalance = newBalance; }


public void handleResponse(Vector DataPackets) {

    super.handleResponse(DataPackets);

    if ( this.isError() == false ) {

        DataPacket dp = (DataPacket)DataPackets.elementAt(0);

        this.fromAccount = dp.getString("FROM_ACCOUNT");

        this.toAccount = dp.getString("TO_ACCOUNT");

        this.amount = dp.getString("AMOUNT");

        this.newBalance = dp.getString("NEW_BALANCE");

    }

}

}

```

CreditTransferPage Code Explanation

This java bean enables the JSP and the Financial Component to communicate. The bean has four attributes: `toAccount`, `fromAccount`, `amount` and `newBalance`. The first three represent information input by the user and the final attribute represents data returned from the Financial Component.

The bean is derived from the `com.BankFrame.fe.jsp.BankFramePage` class. This means the bean inherits `BankFramePage`'s `executeRequest()` method.

The JSP invokes the bean's `executeRequest()` method to send the data to the Financial Component. When the Financial Component has completed processing the bean's `handleResponse()` method will be invoked. This enables the bean to process the data returned from the Financial Component. In this case it stores the `toAccount`, `fromAccount`, `amount`, and `newBalance` values returned by the Financial Component. The JSP then uses the `<jsp:getProperty/>` tags to retrieve these values from the bean.

Fat Client Example

The following example illustrates a console based client application that communicates with MCA over HTTP. The application expects the following command line parameters:

`to` Account number to transfer money to.
`from` Account number to transfer money from.
`amount` Amount of money to transfer.

Sample Implementation

```
package com.BankFrame.examples.credittransfer;

import java.util.Vector;

import com.BankFrame.bo.DataPacket;

import com.BankFrame.ei.comms.EHTTPCommsManager;

public class Client {

    private String toAccount;

    private String fromAccount;

    private String amount;

    public Client() {}

    public void init(String[] args) {

        for ( int i = 0 ; i < args.length ; ++i ) {

            if ( args[i].equals("-to") ) {

                this.toAccount = args[++i];

            }

            if ( args[i].equals("-from") ) {

                this.fromAccount = args[++i];

            }

            if ( args[i].equals("-amount") ) {

                this.amount = args[++i];

            }

        }

    }

    public void doCreditTransfer() {

        try {
```

```

HttpClient client = new HttpClient();

DataPacket dp = new DataPacket("CREDIT TRANSFER");

dp.put(DataPacket.REQUEST_ID,"EX330");

dp.put("TO_ACCOUNT",this.toAccount);

dp.put("FROM_ACCOUNT",this.fromAccount);

dp.put("AMOUNT",this.amount);

Vector responses = client.send(dp);

dp = (DataPacket)responses.elementAt(0);

System.out.println("Transferred: " + dp.getString("AMOUNT") +

    " from a/c: " + dp.getString("FROM_ACCOUNT") +

    " to a/c: " + dp.getString("TO_ACCOUNT") +

    " new balance: " + dp.getString("NEW_BALANCE"));

} catch (Exception e) {

    System.out.println("An exception occurred: " + ex.toString());

}

}

public static void main(String[] args) {

    Client client = new Client();

    client.init(args);

    client.doCreditTransfer();

}

}

```

Code Explanation

The above code carries out the following actions:

- Parses the command-line flags, this is done in the `init()` method.
- Sends a `DataPacket` to the Financial Component, with the credit transfer details.
- Parses the response returned from the Financial Component and displays the results.

The `doCreditTransfer()` method does the following:

- Creates a `HttpClient` instance. This is the channel client used to communicate with MCA. The `HttpClient` instance is initialized with no parameters. This indicates that the channel specific properties from the `BankframeResource.properties` file should be read to initialize parameters.
- Creates a `DataPacket` with the information expected by the `eontec.bankframe.CreditTransfer` Financial Component.
- Uses the `HttpClient.send()` method to send the `DataPacket` to MCA.
- Parses the information returned from the Financial Component and displays this information.

XML B2B Channel Management

The XML/XSL support in MCA Services uses DPTP (`DataPacket` Transmission Protocol) XML format. MCA provides three different options for communicating with Financial Components via XML. These options are:

- A custom XML parser that supports the parsing of DPTP only. This parser is optimized for speed but requires that all input be formatted correctly. This option is the best choice when performance is of paramount importance and the client is able to generate correctly formatted DPTP XML.
- A DPTP parser that uses the JAXP parser to parse the XML. This parser is not as fast as the first option but is more robust in handling incorrectly formatted XML. This option is a good choice for use during the development phase of a project as the JAXP parser will provide detailed error messages about any formatting issues with the incoming data.
- An XSL parser that uses XSL to transform an incoming request from an arbitrary XML format into DPTP XML. This option is the best choice when the client is not able to generate DPTP XML, it provides the most flexibility in the types of XML that can be processed. However the XSL transform requires a certain amount of overhead so this option will not be able to deliver the same levels of performance as the other two options.

These three options are implemented by a number of different codec classes described below:

Package: `com.bankframe.ei.channel.codec`

`XMLDOMCodec`

This is an abstract class that serves as a base class for codecs that use JAXP to encode XML data. This class provides methods for transforming String data to an XML DOM object and vice versa.

`DPTPDOMCodec`

This codec is similar to `DPTPCodec` in that it encodes XML data encoded in DPTP format, however it uses JAXP to parse the XML data. This provides more robust error handling at the expense of slower performance.

`XMLXSLCodec`

This is an abstract class which sub-classes `DPTPDOMCodec`. This class serves as a base class for codecs that use XSL to transform arbitrary XML into DPTP XML. The incoming XML is parsed into a DOM tree,

the transformation is applied to transform this DOM tree into DPTP XML and then the transformed data is parsed by the [DPTPDOMCodec](#). On the return trip the reverse process is applied.

Package: com.bankframe.ei.xml

The codec classes defined above rely on the classes defined in the [com.bankframe.ei.xml](#) package to carry out processing of XML streams.

com.bankframe.ei.xml.EDocumentBuilder

This class is used to build an XML [Document](#) from an XML [InputSource](#), the resulting [Document](#) can be XML of any type. The parse method in the [EDocumentBuilder](#) class is used to create XML [Document](#) objects. The [EDocumentBuilder](#) class also provides a [newDocument\(\)](#) method to create an empty XML [Document](#) object, as well as methods that will let you determine the properties of the underlying XML parser being used.

The default implementation of the [EDocumentBuilder](#) class utilizes the Java API for XML Processing (JAXP), version 1.1 released by Sun. Therefore, the underlying XML parser that you wish the [EDocumentBuilder](#) to use can be specified using Java environment variables as described in the JAXP specification.

com.bankframe.ei.xml.EDocumentBuilderFactory

This class is used to obtain an instance of an [EDocumentBuilder](#). The current release of MCA uses only the default implementation of an [EDocumentBuilder](#), which is described above.

com.bankframe.ei.xml.XMLTransformer

This class is used to transform an XML [Document](#) object from one XML format to another. In most cases, this class will be used to transform non-Siebel Retail Finance XML [Documents](#) into Siebel Retail Finance XML [Documents](#), or to transform Siebel Retail Finance XML [Documents](#) into non-Siebel Retail Finance XML [Documents](#). The [XMLTransformer](#) class provides a [transform\(Document, Document, String\)](#) method that will accept a source XML [Document](#), a result XML [Document](#) and the URL of the style sheet to carry out the transformation. The default implementation of the [XMLTransformer](#) utilizes the Java API for XML Processing, version 1.1 released by Sun. It transforms [Documents](#) using a user-defined XSL style sheet. Since the Siebel Retail Finance [XMLTransformer](#) utilizes JAXP, the underlying XML processor that you wish to use can be specified in Java environment variables, as noted in the JAXP specification.

com.bankframe.ei.xml.XMLErrorHandler

This class is used to report errors encountered during the processing of XML streams. This class redirects the error messages produced by the underlying JAXP parser to the BankFrame logging framework.

Mapping XML Requests to Financial Components

There are two scenarios to consider when MCA handles a request to a Financial Component in XML format:

XML Transactions In Siebel Retail Finance Format

The first scenario is when a client (typically some third party B2B application) sends a request that adheres to the Siebel Retail Finance XML format. Therefore the client sends an XML request encoded in the DPTP XML format. The client also expects a response from MCA in the same format. In this instance, the Siebel Retail Finance XML Channel Manager does not require any extra configuration. Since the client will be using the MCA XML format, the request will be automatically converted into a [Vector](#) of [DataPackets](#) and passed through the [RequestRouter](#) to the appropriate Financial Component. The [Vector](#) of [DataPackets](#) response from the Financial Component will automatically be re-formatted into a Siebel Retail Finance XML [Document](#) and sent back to the client.

XML Transactions in Non-Siebel Retail Finance Format

In the second scenario, the client will be sending MCA a request that is in an arbitrary XML format, for example, FpXML, cXML or fooXML. In this instance, incoming requests must first be transformed into the Siebel Retail Finance XML format so that it can be parsed into a [Vector](#) of [DataPackets](#) for processing. In order to accomplish this, the developer must determine a correspondence between the client XML transaction types, and Siebel Retail Finance Financial Components. It is assumed that it will be possible to find a mapping pattern between the client XML format and Siebel Retail Finance Financial Components. Once these mappings are defined, the developer is responsible for writing an XSL style sheet that transforms the incoming XML Document into a Siebel Retail Finance XML Document.

After the request is processed, the [Vector](#) of [DataPackets](#) returned by the Financial Component must be re-formatted back into an XML format that the client expects. Once again, this is accomplished by defining an XSL stylesheet to transform the Siebel Retail Finance XML format into the client's XML format. Note that you will generally write two separate XSL stylesheets for each mapping. One stylesheet to transform incoming requests into Siebel Retail Finance XML format, and one stylesheet to transform outgoing responses back into the client XML format.

The name of the XSL stylesheets to be used in the transformation is defined by sub-classing the [XMLXSLCodeC](#) class and defining the content types that the sub-class handles (It is assumed that each different XML encoding will have a different MIME content type). These content types are then mapped to the URL of an XSL file via settings defined in the [BankFrameResource.properties](#) file.

Configuring XML B2B Channel Management

XML Channel Management is configured in the BankframeResource.properties file. The XML Channel Management properties are listed in Table 2. XML Channel Management Properties.

Table 2. XML Channel Management Properties

Property	Description
xml.eDocBuilder.systemId	<p>Defines a default location for DTDs for incoming XML documents. This is used as a back-up if the DTD is not specified with a full URL in the incoming XML documents)</p> <p>If an incoming XML document specifies its DTD in the format</p>

Property	Description
	<p><code>SYSTEM "fooXML.dtd"</code>, the parser looks for this file at the location specified by the <code>systemId</code> property.</p> <p>If the incoming XML document specifies its DTD in the format <code>SYSTEM http://www.siebel.com/xml/dtd/fooXML.dtd</code>, the <code>systemId</code> property is ignored.</p>
<code>xml.parser.validating</code>	Defines whether the underlying XML parser used should be validating or non-validating. This property should be set to <code>true</code> or <code>false</code> .
<code>xml.parser.ignoreComments</code>	Defines whether or not the underlying XML parser should ignore comments. This property should be set to <code>true</code> or <code>false</code> .
<code>xml.parser.ignoreElementContentWhiteSpace</code>	Defines whether the underlying XML parser should ignore white space or not. This property should be set to <code>true</code> or <code>false</code> .
<code>xml.parser.namespaceAware</code>	Defines whether or not the underlying XML parser is namespace aware. This property should be set to <code>true</code> or <code>false</code> .
<code>channel.http.xml.xsl.request.content-type.application/x-foo-request-xml</code>	Maps the request MIME content-type with the appropriate XSL style-sheet. The value should be in the format <code>http://localhost/eontec/mca/stylesheets/foo-xml-request.xsl</code> .
<code>channel.http.xml.xsl.response.content-type.application/x-foo-response-xml</code>	Maps the response MIME content-type with the appropriate XSL style-sheet. The value should be in the format <code>http://localhost/eontec/mca/stylesheets/foo-xml-response.xsl</code> .

Developing Custom XML and XSL Codecs

Custom XML Codecs

Codecs that must manipulate an XML stream can sub-class the `XMLDOMCodec` class, which provides methods for marshalling String data to DOM trees, and vice versa.

Custom XSL Codecs

Codecs that must use XSL to transform XML into DPTP format can sub-class the `XMLXSLCodec` class. The sub-class need only specify the `content-type` of the incoming request and outgoing response.

Once this is done and the relevant `BankframeResource.properties` settings are configured, this class will apply the appropriate XSL style-sheet to the incoming request and the outgoing response.

DPTPCodec Transmission Format

The `DPTPCodec` marshals `Vectors` of `DataPackets` to and from an XML based String representation. The XML format used is very simple and very compact in order to keep the request and response message sizes as small as possible. The `DPTPCodec` parses the XML directly, it does not rely on third-party XML parsers. This ensures that the `DPTPCodec` marshals data very quickly, but also requires that the XML data is formatted exactly as described below. The XML data is not validated before parsing therefore it is essential that the data is well formed.

Sample request file

A sample implementation of the `CREDIT_TRANSFER` request follows.

```
<?xml version="1.0"?>

<v n="r">

  <d n="CREDIT_TRANSFER">

    <a n="OWNER">Oracle </a>

    <a n="AMOUNT">1400.00</a>

    <a n="FROM_ACCOUNT">11236745</a>

    <a n="TO_ACCOUNT">11246890</a>

    <a n="REQUEST_ID">EX330</a>

  </d>

</v>
```

- The file starts with the standard XML processing instruction.
- `Vectors` are denoted by the `<v>` element. Every request will have a containing `Vector`, this `Vector` is given the name `r` (denoting root element) by convention.
- `DataPackets` are denoted by the `<d>` element, each `DataPacket` has a name which is defined by the `n` (name) attribute.
- `DataPacket` attributes are denoted by the `<a>` element. Each attribute has a name defined by the `n` attribute. The value of the attribute is defined between the enclosing `<a>` and `` tags.
- The XML element and attribute tags are kept short to ensure the message size is as small as possible.
- The `DPTPCodec` strips all unnecessary white-space between elements. Carriage returns and indentation have been added to the example above for clarity. The actual request would look like this:

```
<?xml version="1.0"?><v n="r"><d n="CREDIT_TRANSFER"><a n="OWNER">Oracle</a><a
n="AMOUNT">1400.00</a><a n="FROM_ACCOUNT">11236745</a><a
n="TO_ACCOUNT">11246890</a><a n="REQUEST_ID">EX330</a></d></v>
```

XML Format Description

- The Document must commence with an XML processing instruction.
- The root element must have a Vector element (<v>).
- All <v> elements must have a name attribute (n).
- The root Vector element's name is always: r.
- The root Vector element can contain one or more DataPacket (<d>) elements.
- Each DataPacket element must have a name (n) attribute.
- Each DataPacket element can contain one or more DataPacket attribute (<a>) elements.
- Each DataPacket attribute element must have a name (n) attribute.
- The DataPacket attribute's value is located between the <a> and tags.

XML and XSL Examples

This example assumes that an XML stream encoded in the third-party `foo-corp-xml` format must be transformed to and from DPTP XML format so that it can be processed by a SRF Financial component. The XML contains a credit transfer request which must be processed by a Siebel Retail Finance Financial component. Below is the input XML:

Input XML

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE foo-corp-xml SYSTEM "foo-corp-xml.dtd">

<foo-corp-xml>

  <payment type="credit-transfer">

    <source-account>1234567890</source-account>

    <destination-account>1111222245</destination-account>

    <amount currency="EUR">1200.00</amount>

    <narrative>J Bloggs</narrative>

  </payment>

</foo-corp-xml>
```

The data in this request must be converted to [DataPackets](#) of information so that they can be passed to a Siebel Retail Finance Financial Component. The sender of the above request must ensure that the [content-type](#) header field in the HTTP request is set to the correct MIME type for the XML format. MCA uses the [content-type](#) field to determine the appropriate codec to use to decode the XML.

XSL Style-sheet

An XSL style-sheet must be defined to transform the foo-corp-xml request into a DPTP request. A sample implementation follows.

```
<?xml version="1.0" encoding="UTF-8"?>

<xsl:transform xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0"
  xmlns:xalan="http://xml.apache.org/xslt">
  <xsl:template match = "/">
    <xsl:call-template name="payment-template"/>
  </xsl:template>
  <xsl:template name="payment-template">
    <v n="r">
      <xsl:for-each select="//payment">
        <d n="CREDIT_TRANSFER">
          <a n="REQUEST_ID">EX330</a>
          <a n="FROM_ACCOUNT"><xsl:value-of select="source-
account" /></a>
          <a n="TO_ACCOUNT"><xsl:value-of select="destination-
account" /></a>
          <a n="AMOUNT"><xsl:value-of select="amount" /></a>
        </d>
      </xsl:for-each>
    </v>
  </xsl:template>
</xsl:transform>
```

This style-sheet supplies the [DataPacket](#) name and [REQUEST_ID](#) which is essential for routing the request to the correct Financial Component.

XSL Codec

A codec must be defined that is capable of applying the above XSL to the incoming request. A sample implementation follows.

```
package com.bankframe.examples.channel.xmlxsl;

import com.bankframe.ei.channel.codec.XMLXSLCodec;

public class FooXmlXslCodec extends XMLXSLCodec {

    public static final String REQUEST_CONTENT_TYPE="application/x-foo-request-xml";

    public static final String RESPONSE_CONTENT_TYPE="application/x-foo-response-xml";

    public FooXmlXslCodec() {

        super(REQUEST_CONTENT_TYPE,RESPONSE_CONTENT_TYPE);

    }

    public String getName() {

        return this.getClass().getName();

    }

}
```

This class sub-classes `com.bankframe.ei.channel.codec.XMLXSLCodec`. `XMLXSLCodec` and provides all the functionality required for applying an XSL transformation to incoming requests and outgoing responses. All that the `FooXmlXslCodec` class needs to do is define the `content-types` of the incoming and outgoing requests. `XMLXSLCodec` uses the `content-type` to determine the XSL file to apply for the specified request or response.

Request Router Web Service

The MCA Services RequestRouter service provides access to all SRF Financial Components. The RequestRouter can be deployed as a web service, effectively web enabling all the underlying services. Any request that is processed by the Request Router can be invoked via the Request Router web service. The Request Router contains a `processDataPackets(Vector dataPackets)` method which allows any `DataPacket` request to communicate with any EJB in the Routes database table. The Web service allows this method to be invoked on the `RequestRouter`. The Web service allows the `processDataPacket` method of the `RequestRouter` EJB to be invoked by any client, regardless of the programming language the client is written in, or operating system that it is run on.

Request Router WSDL

MCA Services provides a WSDL description of the Request Router Web service. This WSDL describes the location of the Web service, and how a client can interact with it. The WebLogic WSDL is shown below:

```
<definitions
  targetNamespace="java:com.bankframe.services.requestrouter.webservice"
  xmlns:tns="java:com.bankframe.services.requestrouter.webservice"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/1999/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
>

<types>

<schema targetNamespace='java:com.bankframe.services.requestrouter.webservice'
  xmlns='http://www.w3.org/1999/XMLSchema'>

</schema>

</types>

<message name="processDataPacketsRequest">

<part name="arg0" type="xsd:string" />

</message>

<message name="processDataPacketsResponse">

<part name="return" type="xsd:string" />

</message>

<portType name="WebserviceRequestRouterPortType">

<operation name="processDataPackets">

<input message="tns:processDataPacketsRequest" />

<output message="tns:processDataPacketsResponse" />

</operation>

</portType>
```



```

<binding name="WebserviceRequestRouterBinding"
type="tns:WebserviceRequestRouterPortType"><soap:binding style="rpc"
transport="http://schemas.xmlsoap.org/soap/http" />

<operation name="processDataPackets">

<soap:operation soapAction="urn:processDataPackets" />

<input><soap:body use="encoded" namespace='urn:WebserviceRequestRouter'
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" /></input>

<output><soap:body use="encoded" namespace='urn:WebserviceRequestRouter'
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" /></output>

</operation>

</binding>

<service name="WebserviceRequestRouter"><documentation>todo</documentation><port
name="WebserviceRequestRouterPort"
binding="tns:WebserviceRequestRouterBinding"><soap:address
location="http://localhost:7001/BankFrameMCA/WebServices/RequestRouter" /></port><
/service></definitions>

```

The `<service>` tag in the WSDL defines the Web service. The sub-tag `<port>` defines where to find the Web service and the operations (methods) supported. In this WSDL the port is called `WebserviceRequestRouterPort`. It has one operation called `processDataPackets`, which itself declares its input and output message. These are defined earlier in the WSDL. The second sub-tag is the `<soap:address>` tag which defines the actual location of the Web service. In this case the Web service can be invoked by sending a SOAP request, adhering to the definitions provided in the WSDL, to <http://localhost:7001/BankFrameMCA/WebServices/RequestRouter>.

The data types that can be defined in the WSDL must be SOAP data types. SOAP data types map to primitive java data types such as `long`, `double`, `float` and `string`, but not to object data types such as `DataPacket`, or `Vector`. In the sample WSDL both `messages` are defined with a single `part` type. The part says that the argument to the message is of type `xsd:String`. When using WSDL, all the arguments to operations that are declared must be a valid SOAP data type. This means that `DataPackets` which are used internally throughout MCA Services, and by the Request Router's `processDataPacket()` method, cannot be used as an input to, or an output from, the Web service. Because of this limitation all requests to the Request Router Web service must be represented in XML. This means that a `DataPacket`, or a request `Vector` of `DataPackets`, must first be mapped to XML using the class `com.bankframe.ei.channel.codec.DPTPCodec` before it can be invoked using the Web service.

Request Router Web Service Class Descriptions

Package `com.bankframe.services.requestrouter.webservice`

This package defines a session bean that talks to the Request Router EJB. This session maps the incoming XML request to a `Vector` of `DataPacket`(s) before forwarding the `DataPacket`(s) to the Request Router.

Class `WebserviceRequestRouterBean`

This class contains a single method with the following signature:

```
public String processDataPackets(String request) throws ProcessingErrorException,
RemoteException
```

This method takes a `String` as a parameter and uses the `DPTPCodec` to convert it to a `Vector` of `DataPackets`. It then calls the `RequestRouterUtils.processDataPackets(Vector dataPackets)` which passes the generated `DataPacket`(s) to the Request Router EJB, which then processes them. When a response from the Request Router is received it converts it back to an XML string, using the codec, and returns this XML.

Package `com.bankframe.ei.channel.codec`

This package contains codecs that are used in MCA Services. The `WebserviceRequestRouter` session bean that that Web service is built on uses the `DPTPCodec`. An explanation of the codec and its usage follows.

Class `DPTPCodec`

The `DPTPCodec` marshals `Vectors` of `DataPackets` to and from an XML based `String` representation. The XML format used is very simple and very compact, in order to keep the request and response message sizes as small as possible. The `DPTPCodec` parses the XML directly, it does not rely on third-party XML parsers such as Xerces or JAXP. This ensures that the `DPTPCodec` marshals data very quickly, but also requires that the XML data is formatted exactly as described below. The XML data is not validated before parsing so it is essential that the data is well formed.

Sample request file

The example below shows how a sample credit transfer request might be encoded using the DPTP codec:

```
<?xml version="1.0"?>

<v n="r">

  <d n="CREDIT_TRANSFER">

    <a n="OWNER">eontec Ltd</a>

    <a n="AMOUNT">1400.00</a>

    <a n="FROM_ACCOUNT">11236745</a>
```

```

        <a n="TO_ACCOUNT">11246890</a>

        <a n="REQUEST_ID">EX330</a>

    </d>

</v>

```

- The file starts with the standard XML processing instruction.
- **Vectors** are denoted by the `<v>` element, every request will have a containing **Vector**, this **Vector** is given the name `r` (denoting root element).
- **DataPackets** are denoted by the `<d>` element, each **DataPacket** has a name which is defined by the `n` (name) attribute.
- **DataPacket** attributes are denoted by the `<a>` element. Each attribute has a name defined by the `n` attribute. The value of the attribute is given between the enclosing `<a>` and `` tags.
- The XML element and attribute tags are kept short to ensure the message size is as small as possible.
- The **DPTPCodec** strips all unnecessary white-space between elements. Carriage returns and indentation have been added to the example above for clarity.

Session Affinity

Session Affinity can be used to route all requests to the same node in environments running multiple Java Virtual Machines, for example, a cluster. Caching occurs at each node in the cluster, therefore at certain points in time it is possible to have inconsistent caches of data across each node. To ensure that all requests within a user's session process data from the same cache, all requests within the user's session must be routed to the same node in the cluster. Session Affinity can address this need to route requests to the same node by configuring the load balancing mechanism for the cluster to determine if requests have come from the same user session via the configured unique token in the HTTP request's header.

Session Affinity is a mechanism whereby a unique string token is placed into all requests, under a configurable key, for the duration of a client's HTTP session. The unique token is placed into a request by the State Machine. The token is added to the HTTP request's header, within the HTTP Channel client, under a configurable key specified in `BankframeResource.properties`.

When a request hits the load balancing mechanism, the HTTP request's header is checked for a pre-defined key, for example, `HTTP_HEADER_ID`, that has been added by the Channel Client, and the load balancer will search for a mapping between `HTTP_HEADER_IDS` and IP addresses of the nodes in the cluster. If a record is not found then a node is chosen at random to route the request to and a mapping between the `HTTP_HEADER_ID` and an IP address is created. Otherwise, if a matching record is found, the request is routed to the node with the IP address matching that of the `HTTP_HEADER_ID`.

Configuring Session Affinity Properties

Session Affinity is configured in the `BankframeResource.properties` file. The Session Affinity properties are listed in Table 3. The value of properties for URL rewriting and cookies are case sensitive.

Table 3. Session Affinity Properties

Property	Value	Description
include.session.id	True or false	This value must be set to <i>true</i> to notify the State Machine to include a unique token with every request.
channel.http.client.header.HTTP_HEADER_ID	<i>SM_SESSION_ID</i>	This setting represents the State Machine configurable key for HTTP request headers. This value must be set to <i>SM_SESSION_ID</i> for HTTP requests.
Channel.http.client.header.jsessionid	SM_SESSION_ID	Property is in lower case when using URL rewriting
channel.http.client.header.JSESSIONID	SM_SESSION_ID	Property is in upper case when using cookies
channel.http.client.addHeaderFieldsToSingleHTTPHeaderField	false	Setting for using URL rewriting
channel.http.client.addHeaderFieldsToSingleHTTPHeaderField	true	Setting for using cookies
channel.http.client.singleHeaderFieldName	Cookie	Setting to specify the header field name when using cookies
channel.http.client.singleHeaderField.seperator	;	Setting to specify the separator used to distinguish different fields in the header
channel.http.client.sessionid.key	jsessionid	This property must be set to the same value as that used for channel.http.client.header<>. For URL rewriting the application servers default to use the lowercase value "jsessionid".
channel.http.client.sessionid.key	JSESSIONID	This property must be set to the same value as that used for channel.http.client.header<>. For cookies the application servers default to examine/modify the uppercase "JSESSIONID" value as a key in the cookie header field

4 Financial Process Integration

This chapter covers the MCA Financial Process Integrator (FPI) and includes the following topics:

- About Financial Process Integration
- Financial Process Integrator Metadata
- Mapping Entity Beans to Transactions
- Entity Bean Persistence and the FPI
- Financial Process Integrator Caching
- Financial Process Integrator Engine
- EIS Connectors
- Store and Forward
- Financial Process Integrator Sample Implementations
- Handling Complex Amend and Find Operations
- Handling Create and Remove Operations
- Sample Data Formatter Implementation

About Financial Process Integration

Data in SRF banking applications is modeled using entity beans, and the business logic is modeled using session beans. Session beans communicate with host systems via entity beans, and entity beans use the Financial Process Integrator to communicate with the host system.

Host systems are accessed via middleware. MCA can integrate with a number of different middleware technologies to communicate with Host systems. These middleware technologies send request data to host systems and pass back response data from the host system. However, each middleware layer does this in a different way. MCA provides an abstraction layer that hides the differences between different middleware technologies. This provides SRF Modules with a simple interface for communicating with host systems. This layer is called the Financial Process Integrator.

Overview of Interfacing with a Host System

Typically a session bean needs to read some information stored on a host system. At a high level, this is how this is modeled:

- The data stored on the host system is modeled as an entity bean.
- The session bean requests a specific instance of the entity bean.
- This request is transformed by the Financial Process Integrator into a host transaction.

- The host system processes the request, and returns some response data.
- The Financial Process Integrator transforms this response data into a format the entity bean can understand.
- The entity bean is initialized with the response data.
- The initialized entity bean instance is returned to the session bean.

The session bean does not interact directly with the Financial Process Integrator. In SRF Modules, all data in the system is modeled as entity beans. Session beans manipulate these entity beans, rather than interacting directly with the data-store (which, in the example above, is the host system). This approach maximizes the flexibility of SRF Modules; the complexity of interacting with the data store is hidden within the implementation of entity beans.

Components of the Financial Process Integrator

Persister

Entity beans can be implemented using either container managed persistence (CMP) or bean-managed persistence (BMP). With CMP, the task of persisting the entity bean's state is delegated to the EJB Container, whereas with BMP, the entity bean is responsible for persisting its state itself.

When entity beans are used to model data on host systems they must be implemented using BMP. To do this they must interact with the Financial Process Integrator. The task of communicating with the Financial Process Integrator is delegated to a persister helper object.

Cache

Typically there is a large overhead in communicating with host systems; it can take a significant amount of time for a request to be transformed into a format understood by a host system, and then processed by that host system. It is important to cache information so that communication with the host system, and transformation of data, is minimized. Caching is used in several places in the Financial Process Integrator to improve performance. The cache can be keyed by the entity primary key, or the cache can be indexed.

Cache Indexing

To facilitate the retrieval of cached data by non-primary key attributes, the cache can be indexed. Cache indexing provides the following benefits:

- Results from non-key host transactions can be reused, without the overhead of firing the host transaction again.
- Data returned by another host transaction can be retrieved even when a corresponding host transaction does not exist. For example, a host system may have a [retrieveCustomerProfile](#) transaction that returns customer details as well as account details, but may not have a single transaction for returning the account details alone.

Indexing does have its own overhead and not all host transaction responses need to be indexed. All cache index implementations must implement the basic methods defined in the [CacheIndexer](#)

interface in the `com.bankframe.services.cache` package. The cache index structures are defined using the `IndexMetaData` entity. The Cache Indexing implementation is completely configurable.

Metadata

Metadata is the information that describes how to transform data into the format that the host system understands. SRF metadata information is modeled as follows:

- Meta-data information for creating the host request is modeled using the `RequestTransactionField` entity EJB.
- Meta-data information for processing the host system response is defined by the `TransactionMetaData` and `ResponseTransactionField` entity EJBs.
- Meta-data information for handling host system error responses is modeled using the `TransactionErrorCondition` entity EJB.
- Meta-data information on which host transaction responses should be indexed is modeled using the `ResponseIndex` entity EJB.

Data Formatter

The data formatter class is responsible for interpreting the metadata and using it to transform the request data into the format that the host system understands, and conversely, to transform the response data into a format the SRF Module can understand.

Transaction Route

The `TransactionRoute` entity defines the SRF connector to use for each transaction.

Destination

The `Destination` entity stores the configuration information required by the SRF connector to locate and communicate with the host system.

SRF Connector

The SRF connector is responsible for delivering data to and receiving data from the host system.

Store and Forward

The Store and Forward mechanism operates between the Siebel Retail Finance mid-tier (that is, the SRF Components) and the host. The Financial Process Integrator's Store and Forward framework provides the means to store transactions, in the event of a host going offline, in order to forward them to the host at a later time. It enables the storing of data for update to the host, it does not store data retrieved from the host.

Interaction of Financial Process Integrator Components

An outline of how the Financial Process Integrator components typically interact, and how the Financial Process Integrator interacts with SRF Financial Components follows.

- The client makes a request of a Session EJB.
- This request is routed to the session bean by MCA Services.
- To fulfil the request the session bean must retrieve data from the host.
- All data on the host is modelled as entity EJBs so the session bean must retrieve an entity EJB instance.
- The entity bean must populate itself with data from the host system. It delegates this task to the persister.
- The persister checks if the data is already in the cache either directly by primary key, or through an index, if one is configured. If data is found in the cache, the persister populates the entity bean with the cached data, otherwise it sends a request to the host for the required data.
- The Financial Process Integrator retrieves the request metadata for the specified request, and passes the request and the associated metadata to the Data Formatter.
- The Data Formatter uses the metadata to transform the request into a format the host can understand.
- The Financial Process Integrator retrieves the Transaction Route information for the specified request, and locates the appropriate SRF connector.
- If the SRF connector is not already initialized, it initializes itself using the Destination information.
- The SRF connector then passes the formatted request to the host system (using the host's middleware technology such as CICS or MQ).
- The host processes the request and returns a response.
- The SRF connector passes the response back to the Financial Process Integrator.
- The Financial Process Integrator retrieves the response metadata and passes the response and associated metadata to the Data Formatter.
- The Data Formatter uses the metadata to transform the response from the host into a format the persister can understand.
- The Financial Process Integrator returns the response to the persister.
- The persister populates the entity bean with the returned data, and stores the data in the cache. The response data in the cache can also be indexed.
- The session bean returns the response from the entity bean to the client.

Financial Process Integrator Metadata

The metadata defines the structure of the data that is sent to the host system for a transaction, and the data returned from the host system, to a transaction request. The metadata is composed of transaction fields. Each transaction field represents an individual block of data in the host system

data. To form a transaction request, all the appropriate transaction fields are extracted from the metadata and combined. To process a transaction response, the transaction fields that are part of that type of transaction are extracted from the metadata and used to process and extract information from the host system response.

Separation of Request and Response

There are two sets of data that are represented by the Financial Process Integrator metadata:

- The structure of the host system request, which will be of a host-specific format.
- The mapping of host system response fields to response [DataPacket](#) fields.

These two sets of data are represented separately.

The request metadata specifies the sequential transaction fields required for the host system request. Values necessary for the transaction are extracted from the [DataPacket](#) transaction request.

In the case of the host system response, the metadata specifies the mappings of result [DataPackets](#) to the host system response data. Only the required fields are extracted from the host system response. The required fields are referenced in the host system response by offset; the entire host system response does not have to be parsed, only the required fields.

Support for Error Conditions

The Financial Process Integrator has support for error condition responses from the host system. The Financial Process Integrator can determine if the host system response is an error response from the host system.

Metadata Response Access by Offset

The Financial Process Integrator metadata for the host system response specifies the location of required transaction fields by offset rather than by sequence. Only the required transaction fields are parsed out of the host system response. This means that if only one field is required in a host system response of one hundred fields, only that one field is parsed and the additional fields are not read. This offset mechanism optimises performance.

Request Transaction Fields

The Financial Process Integrator creates transaction requests using the [RequestTransactionField](#) entity `com.bankframe.ei.txnhandler.transactionlayout.impl.request.RequestTransactionFieldBean`. This entity maps to the [REQUEST_TXN_LAYOUT](#) database table.

The main body of a transaction request, which is passed to the Siebel Retail Finance Connector, is built by determining all the necessary transaction fields for the required transaction request. The [REQUEST_TXN_LAYOUT](#) database table specifies the transaction fields necessary for host system requests. Each row of the [REQUEST_TXN_LAYOUT](#) database table represents a transaction field.

A bank's COBOL Copybook is a typical source for determining the necessary entries in the request metadata. Each transaction field, as defined in the COBOL Copybook, must be defined in the metadata to correctly form a host system request. Typically the [TXN_CODE](#) is the [NAME](#) section of the COBOL Copybook.

The total number and type of columns in the `REQUEST_TXN_LAYOUT` table depends on the host system requirements, and may be customized for a specific host.

The main columns in the `REQUEST_TXN_LAYOUT` table are as follows:

- The transaction code (`TXN_CODE`) specifies the transaction id as defined on the host system. This is an alphanumeric string that uniquely identifies the transaction.
- The transaction type (`TXN_TYPE`) indicates which host system the transaction request is being passed to.
- The `FIELDNAME` element identifies the transaction field as defined for the host system.
- The `SEQUENCE` element specifies the order in which the transaction fields are ordered for sending to, and receiving from, the host system. This element starts at 1.
- The `LENGTH` element is the length of the transaction field as required by the host system.
- `DP_FIELD` defines the name of the field in the Request `DataPacket` that maps to the request transaction field `FIELDNAME`.
- The `MANDATORY` element specifies if the request `DataPacket`, passed to the Financial Process Integrator, must contain an element called `DP_FIELD` with a value to place in the transaction field. The `MANDATORY` element has the value `yes` or `no`. An Exception is thrown if a transaction request `DataPacket` passed to the Financial Process Integrator does not specify a value for a mandatory element, for example, `CUSTOMER_NAME` for a customer details request, this element would likely be mapped to `CUST-NAME` in the host system data request and would be a mandatory element in the `DataPacket` for this type of request.
- The `DATA` element is a default value for the transaction field which is passed to the host system.
- The `FIELD_PAD_CHAR` element specifies the padding character to fill the transaction field data with if the data is less than `LENGTH`.
- The `FIELD_ALIGN` element specifies the alignment of padding data in the transaction field. Valid values are `LEFT` or `RIGHT`.
- The `FIELD_ENCODING` element specifies the encoding used to format the host system data. Examples for textual data are `ASCII`, `EBCDIC`. Examples for numeric data are the Cobol types `COMP-3` and `STD`.
- The `ISSIGNED_FIELD` element specifies if the transaction field is signed.
- The `DEC_BEFORE` element specifies the number of places before the decimal point for numeric data.
- The `DEC_AFTER` element specifies the number of places after the decimal point for numeric data.

The Financial Process Integrator passes all transaction processing duties to the `BasicDataFormat` class. The `BasicDataFormat` class calls the `RequestTransactionField` entity bean home method `findByTransactionCodeAndType(txnCode, txnType)` to get the appropriate transaction fields required for the transaction request being processed by the Financial Process Integrator. This method returns a `List` of `RequestTransactionField` entity beans which are accessed using the interface `TransactionField`.

Example Transaction Request

The `CustomerSearch` example `findByLastName` operation has a transaction request defined by the following Cobol Copybook:

```
000400 01  MAIN-CUSTOMERSEARCH.  
  
001400* INPUT DATA  
  
001600    05 T-CODE                PIC X(12).  
  
001800    05 T-RESTART-INDEX      PIC X(4).  
  
002000    05 C-LAST-NAME          PIC X(20).
```

The request transaction fields for this transaction are defined in Table 4. REQUEST_TXN_LAYOUT .

Table 4. REQUEST_TXN_LAYOUT table

TXN_CODE	TXN_TYPE	FIELDNAME	SEQUENCE	LENGTH	DP_FIELD	MANDATORY
TESTFIND0002	TEST	T-CODE	1	12	TXN_CODE	YES
TESTFIND0002	TEST	T-RESTART-INDEX	2	4	RESTART_INDEX	NO
TESTFIND0002	TEST	C-LAST-NAME	3	20	LAST_NAME	YES

The transaction is identified with [TXN_CODE=TESTFIND0002](#) and the host system is defined as [TXN_TYPE=TEST](#). There are three transaction fields defined according to the Cobol Copybook definition:

- [T-CODE](#). This transaction field is mapped to [TXN_CODE](#) in the request [DataPacket](#) which is passed to the Financial Process Integrator. This field is mandatory in the request [DataPacket](#).
- [T-RESTART-INDEX](#). This transaction field is mapped to [RESTART_INDEX](#) in the request [DataPacket](#) which is passed to the Financial Process Integrator. This field is not mandatory in the request [DataPacket](#). The field is used for maintaining an index while making repeated calls to the host system for results.
- [C-LAST-NAME](#). This transaction field is mapped to [LAST_NAME](#) in the request [DataPacket](#) which is passed to the Financial Process Integrator. This field is mandatory in the request [DataPacket](#).

Processing Host System Response

The [BasicDataFormat](#) class determines the transaction fields in the host system response necessary for the transaction response by using the following steps:

- 1 The mapping of entity [DataPackets](#) elements to required transaction fields in the host system response is specified by the [TransactionMetaData](#) entities.
- 2 The form of the transaction fields in the host system response data that are required for step 1 are specified by the [ResponseTransactionField](#) entities.

Response Metadata Mapping

The response from a host system has to be converted from the host system specific format into entity results which are passed to the persister, which calls the Financial Process Integrator, as [DataPackets](#). Therefore, the first step in extracting the necessary result data from the host system response is to determine which elements are necessary for the [DataPacket](#) result, and map these required elements to transaction fields in the host system response. For example, a [Customer](#) entity might make a request to the Financial Process Integrator, via the persister, to obtain the customer name and ID from the host system. The result [DataPacket](#) would have to contain the elements

`CUSTOMER_NAME` and `CUSTOMER_ID`. These elements in the result `DataPacket` would be mapped to the host system response fields `CUST-ID` and `CUST_NAME`.

The `BasicDataFormat` class determines the required mappings using the `TransactionMetaData` entity:

```
com.bankframe.ei.txnhandler.transactionresponse.metadata.MetaDataBean
```

This entity maps to Table 5. `RESPONSE_META_DATA`.

Table 5. `RESPONSE_META_DATA`

TXN_CODE	TXN_TYPE	DP_NAME	DP_FIELD	TXN_FIELDNAME	DP_INDEX	DP_PK_FIELD	DEFAULT_VALUE
TXN01	TYPE1	ACCOUNT_INFO	ACCOUNT_NUMBER	Account_Info[0].ACCOUNT_NUMBER	1	No	defaultValue
TXN01	TYPE1	ACCOUNT_INFO	ACCOUNT_NUMBER	Account_Info[0].Account_Number	1	Yes	defaultValue
TXN01	TYPE1	ACCOUNT_INFO	ACCOUNT_NUMBER	Account_Info[1].ACCOUNT_NUMBER	2	No	defaultValue
TXN01	TYPE1	ACCOUNT_INFO	ACCOUNT_NUMBER	Account_Info[1].Account_Number	2	Yes	defaultValue

The columns in the `RESPONSE_META_DATA` table are as follows:

- `TXN_CODE` specifies the transaction ID as defined on the host system, this is an alphanumeric string that uniquely identifies the transaction.
- `TXN_TYPE` indicates which host system the transaction request is being passed to.
- The `DP_NAME` element specifies the name of the entity bean that a response from the host system belongs to, for example, `TestBean`.
- The `DP_FIELD` element identifies the field member name in the entity bean that the result maps to.
- The `DP_INDEX` element identifies the entity that the response value belongs to. This is used to uniquely store each entity result returned from the host system. This number must be greater than or equal to 1.
- The `DP_PK_FIELD` element determines if the field is an element of the primary key for the entity object that is being mapped to. The entity primary key may consist of several elements constructed from the host system response data during processing. If `DP_PK_FIELD` is set to `Yes` then the field is a primary key element for the entity result.
- The `TXN_FIELDNAME` element identifies the transaction field in the `RESPONSE_TXN_LAYOUT` table that the metadata element maps to.
- The `DEFAULT_VALUE` element specifies a default value for the field.

The Financial Process Integrator passes all response processing duties to the `BasicDataFormat` class. The `BasicDataFormat` class calls the `TransactionMetaData` entity bean home method

`findByTransactionCodeAndType (txnCode, txnType)` to get the required transaction field mappings for the transaction being processed. This method returns a `List` of `TransactionMetaDatum` entity beans.

Response Transaction Fields

Once the required mappings from entity `DataPacket` elements to transaction fields have been determined, it is necessary to obtain the form of each transaction field to be extracted from the host system response. The Financial Process Integrator determines the form of the required response transaction fields using the `ResponseTransactionField` entity `com.bankframe.ei.txnhandler.transactionlayout.impl.response.ResponseTransactionFieldBean`. This entity maps to the `RESPONSE_TXN_LAYOUT` table. The total number and type of columns in the `RESPONSE_TXN_LAYOUT` table depends on the host system requirements and can be customized for a specific host. The main columns in the `RESPONSE_TXN_LAYOUT` table are as follows:

- The `FIELDNAME` element identifies the transaction field as defined for the host system.
- The `OFFSET` element specifies the offset of the transaction field in the host system data.
- The `LENGTH` element is the length of the transaction field in the host system data.
- The `FIELD_PAD_CHAR` element specifies the padding character to fill the transaction field data with if the data is less than `LENGTH`.
- The `FIELD_ALIGN` element specifies the alignment of padding data in the transaction field. Valid values are `LEFT` and `RIGHT`.
- The `FIELD_ENCODING` element specifies the encoding used to format the host system data. Examples for textual data are `ASCII` and `EBCDIC`. Examples for numeric data are the Cobol types `COMP-3` and `STD`.
- The `ISSIGNED_FIELD` element specifies if the transaction field is signed.
- The `DEC_BEFORE` element specifies the number of places before the decimal point for numeric data.
- The `DEC_AFTER` element specifies the number of places after the decimal point for numeric data.

The `BasicDataFormat` class creates a `Map` of `ResponseTransactionField` entity beans. The `BasicDataFormat` class processes the metadata mappings using necessary `ResponseTransactionField` entities from the `Map`. The transaction field data is extracted from the host system data using the `ResponseTransactionField`. The `ResponseTransactionField` entity beans are accessed using the interface `TransactionField`.

Each of the transaction fields defined in this table must have a unique name and therefore it may be necessary to append the `TXN_CODE` and `TXN_TYPE` to the name of the field where many transactions might be defined in the metadata. The naming convention therefore for transaction fields is `TXN_CODE-TXN_TYPE-GROUP_NAME[INDEX]-FIELD_NAME-OFFSET`.

Caching the Metadata (Transaction Fields)

To improve performance the Financial Process Integrator metadata can be cached. The `transactionHandler.metadata.cache` entry in the `BankframeResource.properties` file specifies whether caching of metadata is used by the Financial Process Integrator. This is either `true` or `false`.

This caching applies to the `RequestTransactionField`, `ResponseTransactionField`, `ResponseMetaData` and `ResponseErrorCondition` entities. If metadata caching is enabled then metadata is obtained from the database tables and stored to memory for quick access. The metadata elements are accessed through the same interface as the entity beans.

The Financial Process Integrator uses the MCA generic caching framework for caching of metadata.

It may be necessary for the `BasicDataFormat` class to determine if caching is enabled. The `BasicDataFormat` class can determine this using the following method:

```
boolean metaDataCached =
com.bankframe.ei.txnhandler.TransactionHandlerUtils.isMetaDataCached();
```

TransactionField Interface

The `BasicDataFormat` class interacts with the `RequestTransactionField` and `ResponseTransactionField` entity beans through the interface `com.bankframe.ei.txnhandler.transactionlayout.TransactionField`.

The remote interface of these entity beans uses the same interface as the caching mechanism, allowing the entity beans and cached entities to be accessed in the same manner.

The `TransactionField` interface is defined as follows:

```
public interface TransactionField {

    public String getValue(String colName) throws ProcessingErrorException,
RemoteException;

    public Map getValuesMap() throws ProcessingErrorException, RemoteException;

}
```

The generic method `getValue(String colName)` allows the `RequestTransactionField` entity bean to work against a `REQUEST_TXN_LAYOUT` database table with any combination of database columns. This is necessary to avoid recoding of the `RequestTransactionField` entity bean for each host system, as each host system may require a different definition of the `REQUEST_TXN_LAYOUT` database table. The same applies to the `ResponseTransactionField` entity with the `RESPONSE_TXN_LAYOUT` database table.

The argument to the `getValue(String colName)` method specifies the column name in the database table. The method returns the value of the specified column as a `java.lang.String`. This value has to be converted to the correct type. See the following `BasicDataFormat` code example for obtaining a `String` entry and an `int` value from a previously obtained transaction field:

```
Transaction txnField;

int fieldLength = new Integer(txnField.getValue("LENGTH")).intValue();

String dataPacketField = txnField.getValue("DP_FIELD");
```

The method `getValuesMap()` returns the `java.util.Map` interface to all the column elements. The keys to entries in the `Map` are the column names. The `Map` values are

`com.bankframe.ei.txnhandler.transactionlayout.HashTableElement` objects describing the value of the database column.

Response Mapping Sample Implementation

The following Account COBOL copybook would be represented as defined in Table 6.

`RESPONSE_TXN_LAYOUT.`

```
05 Account_Info occurs 2.

010 Account_Number Pic X(10).

010 Account_Name Pic X(10).
```

Table 6. `RESPONSE_TXN_LAYOUT`

FIELDNAME	OFFSET	LENGTH	DATA	FIELD_PAD_CHAR
Account_Info[0].Account_Number	0	10		0
Account_Info[0].Account_Name	10	10		' '
Account_Info[1].Account_Number	20	10		0
Account_Info[1].Account_Name	30	10		' '

...

FIELD_ALIGN	FIELD_ENCODING	ISSIGNED_FIELD	DEC_BEFORE	DEC_AFTER
LEFT	COMP	0	10	0
RIGHT	ASCII	0	0	0
LEFT	COMP	0	10	0
RIGHT	ASCII	0	0	0

A `DataPacket` called `ACCOUNT_INFO` that contains the fields `ACCOUNT_NAME` and `ACCOUNT_NUMBER` would be mapped to the above COBOL copybook using the metadata defined in Table 7.

`RESPONSE_META_DATA.`

Table 7. `RESPONSE_META_DATA`

TXN_CODE	TXN_TYPE	DP_NAME	DP_FIELD	TXN_FIELDNAME	DP_INDEX	DP_PK_FIELD	DEFAULT_VALUE
TXN01	TYPE1	ACCOUNT_INFO	ACCOUNT_NUMBER	Account_Info[0].ACCOUNT_NUMBER	1	No	defaultValue
TXN01	TYPE1	ACCOUNT_	ACCOUNT_	Account_Info[0].	1	Yes	defaultValue

TXN_CODE	TXN_TYPE	DP_NAME	DP_FIELD	TXN_FIELDNAME	DP_INDEX	DP_PK_FIELD	DEFAULT_VALUE
		INFO	NUMBER	Account_Number			
TXN01	TYPE1	ACCOUNT_INFO	ACCOUNT_NUMBER	Account_Info[1].ACCOUNT_NUMBER	2	No	defaultValue
TXN01	TYPE1	ACCOUNT_INFO	ACCOUNT_NUMBER	Account_Info[1].Account_Number	2	Yes	defaultValue

- The [TXN_CODE](#) and [TXN_TYPE](#) define what transaction the mapping belongs to.
- [DP_NAME](#) defines the name of the [DataPacket](#) that the persister expects as a result for this transaction.
- [DP_FIELD](#) defines the name of the field in the [DataPacket](#) result.
- [TXN_FIELDNAME](#) defines the name of the field in the [RESPONSE_TXN_LAYOUT](#) table that this result element maps to.
- The [DP_INDEX](#) value specifies the index of the result entity that the element belongs to. The index always starts at 1.
- The [DP_PK_FIELD](#) column determines if the field is a primary key field in the result entity. [ACCOUNT_NUMBER](#) is the primary key for the entities in the above Account example.

Support for Tiered Fields

Cobol copybooks often define a hierarchy or grouping of fields that should not be imposed on SRF entity beans when mapping from entities to cobol copybooks. The fields in the following sample hierarchicahl cobol copybook map to two instances of an [Account](#) entity.

```
05 Card_Number Pic X(10).

05 Account_Info occurs 2.

    010 Account_Number Pic X(10).

    010 Account_Name Pic X(10).
```

To map this data you need to create two [Account DataPackets](#), and to treat [Card_Number](#) as if it belongs to the [Account_Info](#) tier, that is, the [Card_Number](#) field will occur in both [Account DataPackets](#). Table 8 illustrates how the database table would be defined for this hierarchical cobol copybook mapping.

Table 8. [RESPONSE_TXN_LAYOUT](#)

FIELDNAME	OFFSET	LENGTH	FIELD_PAD_CHAR	FIELD_ALIGN
Card_Number	0	10	0	LEFT

FIELDNAME	OFFSET	LENGTH	FIELD_PAD_CHAR	FIELD_ALIGN
Account_Info[0].Account_Number	10	10	0	LEFT
Account_Info[0].Account_Name	20	10	' '	RIGHT
Account_Info[1].Account_Number	30	10	0	LEFT
Account_Info[1].Account_Name	40	10	' '	RIGHT

...

FIELD_ENCODING	ISSIGNED_FIELD	DEC_BEFORE	DEC_AFTER
COMP	0	10	0
COMP	0	10	0
ASCII	0	0	0
COMP	0	10	0
ASCII	0	0	0

- The [Card_Number](#) is defined once for the host system data.
- A group of entries is put in the [RESPONSE_TXN_LAYOUT](#) for each instance of the group [Account_Info](#). The name of these group fields starts with the group name and index of the group occurrence, for example, [Account_Info\[0\]](#) is the first occurrence of the group in the host system data.

Table 9. [RESPONSE_META_DATA](#) defines how these fields are mapped to the [DataPackets](#) entity.

Table 9. [RESPONSE_META_DATA](#)

TXN_CODE	TXN_TYPE	DP_NAME	DP_FIELD	TXN_FIELDNAME	DP_INDEX	DP_PK_FIELD	DEFAULT_VALUE
TXN01	TYPE1	ACCOUNT_INFO	ACCOUNT_NAME	Account_Info[0].Account_Name	1	No	defaultValue
TXN01	TYPE1	ACCOUNT_INFO	ACCOUNT_NUMBER	Account_Info[0].Account_Number	1	Yes	defaultValue
TXN01	TYPE1	ACCOUNT_INFO	CARD_NUMBER	Card_Number	1	No	defaultValue
TXN01	TYPE1	ACCOUNT_INFO	ACCOUNT_NAME	Account_Info[1].Account_Name	2	No	defaultValue
TXN01	TYPE1	ACCOUNT_INFO	ACCOUNT_NUMBER	Account_Info[1].Account_Number	2	Yes	defaultValue
TXN01	TYPE1	ACCOUNT_INFO	CARD_NUMBER	Card_Number	2	No	defaultValue

Since each field in the host system data is given an individual explicit name, it is easy to map from any `DataPacket` element to any transaction field in the host system data.

Deeply Nested Cobol Copybooks

Table 10 illustrates how the sample deeply nested Customer Details cobol copybook is defined in the `RESPONSE_TXN_LAYOUT` table, to map to a single fault entity bean.

```
01 Customer_Details.

    02 Customer_Number Pic X(10).

    02 Last_Name Pic X(10).

    02 First_Name Pic X(10).

    02 Contact_Details.

        05 Best_Contact_Time Pic X(10).

        05 Preferred_Contact Pic X(10).

        05 Work_Details.

            010 Employer_Name Pic X(10).

            010 Phone_No Pic X(10).

        05 Home_Details.

            010 Phone_No Pic X(10).

            010 Home_Address Pic X(20).
```

Table 10. `RESPONSE_TXN_LAYOUT`

FIELDNAME	OFFSET	LENGTH	Fill Char	...
Customer_Details.Customer_Number	0	10	0	
Customer_Details.Last_Name	20	10	' '	
Customer_Details.First_Name	30	10	' '	
Customer_Details.Contact_Details.Best_Contact_Time	40	10	' '	
Customer_Details.Contact_Details.Preferred_Contact	50	10	' '	
Customer_Details.Contact_Details.Work_Details. Employer_Name	60	10	' '	
Customer_Details.Contact_Details.Work_Details. Phone_No	70	10	' '	

FIELDNAME	OFFSET	LENGTH	Fill Char	...
Customer_Details.Contact_Details.Home_Details. Phone_No	80	10	' '	
Customer_Details.Contact_Details.Home_Details. Home_Address	90	20	' '	

The sample Customer Details cobol copybook would use the definition in Table 11.

[RESPONSE_META_DATA](#) to map to a [DataPacket](#) with the following fields:

[CUSTOMER_NUMBER](#)

[LAST_NAME](#)

[FIRST_NAME](#)

[BEST_CONTACT_TIME](#)

[PREFERRED_CONTACT_METHOD](#)

[EMPLOYER_NAME](#)

[WORK_PHONE_NO](#)

[HOME_ADDRESS](#)

[HOME_PHONE_NO](#)

Table 11. [RESPONSE_META_DATA](#)

TXN_CODE	TXN_TYPE	DP_NAME	DP_FIELD
TXN01	TYPE1	CUSTOMER_DETAILS	CUSTOMER_NUMBER
TXN01	TYPE1	CUSTOMER_DETAILS	LAST_NAME
TXN01	TYPE1	CUSTOMER_DETAILS	FIRST_NAME
TXN01	TYPE1	CUSTOMER_DETAILS	BEST_CONTACT_TIME
TXN01	TYPE1	CUSTOMER_DETAILS	PREFERRED_CONTACT_METHOD
TXN01	TYPE1	CUSTOMER_DETAILS	EMPLOYER_NAME
TXN01	TYPE1	CUSTOMER_DETAILS	WORK_PHONE_NO
TXN01	TYPE1	CUSTOMER_DETAILS	HOME_ADDRESS
TXN01	TYPE1	CUSTOMER_DETAILS	HOME_PHONE_NO

TXN_FIELDNAME	DP_INDEX	DP_PK_FIELD
Customer_Details.Customer_Number	1	Yes
Customer_Details.Last_Name	1	No
Customer_Details.First_Name	1	No
Customer_Details.Contact_Details.Best_Contact_Time	1	No
Customer_Details.Contact_Details.Preferred_Contact	1	No
Customer_Details.Contact_Details.Work_Details.Employer_Name	1	No
Customer_Details.Contact_Details.Work_Details.Phone_No	1	No
Customer_Details.Contact_Details.Home_Details.Home_Address	1	No
Customer_Details.Contact_Details.Home_Details.Phone_No	1	No

Each cobol field has its own name therefore it can be easily mapped to any entity bean layout.

Mapping a Subset of Transaction Fields

This is one situation that this solution makes easy. Taking the previous example, if instead of mapping all the fields in the copybook we're only interested in mapping:

[CUSTOMER_NUMBER](#)

[CUSTOMER_LAST_NAME](#)

[CUSTOMER_FIRST_NAME](#)

[HOME_ADDRESS](#)

For this case only define the mappings for those fields in the [RESPONSE_META_DATA](#) table, don't add mappings for the other fields. If the transaction fields are not required by any entity then eliminate the fields from the [RESPONSE_TXN_LAYOUT](#) table.

Padding or "Filler" fields are not required, for example, to deal with the gap between the [CUSTOMER_FIRST_NAME](#) field and the [HOME_ADDRESS](#) field in the host system data. Transaction fields are extracted by their [OFFSET](#), and not a sequence number, therefore only the necessary fields have to be processed.

Recurring Fields

The host system data may contain recurring fields as follows:

```
05 Address_Details
    010 Street_Address Pic X(10) occurs 3
    010 State Pic X(2)
    010 Postcode Pic X(5)
```

For the system in question this has to be mapped to a single [Address](#) Entity. The entity members must be mapped to the [Street_Address](#) field. Entity beans cannot have array fields therefore three separate fields in the entity bean need to be defined to represent each entry in the array, for example:

[STREET_ADDRESS1](#)

[STREET_ADDRESS2](#)

[STREET_ADDRESS3](#)

The above fields need to be mapped to the correct transaction fields, as listed in Table 12 .

Table 12. Transaction Field Mappings

TXN_CODE	TXN_TYPE	DP_NAME	DP_FIELD	TXN_FIELDNAME	DP_INDEX	DP_PK_FIELD	DEFAULT_VALUE
TXN01	TYPE1	ADDRESS	STATE	Address_Details.State	1	No	default
TXN01	TYPE1	ADDRESS	POSTCODE	Address_Details.Postcode	1	No	default
TXN01	TYPE1	ADDRESS	STREET_ADDRESS1	Address_Details.Street_Address[0]	1	No	default
TXN01	TYPE1	ADDRESS	STREET_ADDRESS2	Address_Details.Street_Address[1]	1	No	default
TXN01	TYPE1	ADDRESS	STREET_ADDRESS3	Address_Details.Street_Address[2]	1	No	default

Handling Error Conditions

To determine if the host system response data is an error response the [BasicDataFormat](#) class must analyse the host system data for transaction field values that indicate that the response data is error data. The [TransactionErrorCondition](#) entity provides the information necessary for the [BasicDataFormat](#) class to determine if the host system data is an error result.

[TransactionErrorCondition](#) entity maps to Table 13. [RESPONSE_ERROR_CONDITION](#):

Table 13. [RESPONSE_ERROR_CONDITION](#)

TXN_CODE	TXN_TYPE	SEQUENCE	TXN_FIELDNAME	CONDITION	VALUE
ACCOUNTFIND	TEST	1	Error-Flag	EQUALS	'TRUE'
ACCOUNTFIND	TEST	2	Error-Type	NOT_EQUALS	' '

COMBINE_WITH_NEXT	ERROR_TXN_CODE	ERROR_TXN_TYPE
AND	ACCFIND_ERROR	TEST
NO	ACCFIND_ERROR	TEST

A description of each Response Error Condition field is provided in Table 14.

Table 14. Response Error Condition Fields

Field	Description
TXN_CODE	Defines the transaction the error condition applies to.
TXN_TYPE	Defines the transaction the error condition applies to.
SEQUENCE	Determines the order in which the error-conditions are used to determine if a host system response is an error. SEQUENCE starts at 1.
TXN_FIELDNAME	Defines the name of the transaction field in the host system response that is tested, the transaction field being defined in the RESPONSE_TXN_LAYOUT table.
CONDITION	<p>Defines the condition that must be met to indicate an error, this column can have the following values:</p> <p>EQUALS. The value of the TXN_FIELDNAME must match the VALUE column exactly.</p> <p>STARTS_WITH. The value of the TXN_FIELDNAME must start with the string defined in the VALUE column.</p> <p>ENDS_WITH. The value of the TXN_FIELDNAME must end with the string defined in the VALUE column.</p> <p>CONTAIN. The value of the TXN_FIELDNAME must contain the string defined in the VALUE column somewhere in its contents.</p> <p>NOT_EQUAL. Reverse of EQUALS.</p> <p>NOT_START_WITH. Reverse of STARTS_WITH.</p>
VALUE	Specifies the value to compare the transaction field value to. If the CONDITION is ' EQUALS ' then the VALUE must be the same length as the LENGTH specified in the RESPONSE_TXN_LAYOUT table for the transaction field. The VALUE for Error-Type in the above sample has to specify 20 spaces as the transaction field Error-Type defined in RESPONSE_TXN_LAYOUT table has a LENGTH of 20 bytes.
COMBINE_WITH_NEXT	<p>Allows for combinations of error tests on the host system data. The logical tests can not be complex nested logical tests, only direct combinations as follows:</p> <p>AND. The result of this error test will be logically AND'd with the next error test; If the error is true and the next error is true then the combined error</p>

Field	Description
	<p>result is true.</p> <p>OR. The result of this error test will be logically OR'd with the next error test; If this error is true or the next error is true then the combined error result is true.</p> <p>XOR. The result of this error test will be logically Exclusively OR'd with the next error test; If this error is true or the next error is true, but both are not true, then the combined error result is true.</p> <p>No. The result of this error test will not be combined with the next error test; This is used for the last error test only, otherwise the error result will not be combined in the next or final result.</p> <p>` `. The result of this error test will be the same as No</p>

- The **ERROR_TXN_CODE** and **ERROR_TXN_TYPE** allow the error condition to specify a specific metadata format for the parsing of the remainder of the error result from the host system. The remainder of the error result may contain error information specific to that error result which has to be parsed and returned in a **ProcessingErrorException** to the user.
- The **BasicDataFormat** method **handleHostSystemError()** is over-ridden to specify what action to take when it has been determined that an error has occurred. This may involve parsing the remainder of the host system data using the error transaction meta data, defined by **ERROR_TXN_CODE** and **ERROR_TXN_TYPE**, to extract further error information from the host system response and/or throwing a **ProcessingErrorException**.
- The **ERROR_TXN_CODE** and **ERROR_TXN_TYPE** need not be specified or can be the same as the **TXN_CODE** and **TXN_TYPE** of the transaction currently being processed. This allows the **BasicDataFormat** method **handleHostSystemError()** method to use the metadata of the current transaction to be used to extract the remainder of the host system response if required.

Sample Error Condition Implementation

For demonstration purposes a transaction with **TXN_CODE=TESTFIND** and **TXN_TYPE=TEST** has a host system response defined by the following Cobol copybook:

```
000400 01  MAIN-ACCOUNTFIND.
000410 010  ERROR-FLAG      PIC X(5).
000420 010  ERROR-TYPE     PIC X(20).
001300 010  CARD-NUMBER    PIC 9(5).
001500      05  ACCOUNT-INFO OCCURS 10 TIMES.
001700      010  ACCOUNT-NUMBER PIC 9(5).
```

This results in a transaction defined with the following entries in Table 15. **RESPONSE_TXN_LAYOUT**:

Table 15. **RESPONSE_TXN_LAYOUT**

FIELDNAME	OFFSET	LENGTH	Data	Fill Char	...
Error-flag	0	5	FALSE	0	

FIELDNAME	OFFSET	LENGTH	Data	Fill Char	...
Error-Type	5	20	' '	' '	
Card-Number	25	5		' '	
...					
...					

If it was determined that an error was indicated by the field `ERROR-FLAG` having a value equal to `TRUE` and `ERROR-TYPE` not being empty then the designer creates two entries in Table 16. `RESPONSE_ERROR_CONDITION` like:

Table 16. `RESPONSE_ERROR_CONDITION`

TXN_CODE	TXN_TYPE	SEQUENCE	TXN_FIELDNAME	CONDITION	VALUE
ACCOUNTFIND	TEST	1	Error-Flag	EQUALS	'TRUE'
ACCOUNTFIND	TEST	2	Error-Type	NOT_EQUALS	' '

COMBINE_WITH_NEXT	ERROR_TXN_CODE	ERROR_TXN_TYPE
AND	ACCOUNTFIND	TEST
NO	ACCOUNTFIND	TEST

NOTE: The length of the `VALUE` field must be equal to the length of the field specified in the cobol copybook, that is, the host response field length, the `VALUE` for the `ERROR-TYPE` field must be 20 bytes in `RESPONSE_ERROR_CONDITION`.

The designer then has to determine what the remainder of the error response contains. The designer will implement the `BasicDataFormat` method `handleHostSystemError()` to handle the error. This might involve immediately throwing a `ProcessingErrorException` or might involve parsing the remainder of the response to extract error information to fill the `ProcessingErrorException` with useful information.

The `RESPONSE_ERROR_CONDITION` elements `ERROR_TXN_CODE` and `ERROR_TXN_TYPE` specify what response fields and response metadata to use to parse the error host response.

NOTE: The values of these two elements can be the same as the `TXN_CODE` and `TXN_TYPE` of the transaction that called the host in which case the current response fields and response metadata are used to extract information from the host system response.

So it might be determined that the remainder of the error response is described the following Cobol copybook:

```
002020 01 HOST-SYSTEM-ERROR.
002030     05 ERROR-CODE      PIC 9(5).
002040     05 ERROR-MESSAGE  PIC X(30).
```

Therefore this metadata is entered in the response fields and response metadata tables and given the transaction code and type: `TXN_CODE=ACCOUNTFIND_ERR` and `TXN_TYPE=TEST`.

Now Table 17. `RESPONSE_ERROR_CONDITION`, is updated to contain the following:

Table 17. `RESPONSE_ERROR_CONDITION`

TXN_CODE	TXN_TYPE	SEQUENCE	TXN_FIELDNAME	CONDITION	VALUE
ACCOUNTFIND	TEST	1	Error-Flag	EQUALS	'TRUE'
ACCOUNTFIND	TEST	2	Error-Type	NOT_EQUALS	' '

COMBINE_WITH_NEXT	ERROR_TXN_CODE	ERROR_TXN_TYPE
AND	ACCOUNTFIND_ERR	TEST
NO	ACCOUNTFIND_ERR	TEST

Now the `BasicDataFormat` method `handleHostSystemError()` is coded to get the metadata for `TXN_CODE=TESTFIND_ERR` and `TXN_TYPE=TEST` and processes the response extracting the `ERROR-CODE` and `ERROR-MESSAGE`. The method creates a `ProcessingErrorException` containing the error information, that is, "Error processing transaction, host system error code: 1000, host system error message: ACCOUNT-NUMBER invalid"

Some systems embed the error information in the original transaction response, in that each field in the host response is appended with an error-flag field and therefore the same metadata is used for processing the error response as the normal response.

For example a host response may be defined by the following Cobol copybook:

```
000400 01  MAIN-ACCOUNTFIND.
000410 010  ERROR-FLAG      PIC X(5).
001300 010  CARD-NUMBER     PIC 9(5).
001300 010  CARD-NUMBER-ERR PIC X(5).
001500 05  ACCOUNT-INFO OCCURS 10 TIMES.
001700 010  ACCOUNT-NUMBER  PIC 9(5).
001700 010  ACCOUNT-NUMBER-ERR PIC X(5).
```

Each value field in the above transaction definition is followed by an error flag field. The error flag fields are `CARD-NUMBER-ERR` and `ACCOUNT-NUMBER-ERR`. The host system during processing marks the value field that caused an error by setting the corresponding error-flag field to `TRUE`.

In this case the designer codes the `handleHostSystemError()` method to use the original metadata for the transaction to parse the remainder of the transaction response as normal. The code then determines which field in the response is causing the error by checking each error field, `CARD-NUMBER-ERR` and `ACCOUNT-NUMBER-ERR`.

The error flag field that has a value `TRUE` is shown in the resulting `ProcessingErrorException`. The host system determined that the `ACCOUNT-NUMBER` is invalid therefore `ACCOUNT-NUMBER-ERR=" TRUE"` and the `ProcessingErrorException` is created containing the information "Error processing transaction, host system error field: ACCOUNT-NUMBER".

Notes:

- If `ERROR_TXN_CODE` and `ERROR_TXN_TYPE` are equal to `TXN_CODE` and `TXN_TYPE` respectively then the original metadata is used to process the remainder of the host response.
- The `RESPONSE_ERROR_CONDITION` table only allows specification of one form of error for each transaction code and type. Only one form of checking for an error condition, checking `ERROR_FLAG` and `ERROR_TYPE` in the example above. And only one form of error response metadata, that is, `ERROR_CODE` and `ERROR_MESSAGE` in the example above. This should suffice as the error can contain any error message and therefore theoretically handle any error.
- error-conditions functionality only handles simple logic combinations of error condition fields, no nested combinations of error condition fields.
- The last error-condition field checked for a given transaction code and type determines the `ERROR_TXN_CODE` and `ERROR_TXN_TYPE` to use, that is, the error-condition with the last `SEQUENCE`.

Transaction Field Naming

The names used for `FIELDNAME`, in the `RESPONSE_TXN_LAYOUT` table can be of any form. However, the following rules are guide lines for how the transaction field name, `FIELDNAME`, in the `RESPONSE_TXN_LAYOUT` table should be named:

- Each row in the `RESPONSE_TXN_LAYOUT` and `REQUEST_TXN_LAYOUT` tables represents a single transaction field. Only fields have entries, field groupings do not have an entry, instead each field in the group has an entry. If a group is repeated then each group of transaction fields is repeated in the metadata table.
- The field name will be preceded by the `TXN_CODE` and `TXN_TYPE` and `OFFSET` if necessary to make the field unique to that transaction code and type.
- The field name will be the name of the field preceded by the name of each of the groups it is nested within.
- group names are delimited by the ``.`` Character, for example, `Header-info.restart-flag`.
- If a group has an occurs clause then the fields for that group must be repeated N times where N is the value immediately after the occurs clause.
- If a group has an occurs clause then each occurrence of the group will be named as follows: `groupname[n]` where `n` is the actual occurrence of the group.
- If a field has an occurs clause then each occurrence of that field must be repeated N times where N is the value immediately after the occurs clause.
- If a field has an occurs clause then each occurrence of the field will be named as follows: `fieldname[n]` where `n` is actual occurrence of the group.

The example below illustrates these rules:

```
01 Level1
    02 Field1 Pic X(10)
    02 Field2 Pic X(15) occurs 2
    02 Level2 occurs 2
```

```
03 FieldA Pic X(10)
```

```
03 FieldB Pic X(20) occurs 2
```

This copybook will be mapped as follows:

FIELDNAME	OFFSET	LENGTH	FIELD_PAD_CHAR	...
Level1.Field1	0	10	' '	
Level1.Field2[0]	10	15	' '	
Level1.Field2[1]	25	15	' '	
Level2[0].FieldA	40	10	' '	
Level2[0].FieldB[0]	50	20	' '	
Level2[0].FieldB[1]	70	20	' '	
Level2[1].FieldA	90	10	' '	
Level2[1].FieldB[0]	100	20	' '	
Level2[1].FieldB[1]	120	20	' '	

Mapping Entity Beans to Transactions

A single transaction may contain the information to populate several entity beans, or conversely a single entity bean may need to be populated from the results of several transactions.

One Transaction to One Entity

This is the simplest scenario. The data in the transaction is mapped to a single entity bean instance.

One Transaction to Many Entities

There are several different scenarios where one transaction may map to many entity instances:

Repeating Entities of the Same Type

A search transaction returns one or more results. Each result corresponds to a single entity bean instance. All entity instances are of the same type. For example a search for all accounts could return several results, each corresponding to a single account instance.

Single Entity of One Type Plus Repeating Entities of the Same Type

A search transaction returns several results. The first result corresponds to an entity of one type, while the subsequent results correspond to repeating instances of an entity of a different type. For

example an account statement transaction would return the statement details entity plus one or more account movement entities.

Master Entity with Dependent Entity

A search transaction returns data, which is modeled as two entities of different types. However there is a dependency between the two objects. For example a customer details transaction could contain the information for both a Customer entity and its dependent Address entity.

Entity Bean Persistence and the FPI

The job of a Persister is to manage writing and reading data in an Entity Bean instance to/from the data store. This means all the code for interacting with the data store is encapsulated in the Persister class. The Entity Bean instance talks to the Persister (through a well-defined interface) rather than directly to the data store.

This approach has the following advantages:

- The EJB developer does not have to worry about the complexities of talking to the Financial Process Integrator, for example, having to know transaction codes, making the EJB simpler to code.
- The EJB is protected from changes to the design of the Financial Process Integrator.

com.bankframe.ejb.bmp

This package contains the [EBMPEntity](#) and the [EPersister](#) class interfaces. It also contains the [EPersisterFactory](#) class that is used by an entity to get an instance of the persister.

com.bankframe.ejb.bmp.EBMPEntity

This interface defines the methods that all Siebel Retail Finance BMP Entity Beans must provide. To make it possible to define a single generic Financial Process Integrator persister that can be used by all BMP Entity Beans the [EBMPEntity](#) contains the [populate\(\)](#) and the [createPrimaryKey\(\)](#) methods, these methods are defined in the Entity Bean.

Table 18. EBMPEntity Methods

Method	Description
getPersister()	This method returns an instance of this Entity Bean's persister.
getPrimaryKey()	This method returns an instance of this Entity Bean's primary key.
getEntityName()	This method returns the JNDI name of the Entity Bean.
createPrimaryKey(DataPacket dp)	This method must be implemented by all sub-classes. It takes a DataPacket containing the information necessary to create a primary-key and returns an instance of the correctly initialised

Method	Description
	EPrimaryKey class.
populate(DataPacket dp)	This method must be implemented by all sub-classes. It takes a DataPacket containing the data for the Entity Bean's attributes. The populate() method must initialise the Entity Bean's attributes from this information.

com.bankframe.ejb.bmp.EPersister

This interface defines the methods that all Siebel Retail Finance EJB persisters must provide.

```
public interface EPersister {

    public Enumeration find(EBMPEntity entityBean, String methodName, DataPacket
finderData) throws ProcessingErrorException;

    public void load(EBMPEntity entityBean) throws ProcessingErrorException;

    public void store(EBMPEntity entityBean) throws ProcessingErrorException;

    public void amend(EBMPEntity entityBean, String methodName) throws
ProcessingErrorException;

    public EPrimaryKey create(EBMPEntity entityBean) throws ProcessingErrorException;

    public void remove(EBMPEntity entityBean) throws ProcessingErrorException;}

```

Table 19. EPersister Methods

Method	Description
find(EBMPEntity entityBean, String methodName, DataPacket finderData)	This method takes an instance of the calling entity; a methodName that specifies the name of the find operation to carry out and a finderData DataPacket that specifies the parameters of the find operation. This method will be called from Entity Bean ejbFindBy...() methods. It returns an Enumeration containing the matching primary keys for the specified search.
load(EBMPEntity entityBean)	This method takes an instance of the entity and loads its instance data from the data store. This method will be called from the Entity Bean's ejbLoad() method.
store(EBMPEntity entityBean)	This method takes an instance of the entity and writes it to the data store. This method will be called from the Entity Bean's ejbStore() method.
amend(EBMPEntity entityBean, String methodName)	This method takes an instance of the entity and a methodName that contains the name of the calling method and writes it to the data store. This method will be called from an Entity Bean's

Method	Description
	amend method when some or all of the entity is being updated.
<code>create(EBMPEntity entityBean)</code>	This method takes an instance of the entity and creates it in the data store and returns an instance of the entity's EPrimaryKey class.
<code>remove(EBMPEntity entityBean)</code>	This method takes an instance of the entity and removes it from the data store.

com.bankframe.ejb.bmp.EPersisterFactory

The [EPersisterFactory](#) class is responsible for creating and returning an instance of the Entity Bean's persister.

```
getPersister(String jndiName)
```

This method takes a String containing the JNDI name of the entity bean and returns an instance of the EJB's persister class.

The persister is returned by appending [persister.](#) to the JNDI name of the entity bean and checking the [BankframeResource.properties](#) file for the corresponding persister class. If there is no [persister.<EJB_JNDI_NAME>](#) key the default persister will be used instead.

Below is an example of the persister class settings in the [BankframeResource.properties](#):

```
persister.default=com.bankframe.ei.txnhandler.persister.TxnPersister
```

The default persister to be used for all BMP EJBs.

```
persister.eontec.bankframe.examples.bo.customer=com.bankframe.ei.txnhandler.persister.MasterEntityPersister
```

Specifies the persister to use for the specified EJB JNDI name.

Once the persister class has been identified the factory class checks to see if an instance of the class exists if one does it will return it, other wise it creates a new instance.

The persister is a stateless class that provides utility functions that need no more information than their parameters. No state information can be stored in the class. The factory creates the persister as a singleton.

Writing a Persister

The following are examples of how to implement methods declared in the [EPersister](#) interface using the [com.bankframe.ei.txnhandler.persister.TxnPersister](#) as an example. [TxnPersister](#) is the Financial Process Integrator implementation of the [EPersister](#).

find(EBMPEntity entityBean, String methodName, DataPacket finderData)

The [find\(\)](#) method is the entry-point to all search transactions that can be run against the host-system. This method maps the Entity Bean's name and the [methodName](#) to a transaction code and a

transaction type; it also retrieves the cache policy and decay time for the transaction. If the transaction can be cached it checks the cache for the data, to do this it calls the Cache's `checkPrimaryKeyInCache()` method which takes a `DataPacket` containing the primary key of the entity and a `long` containing the time-out value of the Transaction. If the transaction is not cached or the decay time has elapsed the transaction code and the transaction type are added to a `DataPacket` containing the parameters of the find operation and this `DataPacket` is sent to the Financial Process Integrator. The Financial Process Integrator will return a `Map` containing the search results. The persister stores the results in the cache by calling the Cache's `store()` method passing it the `Map` of results returned from the Financial Process Integrator.

```
public Enumeration find(EBMPEntity entityBean, String methodName, DataPacket
finderData) throws ProcessingErrorException {

    Enumeration result = null;

    //Using the entity name and the methodName get the txnCode,      //txnType,
    cachePolicy, and timeOutValue of the transaction from           //the
    PERSISTER_TXN_MAP database table.

    DataPacket txnMap = this.mapTxn(entityBean.getEntityName(), methodName);

    String cachePolicy = txnMap.getString(PersisterTxnMapConstants.CACHE_POLICY);

    long timeOutValue = new
    Long(txnMap.getString(PersisterTxnMapConstants.TIME_OUT_VALUE)).longValue();

    //check the cache policy to see if the data is cached
    if (!cachePolicy.equalsIgnoreCase(TxnPersisterConstants.NOT_CACHED)) {

        //check cache for the primary key

        if (!this.checkPrimaryKeyInCache(finderData, timeOutValue)) {

            //calling the processTxnRequest() method to send request to //the
            Financial Process Integrator and to receive and cache the //response.

            result = this.processTxnRequest(entityBean, this.getTxnData(finderData,
            txnMap), cachePolicy);

        }

        else {

            //the data is in the cache therefore return an enumeration of the
            //primary key

            Vector entityPk = new Vector();

            entityPk.addElement(entityBean.createPrimaryKey(finderData));

            result = new IteratorEnumeration(entityPk.iterator());

        }

    }

    else {

        //calling the processTxnRequest() method to send request to           //the
        Financial Process Integrator and to receive and cache the
        //response.

        result = this.processTxnRequest(entityBean, this.getTxnData(finderData,
        txnMap), cachePolicy);

    }

}
```



```

    return result;
}

```

processTxnRequest(EBMPEntity entityBean, DataPacket txnData, String cachePolicy)

This protected method is called by the `find()` method. It is responsible for passing the transaction details to the Financial Process Integrator, receiving the response, placing it in the cache and returning an enumeration of primary keys.

```

protected Enumeration processTxnRequest(EBMPEntity entityBean, DataPacket txnData,
String cachePolicy) throws ProcessingErrorException {
    try {
        Vector entityPk = new Vector();
        String txnCode = txnData.getString(TransactionHandlerConstants.TXN_CODE);
        if ((txnCode == null) ||
txnCode.equalsIgnoreCase(TransactionHandlerConstants.FIELD_NA)) {
            // do nothing
        }
        else {
            //Get an instance of the Financial Process Integrator and send the transaction
            //data to the processFindRequest() method.
            TransactionHandler transactionHandler = this.getTxnHandler();
            Map map = transactionHandler.processFindRequest(txnData);
            boolean persistant;

            //Before caching the data check to see if it is persistent or not. //Persistent
            data will be written to a database as well as to memory.
            if
(cachePolicy.equalsIgnoreCase(TxnPersisterConstants.CACHE_PERSISTENT)) {
                persistant = true;
            }
            else if
(cachePolicy.equalsIgnoreCase(TxnPersisterConstants.CACHE_NON_PERSISTENT) ||
cachePolicy.equalsIgnoreCase(TxnPersisterConstants.NOT_CACHED)) {
                persistant = false;
            }
            else {
                //throw an exception
            }

            //get the timeout value for the data and then store it in the cache.
            long timeOutValue = new
Long(txnData.getString(PersisterTxnMapConstants.TIME_OUT_VALUE)).longValue();
            this.storeInCache(map, timeOutValue, persistant);
        }
    }
}

```

```

//Process the keys of the map returned from the Financial Process Integrator to
//return an enumeration of primary keys.

    Set keys = map.keySet();

    Enumeration enum = Collections.enumeration(keys);

    while (enum.hasMoreElements()) {

        EPrimaryKey pk = entityBean.createPrimaryKey((DataPacket)
enum.nextElement());

        if (pk != null) {

            entityPk.addElement(pk);

        }

    }

    return new IteratorEnumeration(entityPk.iterator());

}

catch (CreateException ce) {

    throw new ProcessingErrorException(ce);

}

catch (RemoteException re) {

    throw new ProcessingErrorException(re);

}

}

```

mapTxn(String entityName, String methodName)

The persister class to get instances of the `PersisterTxnMap` Entity uses this protected method. Using the entity name and the `methodName` the `txnCode`, `txnType`, `cachePolicy`, and `timeOutValue` of the transaction from the `PERSISTER_TXN_MAP` database table.

```

protected DataPacket mapTxn(String entityName, String methodName) throws
ProcessingErrorException {

    try {

        PersisterTxnMapHome txnMaphome = (PersisterTxnMapHome)
ObjectLookup.lookup(PersisterTxnMapConstants.PERSISTERTXNMAP_JNDI_NAME,
PersisterTxnMapHome.class);

        PersisterTxnMapPK primaryKey = new PersisterTxnMapPK();

        primaryKey.entityName = entityName;

        primaryKey.methodName = methodName;

        PersisterTxnMap persisterTxnMap = (PersisterTxnMap)
txnMaphome.findByPrimaryKey(primaryKey);

        DataPacket result = persisterTxnMap.toDataPacket();

        return result;

    }

}

```

```

        catch (FinderException fe) {
            throw new ProcessingErrorException(fe);
        }
        catch (RemoteException re) {
            throw new ProcessingErrorException(re);
        }
    }
}

```

load(EBMPEntity entityBean)

The `load()` method is called by the entity bean's `ejbLoad()` method. All data returned by the Financial Process Integrator from the host is cached. The `load()` method uses the Entity Bean's primary key to retrieve the entity's data from the cache. It then calls the Entity Bean's `populate()` method to update the entity's attributes with the cached data.

```

public void load(EBMPEntity entityBean) throws ProcessingErrorException {
    EPrimaryKey pk = entityBean.getPrimaryKey();
    //retrieve the data from the cache.
    DataPacket cacheData = cache.retrieve(pk.toDataPacket());
    if (cacheData == null) {
        //throw an exception
    }
    //call the entity's populate method
    entityBean.populate(cacheData);
}

```

amend(EBMPEntity entityBean, String methodName)

The `amend()` method is called by an entity's `amend...()` method. It takes an instance of the entity and the `methodName`, calls the `toDataPacket()` on the entity bean and then calls the persister's `amend(EBMPEntity entityBean, String methodName, DataPacket amendData)`. The `amend()` is used for updating some or all of an entity's attributes.

```

public void amend(EBMPEntity entityBean, String methodName) throws
ProcessingErrorException {
    try {
        this.amend(entityBean, methodName, entityBean.toDataPacket());
    }
    catch (RemoteException re) {
        throw new ProcessingErrorException(re);
    }
}

```

amend(EBMPEntity entityBean, String methodName)

The `amend()` method is called by an entity's `amend...()` method, it takes an instance of the entity, the `methodName` and a `DataPacket` of data to use to update the entity and then calls the persister's protected `amend(EBMPEntity entityBean, String methodName, DataPacket data, Vector primaryKeys, boolean removeOperation)` method. The `amend()` is used for updating some or all of an entity's attributes.

```
public void amend(EBMPEntity entityBean, String methodName, DataPacket amendData)
    throws ProcessingErrorException {

    Vector pksOfEntitiesToAmend = new Vector();

    pksOfEntitiesToAmend.add(entityBean.getPrimaryKey().toDataPacket());

    this.amend(entityBean, methodName, amendData, pksOfEntitiesToAmend, false);
}
```

amend(EBMPEntity entityBean, String methodName, DataPacket data, Vector primaryKeys, boolean removeOperation)

The protected `amend()` method is called by the persister's `amend...()` method. The `amend()` checks if the `txnCode` is set to `CACHE_ONLY`, if it is then it will only update the cache, otherwise it adds the transaction code and the transaction type to a `DataPacket` containing the entity bean's update attributes and sends the `DataPacket` to the Financial Process Integrator. It also takes a `boolean` value which indicates if a remove operation is to be carried out on the host or from the cache. The `amend()` is used for updating some or all of an entity's attributes.

The key `persister.cache.updateOnAmend` in `BankframeResource.properties` determines if the cache is updated or removed after the amend operation is sent to the Financial Process Integrator.

```
protected void amend(EBMPEntity entityBean, String methodName, DataPacket data, Vector
    primaryKeys, boolean removeOperation) throws ProcessingErrorException {
    try {
        //Using the entity name and the methodName get the txnCode,
        //txnType, cachePolicy, and timeOutValue of the transaction from
        //the PERSISTER_TXN_MAP database table.
        DataPacket amendData = this.mapTxn(entityBean.getEntityName(), methodName);
        String txnCode = amendData.getString(TransactionHandlerConstants.TXN_CODE);
        String txnType = amendData.getString(TransactionHandlerConstants.TXN_TYPE);
        long timeOutValue = new
        Long(amendData.getString(PersisterTxnMapConstants.TIME_OUT_VALUE)).longValue();

        if (getIgnoreHost(txnCode) == false) {
            TransactionHandler transactionHandler = this.getTxnHandler();
            DataPacket update = new DataPacket(data.DATA_PACKET_NAME);
            update.append(update, data);
            //Add txnCode and txnType
            update.put(TransactionHandlerConstants.TXN_CODE, txnCode);
            update.put(TransactionHandlerConstants.TXN_TYPE, txnType);
            //send data to the Financial Process Integrator processRequest() method
            transactionHandler.processRequest(update);
        }
        if (removeOperation || getRemoveFromCache()) {
            this.removeFromCache(primaryKeys);
        }
        else {
            //put data into a map (same data used for each primary key):
            Map entityMap = new HashMap();
```

```

        for (int index = 0; index < primaryKeys.size(); index++) {
            entityMap.put(primaryKeys.elementAt(index), data);
        }
        String cachePolicy =
amendData.getString(PersisterTxnMapConstants.CACHE_POLICY);
        boolean bCachePolicy =
(cachePolicy.equalsIgnoreCase(TxnPersisterConstants.CACHE_PERSISTENT)) ? true : false;
        this.storeInCache(entityMap, timeOutValue, bCachePolicy);
    }
}
catch (RemoteException re) {
    throw new ProcessingErrorException(re);
}
catch (CreateException ce) {
    throw new ProcessingErrorException(ce);
}
}
}

```

store(EBMPEntity entityBean)

The `store()` method notifies the Financial Process Integrator of a change to an Entity Bean instance. This method maps the Entity Bean's name to a transaction code and a transaction type. It adds the transaction code and the transaction type to a `DataPacket` containing the entity bean's update attributes and sends the `DataPacket` to the Financial Process Integrator. The `store()` is used for updating all of an entity's attributes. The `store()` method is called from the Entity Bean's `ejbStore`. This `store()` method is provided to allow for it to be overwritten for a specific implementation but typically it calls the `amend(EBMPEntity entityBean, String methodName)` method with a `methodName` variable with a value of `store`.

```

public void store(EBMPEntity entityBean) throws ProcessingErrorException {
    this.amend(entityBean, TxnPersisterConstants.STORE_NAME);
}

```

storeInCache(Map data, long timeOutValue, boolean persistent)

The protected `storeInCache()` method used by the persister to determine which cache to store the host data in either the default cache or the time out cache.

```

protected void storeInCache(Map data, long timeOutValue, boolean persistent)
throws ProcessingErrorException {
    if (this.timeoutCache != null) {
        this.timeoutCache.store(data, timeOutValue, persistent);
    }
    else {
        cache.store(data, persistent);
    }
}

```

create(EBMPEntity entityBean)

The `create()` method notifies the Financial Process Integrator that a new record needs to be created on the Host System. The `create()` method is called from the entity bean's `ejbPostCreate()` method to create a new record on the Host System. This `create()` method is provided to allow for it to be overwritten for a specific implementation but typically it calls the `amend(EBMPEntity entityBean, String methodName)` method with a `methodName` variable with a value of `create`. Returns the primary key if the create was successful.

```
public EPrimaryKey create(EBMPEntity entityBean)
    throws ProcessingErrorException {
    try {
        this.amend(entityBean, TxnPersisterConstants.CREATE_NAME);
        EPrimaryKey pk =
entityBean.createPrimaryKey(entityBean.toDataPacket());

        return pk;
    } catch (RemoteException re) {
        throw new ProcessingErrorException(re);
    }
}
```

remove(EBMPEntity entityBean)

The `remove()` method notifies the Financial Process Integrator that a record on the Host System should be deleted. The `remove()` method notifies the Financial Process Integrator that a record on the Host System should be deleted. This `remove()` method is provided to allow for it to be overwritten for a specific implementation but typically it calls the `amend(EBMPEntity entityBean, String methodName)` method with a `methodName` variable with a value of `remove`.

```
public void remove(EBMPEntity entityBean) throws ProcessingErrorException {
    this.amend(entityBean, TxnPersisterConstants.REMOVE_NAME, true);
}
```

removeFromCache(EBMPEntity entityBean)

The `removeFromCache()` method is used to delete an Entity's cached data from the cache.

```
public void removeFromCache(EBMPEntity entityBean) throws
ProcessingErrorException {
    try {
        DataPacket pk =
entityBean.createPrimaryKey(entityBean.toDataPacket()).toDataPacket();
        Vector pks = new Vector();
        pks.addElement(pk);
        this.removeFromCache(primaryKeys);
    }
}
```

```

        catch (RemoteException re) {
            throw new ProcessingErrorException(re);
        }
    }
}

```

removeFromCache(Vector primaryKeys)

This protected method is used by the persister to delete an Entity's cached data from the cache.

```

protected void removeFromCache(Vector primaryKeys) throws
ProcessingErrorException {
    cache.remove(primaryKeys, true);
}

```

PersisterTxnMap

PERSISTER_TXN_MAP Table

The Persister transfers information to and from the Financial Process Integrator. In order to do this the persister must be able to match the entity and method called to the `txnCode` and `txnType` and does so using the `PERSISTER_TXN_MAP` table. The Persister retrieves the `txnCode` and `txnType` by using the method name and the entity's `JNDI` name. The `PERSISTER_TXN_MAP` table also contains details of the caching policy and decay time for the specified Transaction. The persister checks the cache to see if the information it needs is stored there. If the Transaction is cached a time out value is specified so that the persister can check if the data in the cache needs to be refreshed or is still valid.

Table 20. PERSISTER_TXN_MAP

ENTITY_ NAME	METHOD_ NAME	TXN_ CODE	TXN_ TYPE	CACHE_ POLICY	INDEX_ NAME	TIME_OUT VAULE
eontec. bankframe.Account	getAccountDetails()	MQ_ACC01	MQIMS	none		5

ENTITY_NAME

The `ENTITY_NAME` attribute in the `PERSISTER_TXN_MAP` table maps to the `entityName` attribute in the Persister class. The `entityName` is the `JNDI` name of the bean, for example, `eontec.bankframe.Account`.

METHOD_NAME

The `METHOD_NAME` attribute in the `PERSISTER_TXN_MAP` table maps to the `methodName` attribute in the Persister class. The `methodName` is the name of the method which is being called, for example, `getAccountDetails()`.

TXN_CODE

This attribute contains the code number for the host transaction.

TXN_TYPE

This attribute identifies the middleware associated with a transaction such as MQSeries, IMS, TUXEDO or CICS.

CACHE_POLICY

The [CACHE_POLICY](#) field states whether the data from the Financial Process Integrator is cached or not. The [CACHE_POLICY](#) should be configured as follows:

CACHE_POLICY Setting	Description
none	The transaction results cannot be cached.
persistent	The cache is to be written to a database so it is available even if there is a system failure.
memory	The transaction results are to be cached in memory.

Note that unless an [INDEX_NAME](#) is provided, the cache will be queried by the primary key.

INDEX_NAME

The name of the cache index to use to look up request data in the cache. This column is only applicable if [CACHE_POLICY](#) is set to [memory](#). The [INDEX_NAME](#) value corresponds to the name of a cache index defined in the [BankframeResource.properties](#) file under the [cache.index](#) key. If there is no entry in the [BankframeResource.properties](#) file, the [CacheIndexFactory](#) returns an instance of [CacheIndex](#) by default. The [CacheIndex](#) class uses the [IndexMetaData](#) bean to determine the index structure and the name of the index to cache.

TIME_OUT_VALUE

The [TIME_OUT_VALUE](#) attribute in the [PERSISTENT_TXN_MAP](#) specifies the length of time in milliseconds that the stored data remains valid. When data is retrieved from the cache its creation time is compared to the current time and if the difference is greater than the [TIME_OUT_VALUE](#) the data is requested from the host.

Configuring the PERSISTENT_TXN_MAP Table

Refer to the CustomerSearch and AccountSearch examples section for information on how to configure the [PERSISTENT_TXN_MAP](#) table.

com.bankframe.ei.txnhandler.persistertxnmap

PersisterTxnMapBean

[PersisterTxnMapBean](#) is a container-managed entity bean that houses information about the relation of an entity bean's methods to host transactions. It maps to the [PERSISTER_TXN_MAP](#) table in the database. The [PersisterTxnMapBean](#) solution set layer is located in the [com.bankframe.ei.txnhandler.persistertxnmap](#) package and its implementation is in the [com.bankframe.ei.impl.txnhandler.persistertxnmap](#) package.

Configuring FPI Persister Settings

The FPI Persister is configured in [BankFrameResource.properties](#) in the settings listed in Table 42.

Table 21. Configuring FPI Persister Settings

Key Name	Example Value	Description
persister.cache.updateOnAmend	yes	Determines if the cache is updated or removed after an amend operation. Possible values are yes or no .
persister.default	com.bankframe.ei.txnhandler.persistertxnmap.TxnPersister	The default persister to be used for all BMP EJBs.
persister.<EJB_JNDI_NAME>	com.bankframe.ei.txnhandler.persistertxnmap.MasterEntityPersister	Specifies the persister to use for the specified EJB JNDI name.

Financial Process Integrator Caching

The host cache package is superseded by the caching framework package. Each cache class in the [com.bankframe.ei.txnhandler.hostcache](#) package can be described by a [Cache/CachePolicy](#) combination from the [com.bankframe.services.cache](#) package.

Host Cache Examples

Generally it is recommended to create a new [com.bankframe.services.cache.Cache](#) instance with a given cache policy whenever caching is needed. However should you need to create a cache based on the deprecated host cache settings in [BankFrameResource.properties](#) then the following method should be used:

```
com.bankframe.services.cache.CacheFactory.getHostCache(String cacheName)
```

This method will return an instance of `com.bankframe.services.cache.Cache`. This cache can be manipulated by methods described in the Caching Framework document. This cache will also have a Caching Policy associated with it that describes how the cache deals with removal of expired entries.

Deprecated FPI Caching Settings

The hostcache settings listed in Table 22 have been deprecated and these should be replaced with caching framework settings. See Caching Framework for information on configuring all caching settings.

The deprecated settings listed in Table 22 are retained for backwards compatibility with `com.bankframe.services.cache.CacheFactory.getHostCache`.

Table 22. Deprecated Host Cache Settings

Key Name	Example Value	Description
<code>transactionHandler.hostcache.maxMemCacheSize</code>	500	The maximum memory cache size. This should be less than the <code>maxDbCacheSize</code> value.
<code>transactionHandler.hostcache.threshold</code>	20	Used to determine how many entries to move at once. This value should be greater than zero and less than the <code>maxMemCacheSize</code> value.
<code>transactionHandler.hostcache.cacheType</code>	SINGLEJVM, CLUSTERABLE or MEMORY	The cache implementation to use.

Financial Process Integrator Engine

The Financial Process Integrator engine is the core of the Financial Process Integrator; it must perform the following tasks:

- Transform `DataPacket` requests into data messages of the correct format for the host system.
- Route data messages to the appropriate host system using a Siebel Retail Finance Connector.
- Transform incoming data responses from Siebel Retail Finance Connector into `DataPacket` results.

The Financial Process Integrator has two usage scenarios:

- It is invoked by a persister class, this is usually done in response to a call from an Entity Bean finder method, that is, a search operation.
- It is invoked from a session bean, this is usually done for amend operations.

The Financial Process Integrator provides an interface to support both these usage scenarios.

For each new host system that MCA Services is to transact with, the following customizations have to be made in the Financial Process Integrator Engine:

- The `DESTINATION` and `TXN_ROUTE` database tables have to be edited to specify a Siebel Retail Finance Connector appropriate for the type of host system.
- The metadata has to be designed and edited. The metadata defines the form of the host system requests and responses. The Financial Process Integrator engine uses the metadata definitions to process the transaction requests to and from the host system. The metadata is explained further in the metadata chapter.
- The `BasicDataFormat` class may have to be customized. The Financial Process Integrator engine uses the `BasicDataFormat` class for host system specific formatting and processing of transaction requests and responses.
- The necessary entries in `BankframeResource.properties` have to be edited.

These steps are described in the following topics.

Financial Process Integrator Engine Interface

The Financial Process Integrator engine is implemented as a stateless EJB session bean called `TransactionHandler`. The `TransactionHandler` solution set layer is located in the `com.bankframe.ei.txnhandler.transactionhandler` package and its implementation is in the `com.bankframe.ei.impl.txnhandler.transactionhandler` package. Its remote interface provides the following methods:

<code>java.util.Map processFindRequest (DataPacket txnData)</code>	process a <code>findBy</code> request transaction. This is a search.
<code>Vector processRequest (DataPacket txnData)</code>	process a create, amend or remove operation.

processFindRequest (DataPacket dataPacket)

This method is called whenever a `findBy` request transaction needs to be sent to the host system. The `DataPacket` parameter `txnData` specifies values that will be placed in the transaction request that is sent to the host system. The method `processFindRequest()` returns a `Map` that contains all the entities that make up the host system response. The key to a `Map` element is a `DataPacket` of the primary key for that entity in the `Map`. This method throws a `java.rmi.RemoteException` or a `com.bankframe.ejb.ProcessingErrorException` if an error occurs.

processRequest (DataPacket dataPacket)

This method is called by a session bean to update data on the host system. It takes a [DataPacket](#) indicating what fields to amend. The session bean creates a [DataPacket](#) of all the values in the host system that have to be updated and passes the [DataPacket](#) to this method on the Financial Process Integrator.

The [processRequest\(\)](#) returns a [Vector](#) containing all the entities that make up the host system response.

This method throws a [java.rmi.RemoteException](#) or a [com.bankframe.ejb.ProcessingErrorException](#) if an error occurs.

Transaction Request DataPacket

The transaction request [DataPacket](#) is the [DataPacket](#) passed to the Financial Process Integrator by a client, that is, the persister, to request that a transaction be processed. The table below shows the elements of a sample transaction request [DataPacket](#).

TXN_CODE	TEST_ACC
TXN_TYPE	TXNMQ
ACCOUNT_NAME	John Williams

The transaction code, [TXN_CODE](#), specifies the transaction ID as defined on the host system.

The transaction type, [TXN_TYPE](#), specifies which host system the transaction is sent to.

[TXN_CODE](#) and [TXN_TYPE](#) are used to determine:

- Which Siebel Retail Finance Connector will be used to communicate with the host system.
- Which transaction fields the specific transaction request to the host system must contain.
- Which transaction fields the specific transaction response from the host system contains.

In the sample [DataPacket](#) shown above [ACCOUNT_NAME](#) is the data value that is required for the host system to process the transaction request. The name of the customer in this case is '[John Williams](#)'. This name will be used in all the transaction fields passed to the host system that require an [ACCOUNT_NAME](#) value.

Transaction Request Processing Steps

A transaction request data object has to be created from the transaction request [DataPacket](#), to pass to the host system. The form of this transaction request depends on the host system and the Siebel Retail Finance Connector being used to connect to the host system. The transaction request has to be built by the Financial Process Integrator to work with the appropriate Siebel Retail Finance Connector and host system, this requires a conversion from the string based transaction request [DataPacket](#) to a host system specific format.

The steps the Financial Process Integrator performs to handle a transaction request are:

- 1 Build all the necessary fields for the transaction request by querying the entity bean `RequestTransactionField` with the `TXN_CODE`, `TXN_TYPE`, that is, obtain all the transaction fields that are necessary for this type of transaction request to be processed on the host system. This entity bean is covered further in the metadata chapter.
- 2 The `BasicDataFormat` class fills the appropriate transaction fields with data from the transaction request `DataPacket`, that is, using the transaction request `DataPacket` shown in the previous section the transaction field values that require a value for the `ACCOUNT_NAME` are filled with the value `John Williams`.
- 3 The `BasicDataFormat` class forms a host system formatted data object request consisting of the selected transaction fields.
- 4 The host system data object is passed to the appropriate Siebel Retail Finance Connector. The appropriate Siebel Retail Finance Connector is determined by querying the `TransactionRoute` and `Destination` entity beans.
- 5 The Connector's responsibility is to pass the request on to the host system. This is covered further in the Connectors chapter.
- 6 The data object response is returned by the host system via the Siebel Retail Finance Connector.
- 7 The necessary transaction fields for the host system response are determined by querying the entity beans `ResponseTransactionField` and `TransactionMetaData` with the `TXN_CODE`, `TXN_TYPE`. These entity beans are covered further in the metadata chapter.
- 8 The `BasicDataFormat` class extracts the appropriate fields from the host system response using the transaction fields determined in 7.
- 9 The `BasicDataFormat` class determines if the host system response is an error result by querying the entity bean `TransactionErrorCondition` with the `TXN_CODE`, `TXN_TYPE` and the host system response data. This is described in more detail in the metadata chapter.
- 10 The `BasicDataFormat` class creates a `Map` or `Vector` (depending if the operation is a find or an amend) of response `DataPackets` from the extracted data.
- 11 The `BasicDataFormat` determines if another request has to be sent to the host system due to the host sending the response data in sub-parts, the entire process is repeated if necessary.

The Response `DataPackets` are returned to the calling client in the form of a `Map` or `Vector`.

Transaction Data-Format Class

The Financial Process Integrator uses a data-format class for:

- Processing of request `DataPackets` into host system specific data.
- Processing of host system response data into `DataPackets`.
- Creating/removing and processing of the transaction headers.
- Pre-processing the response before the transaction fields are processed.
- Formatting the transaction fields for making a request to the Siebel Retail Finance Connector.
- Formatting the transaction fields in the response from the Siebel Retail Finance Connector.
- Determine if repeated requests are required to be sent to the host system.

The Financial Process Integrator engine determines the correct data-format class to use at run-time by querying the [TransactionRoute](#) entity bean.

For each form of host system the [BasicDataFormat](#) class may have to be customized. The MCA class [com.bankframe.ei.txnhandler.dataformat.basic.BasicDataFormat](#) is a generic base data-format class implementation. This can be sub-classed to reuse the main functionality.

DataFormat Class Interface

All data-format classes must implement the [DataFormat](#) interface [com.bankframe.ei.txnhandler.dataformat.DataFormat](#). This interface has the following definition:

```
import com.bankframe.ejb.ProcessingErrorException;

public interface DataFormat {

    public void toDataPacketsMap(Object txnData, Map responseEntitiesMap,
DataPacket txnDataPacket, String txnCode, String txnType) throws
ProcessingErrorException;

    public void toDataPacketsVector(Object txnData, Vector
responseEntitiesVector, DataPacket txnDataPacket, String txnCode, String txnType)
throws ProcessingErrorException;

    public Object buildRequestTxn(DataPacket txnDataPacket, String txnCode,
String txnType) throws ProcessingErrorException;

    public boolean moreToRequest();

    public void notifyProcessingFinished();

    public void setConnectionSpecification(Object command, String
connectorProperties) throws ProcessingErrorException;
}
```

Any modifications necessary for transaction processing can be made in the data-format class without modifying the Financial Process Integrator source code.

The methods [buildRequestTxn\(\)](#), [toDataPacketsMap\(\)](#) and [toDataPacketsVector\(\)](#) use:

- the utility class `com.bankframe.ei.txnhandler.dataformat.DataFormatUtils` to perform common routines such as converting ASCII text to EBCDIC format.
- the following class to get all metadata required to process the transaction:

```
com.bankframe.ei.txnhandler.dataformat.TransactionHandlerUtils
```

Instantiating the Data-Format Class

The Financial Process Integrator instantiates the specified data-format class as shown in the following pseudo-code:

```
//The Transaction Route Entity Bean used to get the DataFormat class name:

TransactionRoute txnRoute;

//Obtain DataFormat class name from the Transaction Route Entity Bean

String dataFormatClass = txnRoute.getDataFormatName();

//load and instantiate class using reflection

Class classFactory = Class.forName(dataFormatClass);

DataFormat dataFormat = (DataFormat) classFactory.newInstance();

//call the required method, for example,

boolean moreToRequest = dataFormat.moreToRequest();
```

Data-Format Class Request Processing Steps

The Financial Process Integrator creates the host system request using the Data-Format method `buildRequestTxn(txnDataPacket, txnCode, txnType)`. The implementation of the `BasicDataFormat` processing depends on the host system format and can be customized depending on the host system requirements.

`buildRequestTxn(txnDataPacket, txnCode, txnType)` makes the following processing steps:

- A byte stream is created to contain the transaction request that will be passed to the host system via the host Connector.
- The request transaction fields necessary for the specified transaction code and type are obtained by calling the `TransactionHandlerUtils` method `generateTxnRequestFields(txnCode, txnType)`.
- For each request transaction field a value for the field is obtained from the request `DataPacket`, `txnDataPacket`. If the field value is not a `MANDATORY` field in the request `DataPacket` then the default value specified in `REQUEST_TXN_LAYOUT` is used.

- Each request transaction field value is formatted according to the settings specified in `REQUEST_TXN_LAYOUT` and added to the byte stream. This is performed by the method `fillTxnField(TransactionField txnField, String dataValue)`.
- The byte stream is returned to the Financial Process Integrator.

Data-Format Class Response Processing Steps

The Financial Process Integrator calls the method `toDataPacketsMap()` to process the host system response for a find operation. The Financial Process Integrator calls the method `toDataPacketsVector()` to process the host system response for an amend operation.

The implementation of the `BasicDataFormat` processing depends on the host system format and can be customized depending on the host system requirements.

The two methods make the following processing steps:

- 1 `processTxnResponse ()` is called for the host system response data.
- 2 `processTxnResponse ()` calls the method `checkForErrorCondition()` to test the host system response to determine if it is an error result from the host system.
- 3 `checkForErrorCondition()` calls `checkForErrorValue()` to determine if a transaction field value matches an error condition.
- 4 If an error occurred then `processTxnResponse ()` calls the method `handleHostSystemError()`. The method `handleHostSystemError()` is customised if error handling is required for a host system. It takes appropriate action such as further processing of metadata and throwing of a `ProcessingErrorException`. See `BasicDataFormat` class for an example.
- 5 If no error occurred then `processTxnResponse()` calls the method `processTransactionRecord()`.
- 6 The method `processTransactionRecord()` gets the necessary metadata specified by the `TXN_CODE` and `TXN_TYPE` in the request `DataPacket` using the class `com.bankframe.ei.txnhandler.dataformat.TransactionHandlerUtils`.
- 7 The method `processTransactionRecord()` calls the method `preProcessTxnData()` to pre-process the response data, that is, removes the header information if necessary.
- 8 The method `processTransactionRecord()` processes the host system data extracting data necessary for each entity specified by the metadata. Entity `DataPacket` results processed from the host system data are added to the `Vector` of entity bean results, `responseEntities`. If a `Map` of entities is being created (due to `toDataPacketMap()` starting the process) then for each element in the `Vector responseEntities` a `Vector` of all the associated primary key `DataPackets` is added for later processing. The `Vector` of associated primary keys is updated as primary key values are extracted from the host system response data.
- 9 Processing returns at this point to `toDataPacketsMap()` and `toDataPacketsVector()`.
- 10 `checkIfNoEntitiesFound()` is called to check if any entity `DataPackets` were processed from the host system data. If none were processed then the `BasicDataFormat` class returns from processing.
- 11 `checkIfMoreToRequest()` is called to update the flag indicating if this transaction requires further calls to the host system.
- 12 The method `postProcessResponseData()` is called to perform any necessary post processing of the `Vector` of entities, `responseEntities`, which were created from the host system data.

- 13** At this point the method `toDataPacketsMap()` converts the `Vector` of entities into a `Map` of entities. The key to each entity in the `Map` is the primary key `DataPacket` created previously during step 8.
- 14** The method `toDataPacketsMap()` returns the `Map` of entities to the Financial Process Integrator engine. The method `toDataPacketsVector()` returns the `Vector` of entities to the Financial Process Integrator engine.

The Financial Process Integrator engine will call `moreToRequest(...)` to check if the request has to be generated again, more data retrieved from the host system and the above steps repeated to process the response.

toDataPacketsVector()

The method `toDataPacketsVector(Object txnData, Vector responseEntitiesVector, DataPacket txnDataPacket, String txnCode, String txnType)` converts the host system response data object elements into `DataPackets` to respond to the client. The method returns the results in a `Vector` of `DataPackets` called `responseEntitiesVector` that will be sent to the client.

The resulting `DataPacket` contents depend on the metadata definition.

The names of the `DataPackets` in the `Vector` are specified by the `DP_NAME` field in the metadata table `RESPONSE_META_DATA`. The names of the elements in the `DataPacket` are the `DP_FIELD` values specified in the metadata table `RESPONSE_META_DATA`, this is described in detail in the metadata chapter.

This is called by the Financial Process Integrator method `processRequest()` for amending data on the host system. The session bean that called the Financial Process Integrator in this case expects a `Vector` of results `DataPackets`.

toDataPacketsMap()

The method `toDataPacketsMap(Object txnData, Map responseEntitiesMap, DataPacket txnDataPacket, String txnCode, String txnType)` converts the response from the host system into `DataPackets` to respond to the client. The method returns the results in the `Map responseEntitiesMap` in the form of `DataPackets` which will be sent to the client. This is called by the Financial Process Integrator method `processFindRequest()` for getting data on the host system. The entity bean that called the Financial Process Integrator in this case expects a `Map` of results `DataPackets`.

The `Map` elements are the entity elements determined from the host system response.

The key to an element in the map is a `DataPacket` object containing the primary key elements of the entity in question.

The name of the `DataPacket` takes the form: `<ENTITYNAME>`

For example an entity called `TEST` could have a primary key `DataPacket` with the following values:

`DATA_PACKET_NAME = TEST`

`SORT_CODE = 99-99-99`

`ACCOUNT_NUMBER = 11223344`

This is the key to the entity element in the `Map`. Associated with the key is an element containing the `DataPacket` of values for the entity in question.

The name of the `DataPacket` is specified by the `DP_NAME` field in the metadata table `RESPONSE_META_DATA`. The names of the elements in the `DataPacket` are the `DP_FIELD` values specified in the metadata table `RESPONSE_META_DATA`, these are the names understood by the persister object that calls the Financial Process Integrator. The metadata tables are described in detail in the metadata chapter.

For example the element associated with the key shown previously could be a `DataPacket` with the following values:

```
DATAPACKET NAME = TEST

ACCOUNT_NAME = John Williams

SORT_CODE = 99-99-99

ACCOUNT_NUMBER = 11223344
```

moreToRequest ()

When the Financial Process Integrator uses the `BasicDataFormat` class to process the response data from the host system the `BasicDataFormat` class determines if there is more data still to process from the host system. This may be the case where the header in the response data specifies that the response from the host system has been broken into several parts. This method allows the Financial Process Integrator to detect if the host system is finished sending response data or if there is more data to be received and processed.

The `BasicDataFormat` class generally determines if there are more requests to send to the host system as follows:

- 1 The definition of the metadata for the host system defines two header fields: a flag indicating that the host system has to be called again and a counter for the current count of calls made to the host system for the request.
- 2 After processing the host system response the `BasicDataFormat` checks the above flags, this is performed in the `BasicDataFormat` method `checkIfMoreToRequest(DataPacket txnRequest, Vector responseEntities)`.
- 3 `checkIfMoreToRequest(DataPacket txnRequest, Vector responseEntities)` modifies the request settings in `txnRequest` if necessary for the next call to the host system, that is, the current count of calls is incremented and updated in the request settings.
- 4 If the method `checkIfMoreToRequest(DataPacket txnRequest, Vector responseEntities)` determines from the flags in the header fields that there are more requests to be made then a `boolean` flag is set to `true`. The request `DataPacket` is updated if necessary with new settings if further requests will be needed to the host system. The method `moreToRequest()` returns the value of this `boolean` flag when called by the Financial Process Integrator engine.
- 5 The Financial Process Integrator calls the method `moreToRequest()`. If the result is `true` then the Financial Process Integrator generates another transaction request and posts the request to the host system requesting further data. The updated request settings are used by the `BasicDataFormat` class to process the transaction request.

- 6 The Financial Process Integrator Engine repeats this process until `moreToRequest()` returns `false`. The default value returned by `moreToRequest()` is `false`.

See the example data-format class:

```
com.bankframe.examples.txnhandler.dataformat.testcustomer. TestCustomerDataFormat
```

notifyProcessingFinished()

The method `notifyProcessingFinished()` is called by the Financial Process Integrator engine when all processing of a transaction is complete. This allows the data-format class to clean up any temporary data and variables.

setConnectionSpecification(Object command, String connectorProperties)

The method `setConnectionSpecification(Object command, String connectorProperties)` is called by the Financial Process Integrator engine to set the Connector Specification of an EAB Command Bean. These are the Connector properties obtained from the Destination EJB.

TransactionHandlerUtils helper class

The methods `buildRequestTxn()`, `toDataPacketsMap()` and `toDataPacketsVector()` use the helper class `com.bankframe.ei.txnhandler.TransactionHandlerUtils` to obtain the necessary metadata for processing of transactions.

Table 23. TransactionHandlerUtils Helper Methods

Method	Description
<code>boolean isMetaDataCached()</code>	Determines from <code>BankframeResource.properties</code> if caching has been enabled for the metadata.
<code>boolean isRoutesCached()</code>	Determines from <code>BankframeResource.properties</code> if caching has been enabled for the routes.
<code>TransactionField getTxnFieldFromList(Iterator txnFields)</code>	Returns the next <code>TransactionField</code> interface from the <code>List</code> .
<code>MetaData getMetaDataFromIterator(Iterator txnMetaData)</code>	Returns the interface of the next <code>MetaData</code> interface from a <code>List</code> .
<code>TransactionField getTxnResponseFieldFromName(ResponseTransactionFieldHome txnFieldHome, String txnFieldName, boolean metaDataCached)</code>	Finds the <code>TransactionField</code> interface to a transaction field entity from the transaction field name.
<code>List generateTxnRequestFields(String txnCode, String</code>	Generates the Transaction

Method	Description
<code>txnType)</code>	Request fields List for specified transaction code and type.
<code>List generateTxnResponseMetaData(String txnCode, String txnType)</code>	Generates the Transaction Response Meta-data List of entity mappings for specified transaction code and type.
<code>Map generateTxnResponseFields(List txnMeta dataList)</code>	Generates a Map of the Response Transaction Fields from the field names that are specified in the Meta-data List .
<code>Map generateTxnResponseErrorConditions(String txnCode, String txnType)</code>	Generates the Map of Transaction Response Error-Conditions for the specified transaction code and type.
<code>getErrorConditionFromEnum(Enumeration txnErrorConditions)</code>	Returns a TransactionErrorCondition interface from the Enumeration.
<code>RequestTransactionFieldHome getRequestTransactionFieldHome()</code>	Returns a RequestTransactionFieldHome object.
<code>ResponseTransactionFieldHome getResponseTransactionFieldHome()</code>	Returns a ResponseTransactionFieldHome interface.
<code>MetaDataHome getMetaDataHome()</code>	Returns a MetaDataHome interface representation.
<code>TransactionErrorConditionHome getTxnErrorConditionHome()</code>	Returns a TransactionErrorConditionHome interface.

DataFormatUtils Helper Class

The methods `buildRequestTxn()`, `toDataPacketsMap()` and `toDataPacketsVector()` use the helper class `com.bankframe.ei.txnhandler.dataformat.DataFormatUtils` to perform common routines such as converting ASCII text to EBCDIC format.

Table 24. DataFormatUtils Helper Methods

Method	Description
<code>byte[] subset(byte data[], int startIndex, int endIndex)</code>	extracts the specified amount from the data byte-array and

Method	Description
	returns the result
<code>byte[] toEbcdic(String input)</code>	converts ASCII to EBCDIC
<code>String ebcdicToString(byte ebcdic[])</code>	converts EBCDIC to ASCII String
<code>byte[] toComp(String input, Boolean signed, int inputSize)</code>	converts the numerical string to a Cobol number
<code>byte[] toComp3(String input, boolean signed, int maxWholeDigits, int maxFractionalDigits)</code>	converts numerical <code>String</code> to a Cobol number COMP-3 format
<code>String compToString(byte input[])</code>	converts a Cobol number into a numerical <code>String</code>
<code>String comp3ToString(byte input[], int numWholeDigits, int numFractionalDigits)</code>	converts a Cobol number, Comp 3, into a numerical <code>String</code>
<code>byte[] toStandard(String input, boolean signed, int maxWholeDigits, int maxFractionalDigits)</code>	converts a numerical <code>String</code> to a Cobol Standard format
<code>String standardToString(byte input[], int numWholeDigits, int numFractionalDigits)</code>	converts a Cobol Standard to a numerical <code>String</code>
<code>ToHex(byte input, StringBuffer buf)</code>	converts an input <code>byte</code> into a <code>StringBuffer</code> hexadecimal representation
<code>ToHex(byte input[], StringBuffer buf)</code>	converts an input <code>byte[]</code> into a <code>StringBuffer</code> hexadecimal representation
<code>ToHex(int input, StringBuffer buf)</code>	converts an input <code>int</code> into a <code>StringBuffer</code> hexadecimal representation
<code>String toHexString(byte input)</code>	converts an input <code>byte</code> into a <code>String</code> hexadecimal representation
<code>String toHexString(byte input[])</code>	converts an input <code>byte[]</code> into a <code>String hexadecimal</code> representation
<code>String toHexString(int input)</code>	converts an input <code>int</code> into a <code>String</code> hexadecimal representation

Transaction Route Entity Bean

To determine which Siebel Retail Finance Connector the Financial Process Integrator will use to communicate with the host system the `TransactionRoute` and `Destination` entity beans are queried.

The `TransactionRoute` solution set layer is located in the `com.bankframe.ei.txnhandler.transactionroute` package and its implementation is in the `com.bankframe.ei.txnhandler.impl.transactionroute` package.

The `TransactionRoute` entity bean maps to Table 25. `TXN_ROUTE` Database, which has the following form:

Table 25. `TXN_ROUTE` Database

TXN_CODE	TXN_TYPE	DESTINATION_ID	DATAFORMAT
TEST_ACC	TXN_DUMMY	C002	<code>com.ims.DataFormat</code>
TEST_ACC	TXNMQ	C001	<code>com.mqs.DataFormat</code>

The `TransactionRoute` entity bean is queried with the `TXN_CODE` and `TXN_TYPE` specified in the transaction request `DataPacket` to determine:

- The Siebel Retail Finance Connector used to communicate with the host system; the `DESTINATION_ID` is a key into the `DESTINATION` database table.
- The data-format class used to convert the request transaction into a host-specific format and to convert the response into a Siebel-specific format.

Caching of Transaction Routes

The Financial Process Integrator can cache the queried transaction routes to improve performance.

The `transactionHandler.routes.cache` entry in the `BankframeResource.properties` file specifies whether caching of Transaction Routes is enabled for the Financial Process Integrator.

The caching is performed by the class `com.bankframe.ei.txnhandler.transationroute.TransactionRouteCache`. This class uses the MCA generic caching framework.

Destination Entity Bean

To determine which Siebel Retail Finance Connector to instantiate and which Connector properties to use the `Destination` entity bean is queried. The `Destination` solution set layer is located in the `com.bankframe.ei.txnhandler.destination` package and its implementation is in the `com.bankframe.ei.txnhandler.impl.destination` package.

The `Destination` entity bean maps to Table 26. `DESTINATION` database, which has the following form:

Table 26. `DESTINATION` database

DESTINATION_ID	CONNECTOR_FACTORY_CLASSNAME	CONNECTOR_PROPERTIES
C001	<code>com.bankframe.examples.</code>	<code>offlineMode=disable;</code>

DESTINATION_ID	CONNECTOR_FACTORY_CLASSNAME	CONNECTOR_PROPERTIES
	<code>txnhandler.connector. testcustomer.TestCusto merConnectionFactory</code>	<code>Port=9999; channel=SENDER.CHANNEL; hostname=99.999.999.99; queueManager=QM_test; requestQueue=QUEUE.REQ; responseQueue=QUEUE.REPLY; wait.interval=200; characterSet=37</code>
<code>C002</code>	<code>com.bankframe.examples. txnhandler.connector. coboltest. CobolTestConnectionFactory</code>	<code>offlineMode=fetch;</code>

The `DESTINATION` table has three fields:

- The `DESTINATION_ID` is a key index into the table from the `TXN_ROUTE` table.
- The `CONNECTOR_FACTORY_CLASSNAME` is the Factory class name of the Siebel Retail Finance Connector Factory, which is instantiated to obtain a Connector.
- The `CONNECTOR_PROPERTIES` is a semi-colon delimited string containing connector properties, which the Siebel Retail Finance Connector Factory uses during initialization.

The Siebel Retail Finance Connector properties determine if an off-line Connector will be used for testing the system.

The off-line Connector setting can be either:

- `"disable"`, not to be used.
- `"fetch"` mode.
- `"store"` mode.

The Siebel Retail Finance Connector properties has the following key to specify the off-line mode:

```
offlineMode=<mode>;
```

The Siebel Retail Finance Connector properties string is passed to the `open()` method of the instantiated Siebel Retail Finance Connector Factory.

Caching of Destinations

The Financial Process Integrator can cache the queried destinations to improve performance.

The `transactionHandler.routes.cache` entry in the `BankframeResource.properties` file specifies whether caching of destinations is enabled for the Financial Process Integrator.

The caching is performed in the class

`com.bankframe.ei.txnhandler.destination.DestinationCache`. This class uses the MCA generic caching framework.

Posting the Transaction Request data Object to the Host Connector

Once the transaction request `DataPacket` has been converted into the appropriate data format for the host system the data object is passed to the specified Siebel Retail Finance Connector. All Connectors implement the interface:

```
com.bankframe.ei.txnhandler.connector.EConnection
```

The Financial Process Integrator interacts with all Connectors through the methods of this interface. The steps to post the transaction request `java.lang.Object` to the Siebel Retail Finance Connector are:

The Siebel Retail Finance Connector Factory class specified by the `DESTINATION` table is instantiated.

- 1 An interface to the required Connector is obtained from the Connector Factory using the method `getConnection(String connectorProperties)`. The parameter `connectorProperties` is the Connector Properties String obtained from the `DESTINATION` entity bean.
- 2 The `EConnection` method `public Object post(Object txns)` is called. The parameter `Object txns` is the host system specific transaction request data object.
- 3 The method `post(Object txns)` returns a data `Object` containing the results from the host system.

Configuring Financial Process Integrator Properties

The Financial Process Integrator is configured in the `BankframeResource.properties` using the settings listed in the table below.

Table 27. Financial Process Integrator Properties

Method	Description
<code>transactionHandler.dataSource.jndiName=jdbc/bankfrm</code>	The data source that the Financial Process Integrator uses for database access, for example, <code>jdbc/bankfrm</code> .
<code>transactionHandler.metaData.cache</code>	Specifies if the meta data caching is enabled, <code>true</code> or <code>false</code> .
<code>transactionHandler.routes.cache</code>	Specifies if caching for the transaction routes and destinations is enabled, <code>true</code> or <code>false</code> .

Method	Description
<code>transactionHandler.routes.cache.maxSize</code>	Max size of the routes cache.
<code>transactionHandler.requesttxnlayout.cache.maxSize</code>	Max size of the request transaction layout cache.
<code>transactionHandler.responsetxnlayout.cache.maxSize</code>	Max size of the response transaction layout cache.
<code>transactionHandler.errorConditions.cache.maxSize</code>	Max size of the response error conditions cache.
<code>transactionHandler.metadata.cache.maxSize</code>	Max size of the response metadata cache.

Financial Process Integrator Testing using Test Servlet

MCA Services supplies several servlets for testing the core functionality of the Financial Process Integrator Engine. The servlets are described in the following sub-topics.

TransactionHandlerHomePage

The main Financial Process Integrator servlet is

`com.bankframe.ei.txnhandler.TransactionHandlerHomePage`

This servlet provides links to all the Financial Process Integrator test servlets and is accessible from the main MCA `ServiceServlet`.

TransactionHandlerTestServlet

The main servlet for testing the functionality of the Financial Process Integrator is

`com.bankframe.ei.txnhandler.TransactionHandlerTestServlet`

`TransactionHandlerTestServlet` tests the entire transaction processing cycle of the Financial Process Integrator engine. It generates the specified transaction, determines the route and destination, sends the generated request to the specified Connector, processes the response from the host system and displays the results of the request. The caching configuration specified in the `BankframeResource.properties` file is used for the processing cycle.

To use the servlet to test the Financial Process Integrator the user first creates the necessary request `DataPacket` that will be sent to the Financial Process Integrator. The two operations provided for this are:

- "Add a new field", adds a field to the request `DataPacket`. The user specifies the `DataPacket` field name and its value and clicks on the button "Add".
- "Remove a field", removes a field from the request `DataPacket`. The user specifies the `DataPacket` field name to remove and clicks on the button "Remove".

After the necessary request `DataPacket` fields have been created and given the correct values for the transaction request the "Update" button is clicked to update the text box displaying the "Current `DataPacket`".

The user can choose the following requests to send to the Financial Process Integrator:

- find operation, this calls the Financial Process Integrator method `processFindRequest()` with the specified `DataPacket` to simulate a findBy operation being performed.
- amend operation, this calls the Financial Process Integrator method `processRequest()` with the specified `DataPacket` to simulate an amend operation being performed.

For example the `AccountSearch` findBy example requires the following settings:

- `TXN_CODE=ACCOUNTFIND`
- `TXN_TYPE=TEST`

The `CustomerSearch` findBy example requires the following settings:

- `TXN_CODE=TESTFIND0001`
- `TXN_TYPE=TEST`
- `OWNER_ID=1234560010`

The `CustomerSearch` findBy example operation requires that the `OWNER_ID` field is added to the request `DataPacket`. The Financial Process Integrator throws an exception if this is missing because it is specified in the metadata for the example as a mandatory field.

The `CustomerSearch` amend example requires the following settings:

- `TXN_CODE=TESTAMND0001`
- `TXN_TYPE=TEST`
- `OWNER_ID=1234560010`
- `FIRST_NAME=JOHN`

This amend operation will amend the first name of the user with the `OWNER_ID 1234560010` to `JOHN` and remove all the other settings for this user.

The results of the transaction request are displayed on a result page. The results consist of a table of all the entity `DataPacket` results. The time to process the transaction request is determined by the servlet and shown on the result page.

TransactionRouteTestServlet

The servlet for testing the transaction route functionality of the Financial Process Integrator is

```
com.bankframe.ei.txnhandler.transactionroute.TransactionRouteTestServlet
```

`TransactionRouteTestServlet` tests that the transaction route details for a given `TXN_CODE` and `TXN_TYPE` can be determined from the MCA database. These details are used for determining which data-format class to instantiate and which `DESTINATION_ID` to use. This test however does not instantiate the data-format class or use the `DESTINATION_ID`, it just displays details for the transaction route.

The caching configuration specified in the `BankframeResource.properties` file is used.

The `TXN_CODE` and `TXN_TYPE` are modified for the transaction route that has to be tested and the "Update" button clicked. The transaction route details are requested by clicking on the "Request" button.

If the details are obtained successfully than they are displayed.

The `AccountSearch` example uses `TXN_CODE=ACCOUNTFIND` and `TXN_TYPE=TEST`.

DestinationTestServlet

The servlet for testing the destination functionality of the Financial Process Integrator is

```
com.bankframe.ei.txnhandler.destination.DestinationTestServlet.
```

`DestinationTestServlet` tests that the destination details for a given `DESTINATION_ID` can be determined from the MCA database. These details are used for creating and initializing the Connector for communicating with the host system. This test however does not communicate with the host system, it just displays details for the host Connector.

The caching configuration specified in the `BankframeResource.properties` file is used.

The `DESTINATION_ID` is modified for the destination that has to be tested and the "Update" button clicked. The destination details are requested by clicking on the "Request" button.

If the details are obtained successfully than they are displayed.

The `AccountSearch` example uses the `DESTINATION_ID=C002`.

RequestTransactionFieldServlet

The servlet for testing the transaction request fields functionality of the Financial Process Integrator is

```
com.bankframe.ei.txnhandler.transactionlayout.impl.request.  
RequestTransactionFieldServlet
```

`RequestTransactionFieldServlet` tests that the transaction request field details for a given `TXN_CODE` and `TXN_TYPE` can be determined from the MCA database. These details are used for creating the transaction request to send to the host system. This test however does not generate the host system specific request, it just displays details for the transaction request fields.

The caching configuration specified in the `BankframeResource.properties` file is used.

The `TXN_CODE` and `TXN_TYPE` are modified for the transaction request fields that have to be tested and the "Update" button clicked. The transaction request field details are requested by clicking on the "Request" button.

If the details are obtained successfully than they are displayed as bullet points for each transaction request field.

The `AccountSearch` example uses `TXN_CODE=ACCOUNTFIND` and `TXN_TYPE=TEST`.

ResponseTransactionFieldServlet

The servlet for testing the transaction response fields functionality of the Financial Process Integrator is

```
com.bankframe.ei.txnhandler.transactionlayout.impl.response.ResponseTransactionFieldServlet.
```

`ResponseTransactionFieldServlet` tests that the transaction response field details for a given transaction `FIELDNAME` can be determined from the MCA database. These details are used for processing the transaction response data from the host system. This test however does not process a host system response, it just displays details for the specified transaction response field.

The caching configuration specified in the `BankframeResource.properties` file is used.

The `FIELDNAME` is modified for the transaction response field to be tested and the "Update" button clicked. The transaction response field details are requested by clicking on the "Request" button.

If the details are obtained successfully than they are displayed.

The `AccountSearch` example uses a transaction field with `FIELDNAME=CARD-NUMBER`.

MetadataServlet

The servlet for testing the transaction response metadata functionality of the Financial Process Integrator is

```
com.bankframe.ei.txnhandler.transactionresponse.metadata.MetadataServlet.
```

`MetadataServlet` tests that the transaction response metadata details for a given `TXN_CODE` and `TXN_TYPE` can be determined from the MCA database. These details are used for mapping transaction fields in the host system response to result entity `DataPacket` results. This test however does not process the mappings, it just displays details for the transaction response metadata.

The caching configuration specified in the `BankframeResource.properties` file is used.

The `TXN_CODE` and `TXN_TYPE` are modified for the transaction response metadata that have to be tested and the "Update" button clicked. The transaction response metadata details are requested by clicking on the "Request" button.

If the details are obtained successfully than each entity mapping is displayed as a bullet point.

The `AccountSearch` example uses `TXN_CODE=ACCOUNTFIND` and `TXN_TYPE=TEST`.

TransactionErrorConditionServlet

The servlet for testing the transaction response error-condition functionality of the Financial Process Integrator is

```
com.bankframe.ei.txnhandler.transactionresponse.errorcondition.TransactionErrorConditionServlet.
```

`TransactionErrorConditionServlet` tests that the transaction response error-condition details for a given `TXN_CODE` and `TXN_TYPE` can be determined from the MCA database. These details are used to determine if a host system response is an error. This test however does not process any host system response, it just displays details for the transaction response error-conditions.

The caching configuration specified in the `BankframeResource.properties` file is used.

The `TXN_CODE` and `TXN_TYPE` are modified for the transaction response error-conditions that have to be tested and the "Update" button clicked. The transaction response error-condition details are requested by clicking on the "Request" button.

If the details are obtained successfully than each response error-condition is displayed as a bullet point, otherwise there are no error-conditions for the specified transaction code and type.

The `AccountSearch` example uses `TXN_CODE=ACCOUNTFIND` and `TXN_TYPE=TEST`.

EIS Connectors

This topic covers MCA Services support for Enterprise Information Systems (EIS) Connectors and includes the following sub-topics:

- MCA Services Connector Architecture
- JCA Support

MCA Services Connector Architecture

The MCA Services Connector architecture defines a standard architecture for connecting Siebel Retail Finance applications to heterogeneous host or middleware systems. Examples of systems that a host connector might communicate with include MQSeries, IMS or CICS. The Connector architecture allows you utilize pre-built connectors provided with MCA, or build customized Connectors for any number of enterprise host systems.

An MCA Connector is a package of Java classes, which are used to connect an enterprise Java application to a Host or middleware system. The connector architecture enables a developer to provide a standard connector for a given host system. The connector plugs into an application server and provides connectivity between the Siebel Retail Finance application, the application server, and the host system.

The Siebel Retail Finance Host Connector Architecture is similar in structure to the Java Database Connectivity (JDBC) interfaces. A Host Connector provides similar functionality to a JDBC driver, except that it connects to a host system instead of a relational database. In fact, it is possible to write a host connector for a DBMS quite easily.

Host Connectors can also optionally provide functionality for connection pooling and connection management. The Connector architecture defines a standard interface for integrating with connection management implementations, whether they are provided by the connector provider or an application server.

The Connector architecture also defines the manner in which all clients connect to host system resources. Once a connector has been successfully deployed on an application server, Siebel Retail Finance applications call the `post(Object)` method of the desired connector to forward the request onto the host system. When used within the Financial Process Integrator environment, the connectors are called automatically from the Financial Process Integrator engine.

Siebel Retail Finance Connector Interfaces and Components

A Siebel Retail Finance Connector is made up of several Java components that make it easy to support connection pooling and management. The following interfaces make up the generic Siebel Retail Finance Connector architecture, and are implemented by all MCA Host Connectors. They are found in the package `com.bankframe.ei.txnhandler.connector`. They are as follows:

EConnection Interface

An `EConnection` represents an application-level handle that is used by a client to access the underlying physical connection. The actual physical connection associated with an `EConnection` instance is represented by an `EManagedConnection` instance. A client gets an `EConnection` instance by using the `getConnection()` method on an `EConnectionFactory` instance.

All Siebel Retail Finance Host Connectors must implement the `post(Object)` and `close()` methods of the `EConnection` interface. The `post()` method of all connectors should forward a client's transaction request to the middleware or host system that the Connector interfaces with, and should return an object representing the response from the system. The `close()` method must close the physical connection between the connector and its host system, or if it is running in a pooled environment it must release the connection back to the connection pool, for re-use by another client.

EConnectionEvent Class

The `EConnectionEvent` class provides information about the source of a connection related event. An `EConnectionEvent` instance contains the following information:

- The type of the connection event, that is, `CONNECTION_CLOSED` or `CONNECTION_ERROR_OCCURRED`.
- The `EManagedConnection` instance that generated the connection event. An `EManagedConnection` instance is returned from the method `EConnectionEvent.getSource()`.
- The `EConnection` handle associated with the `EManagedConnection` instance. This is required for the `CONNECTION_CLOSED` event and optional for the other event types.
- Optionally, an exception indicating the connection related error. Note that the exception is used for `CONNECTION_ERROR_OCCURRED`.

The `EConnectionEvent` class defines a `CONNECTION_CLOSED` and a `CONNECTION_ERROR_OCCURRED` type of event notifications.

EConnectionEventListener Interface

The `EConnectionEventListener` interface provides an event callback mechanism to enable a Connection Manager to receive notifications from an `EManagedConnection` instance. A Connection Manager uses these event notifications to manage its connection pool, and to clean up any invalid or terminated connections. Typically, the Connection Manager will implement a `ConnectionEventListener` interface (or one of its helper classes will). The Connection Manager registers a connection listener with an `EManagedConnection` instance by using `EManagedConnection.addConnectionEventListener(EventListener)` method.

The Connection Manager (or helper class that implements the `EConnectionEventListener` interface) must ensure that it handles the events to close a connection and to handle errors. It does this by implementing the `connectionClosed(EConnectionEvent)` and `connectionErrorOccurred(EConnectionEvent)` interfaces of the `EConnectionEventListener`.

EConnectionFactory Interface

The `EConnectionFactory` provides an interface for getting a connection to a

Host system. Each individual SRF connector will provide an implementation of the `EConnectionFactory` interface. A client application that wishes to use a Siebel Retail Finance Host Connector must first instantiate the Connection Factory class.

A client application obtains an `EConnection` from an `EConnectionFactory` implementation in the following manner:

```
String connectorFactoryClassName="com.test.MyConnectionFactory";

Class classFactory = Class.forName(connectorFactoryClass);

EConnectionFactory cxf = (EConnectionFactory) classFactory.newInstance();

EConnection connection = cxf.getConnection(connectorProperties);
```

EConnectionManager Interface

The `EConnectionManager` interface provides a hook for a Siebel Retail Finance Connector to pass a connection request to the application server or Connection Manager. The application server or the Connector provider typically provides an implementation of the `EConnectionManager` interface. The `EConnectionManager` implementation handles or delegates connection pooling and management. The connector architecture does not specify how a Connection Manager implements these services; the implementation can be specific to an application server, or to a specific connector.

After a Connection Manager hooks-in its services, the connection request gets delegated to an `EManagedConnectionFactory` instance either for the creation of a new physical connection or for the matching of an already existing physical connection.

An implementation class for `EConnectionManager` interface is required to implement the `java.io.Serializable` interface. In the non-managed application scenario, the `EConnectionManager` implementation class can be provided either by a connector (as a default `EConnectionManager` implementation) or by application developers.

EManagedConnection Interface

The `EManagedConnection` class represents a physical connection to the underlying Host system. Managed connections are often re-cycled and used in connection pools to improve performance.

EManagedConnectionFactory Interface

The `EManagedConnectionFactory` instance is a factory of both `EManagedConnection` and connector-specific connection factory instances. This interface supports connection pooling by providing methods for the matching and creation of `EManagedConnection` instances. Implementations of this interface must provide a `createManagedConnection(String)` method and a `matchManagedConnections(Set, String)` method.

Using a Siebel Retail Finance Connector with the Financial Process Integrator

The Financial Process Integrator engine is set-up to automatically format data for a host or middleware system, and pass these requests to the SRF connector that corresponds to that system.

There is a Database table (that is created when you install MCA Services) named **DESTINATION**. This table is the key mediator between the Financial Process Integrator engine and Connectors.

Table 28. Destination Table Schema

Method	Description
DESTINATION_ID	This column corresponds to the foreign key DESTINATION_ID in the TXN_ROUTE database table. It is used to correlate a particular host transaction request to its corresponding Host Connector information in the DESTINATION table.
CONNECTOR_FACTORY_CLASSNAME	This column specifies the Connection Factory class to instantiate. From this Factory class an EConnection is obtained to the Host Connector. The Host Connector is used to send a transaction to its destination host system. This name must correspond to the value of the <code>transactionHandler.connector.~~.ConnectionFactory_Impl</code> key specified in the <code>BankframeResource.properties</code> file for the Connector:
CONNECTOR_PROPERTIES	<p>This column is a list of properties that are specific to a connection created by an MCA connector. The properties must be in the format: <code><name>=<value>;<name2>=<value2></code>. Note that multiple properties are separated by a semi-colon delimiter, for example, <code>offlineMode=fetch;user=bankfrm;password=<></code></p> <p>Note that all Connectors that support OffLine processing must contain a property called offlineMode in the CONNECTOR_PROPERTIES field.</p>

Therefore, to configure which connector you want to use through the Financial Process Integrator engine, you will have to manipulate the **DESTINATION** database table. For each transaction code you have, you must insert the correct Connector Factory class name of the connector that you wish to use (in the **CONNECTOR_FACTORY_CLASSNAME** column), and insert the desired properties of that connector (in the **CONNECTOR_PROPERTIES** column), where individual properties are separated by semi-colons.

For more details on how to configure the Financial Process Integrator engine for processing and formatting requests, refer to the chapter on the Financial Process Integrator engine.

OffLine Connector

One of the pre-built connectors that are provided with MCA is the OffLine Connector. This connector is designed for testing and development purposes, to simulate posting transactions to a live host system. The OffLine Connector sits between a standard SRF connector and a middleware or host system. The OffLine Connector simulates transactions to a live host system by capturing request and response data that passes through the original SRF connector and storing it in a relational database

table. Then, the original SRF connector has the option of setting its `offlineMode` property to either `fetch`, `store` or `disable`.

OffLine Disable Mode

If a SRF connector is running in OffLine `disable` mode (that is, it is not in `fetch` or `store`), then the original SRF connector sends all requests directly to the host system, and returns responses directly to the Financial Process Integrator engine. There is no interaction with the OffLine Connector. This mode should be the default mode for all connectors.

OffLine Store Mode

A SRF connector can run in OffLine store mode by setting its `offlineMode` property to `store`. When a connector is in store mode, it continues to send transaction requests to its live middleware or host system. However, after the response has been obtained from the host or middleware system, the original connector makes a call to the OffLine connector to store both the transaction request and the transaction response in the OffLine database. This ensures that the connector can process this same request at a future time when running in offline `fetch` mode.

OffLine Store Processing Sequence

- Financial Process Integrator forwards client request to an MCA Connector.
- MCA Connector `posts` request to the Host system and waits for the response.
- MCA Connector sends original request and host response to the OffLine Connector before sending host response back to the Financial Process Integrator.
- Siebel Retail Finance OffLine Connector stores the request and response in a Database.

OffLine Fetch Mode

A SRF connector can run in OffLine fetch mode by setting its `offlineMode` property to `fetch`. When a connector is in fetch mode, it re-directs all transaction requests to the OffLine Connector, instead of making a connection to the live host or middleware system. The OffLine Connector will then look-up the response to the transaction request in the OffLine Database and return the expected response back to the original connector, which in turn returns to the Financial Process Integrator engine. Note that a request sent to the OffLine Connector will only be retrieved properly if that same transaction request had previously been made while the connector was in offline `store` mode.

OffLine Connector Implementation

The Siebel Retail Finance OffLine Connector is a standard implementation of the Siebel Retail Finance Connector interfaces. It also provides an implementation of a connection manager and a connection pool, which utilize JDBC `DataSource` objects to obtain sql connections to the OffLine database table.

The OffLine Connector contains the following Java classes, found in the `com.bankframe.ei.txnhandler.connector.offline` package.

OfflineConnection. This class represents an application-level handle to the OffLine Database that is used by a client to access the underlying physical connection. Siebel Retail Finance Host Connectors will call the `post(Object)` method of this connection to either fetch requests from the offline database when they do not want to run against the live host system, or they will call the `post(Object,`

`Object`) method to store requests and responses in the offline database for later offline transactions. All objects sent through the `post()` method must be serializable, so that they can be stored offline. If they are not serializable then the `post()` method will return null, and requests will not be stored or fetched from the OffLine database. The OffLine connector writes and retrieves the objects passed into the `post()` methods as serializable `byte` streams to the OffLine Database. The `OfflineConnection` also provides a `close()` method that must be called when you are finished with the connection, so that it can be released back to the pool, or destroyed.

OfflineConnectionFactory. This class provides a means for an MCA Connector to obtain a connection to the OffLine Connector database. The `OfflineConnectionFactory` is instantiated by a Connector to enable access to the Offline Connector. The application then uses the `getConnection(String)` method to obtain an instance of the corresponding `EConnection` class.

The only parameter that needs to be passed in to the `getConnection(String)` method of the OffLine Connector is the `offlineMode` value. This value can be set to `disable`, `fetch`, or `store`. The `getConnection(String)` method for setting the OffLine Connector to store mode would be:

```
EConnection con = cf.getConnection("offlineMode=store");
```

When an MCA Connector calls the `post(Object)` method of the OffLine Connector, it will receive back the exact same type of object that it would expect to receive from the host or middleware system that it communicates with.

OfflineConnectionManager. This class acts as a resource manager for the OffLine Connector. It provides connection pooling and management for an application that is using multiple OffLine Connectors. The connection manager is initialized and associated with the connector at deploy time, and its execution is invisible to the developer during connector interaction. There are two settings in the `BankframeResource.properties` file for configuring the OffLine connection manager. The `maxConnections` setting lets you specify a maximum number of settings that you want the OffLine Connector to be allowed. Setting this to 0 will allow unlimited number of connections to be created by the connector (although, this is in turn limited by a `DataSource` and the connection pool settings that you have in your application server).

```
transactionHandler.connector.OfflineConnector.maxConnections=3
```

The `timeOut` setting lets you specify the amount of time to wait for a connection that is in use. If all of the connections in a pool are currently in use, the connector will wait for a period of `timeOut` seconds for a connection. If it does not obtain a connection when this time has expired, it will stop waiting and return `null`.

```
transactionHandler.connector.OfflineConnector.timeOut=10
```

OfflineConnectionPool. This class is a Connection Pool for the OffLine Connector. It stores and manages a series of physical (`EManaged`) connections to the offline database. This class is used in conjunction with the `OfflineConnectionManager`, for situations where a JDBC `DataSource` object is available from the application server. The `OfflineConnectionPool` is used by the `OfflineConnectionManager`, and its interaction with the connector is invisible to the user.

OfflineManagedConnection. This class is an implementation of the `EManagedConnection` class for the OffLine Connector. It represents the physical connection to the offline database. All interaction with the OffLine Connector should be through the `OfflineConnection`, and you should never need to use the `OfflineManagedConnection` directly.

OfflineManagedConnectionFactory. The `OfflineManagedConnectionFactory` class is a factory for `OfflineManagedConnection` instances. This class supports connection pooling by providing methods

for the matching and creation of [OfflineManagedConnection](#) instances. All interaction with the Offline Connector should be through the [OfflineConnection](#), and you should never need to use the [OfflineManagedConnection](#) directly.

HTTPConnector

One of the pre-built connectors provided with MCA is the HTTPConnector. This connector is designed for connecting to systems over the HTTP protocol and can be used in a message based SOAP environment. It has one connection property: [URL_STRING](#). Use the Financial Process Integration tool to config the [URL_STRING](#) connection property for the HTTPConnector.

XMLDataFormat

HTTPConnector uses XMLDataFormat to encode and decode the request for transport over HTTP. XMLDataFormat uses DPTPDmCodec to convert a Vector of DataPackets to and from an XML string. When using DPTPDmCodec, XML validation should be disabled. To do this, set `xml.parser.validating=false` in the properties file `BankframeResource.properties`. The XMLDataFormat can transform the DPTPDmCodex XML string by applying an XML stylesheet. Different XSLT strings can be defined for requests and responses using the `XSL_STYLESHEET` column in `REQUEST_TXN_LAYOUT`, `RESPONSE_META_DATA` and `RESPONSE_TXN_LAYOUT` tables. Note that for a request or response, because the XSLT will define the record structure, and the DPTPDmCodec will be used to convert to and from a Vector of DataPackets, there is only one record required in `REQUEST_TXN_LAYOUT`, `RESPONSE_META_DATA` and `RESPONSE_TXN_LAYOUT` tables for each host request and response.

For example, `RESPONSE_TXN_LAYOUT` normally defines the response field positions and the `RESPONSE_META_DATA` is used to define the mapping of fields to DataPacket keys. Since the XSLT will define the response structure, and the DPTPDmCodec will be used to produce the Vector of DataPackets, there is only one record required for `RESPONSE_TXN_LAYOUT` with the `TXN_CODE` and `XSL_STYLESHEET` columns set. Other columns, while can have default values. Similarly, the `RESPONSE_META_DATA` also only requires one record with the `TXN_CODE` and `XSL_STYLESHEET` columns set.

JCA Support

This topic outlines how the MCA Financial Process Integrator facilitates support for JCA connectors. JCA is an open-ended specification for connecting to EIS systems from within an application server environment. JCA resource adapters are packaged within .rar files and deployed on an application server in the same way as EJBs or Web applications. Generally a middleware vendor will supply this resource adapter for interaction with their software. These resource adapters are likely to support connection management, transaction management and security management. To interact with an EIS via a resource adapter a client API is needed. This can be a standard API such as the client Connection Interface (CCI) from JCA, or a proprietary API supplied by the middleware vendor. It is at the discretion of the middleware vendor as to which API they support.

A simple resource adapter is supplied with MCA to demonstrate the potential use of JCA within the Financial Process Integrator. This is deployed in the application server. For this example the resource adapter will interact with a file containing customer data. This is the same file used by the customer search example. Only a brief examination of the resource adapter follows because in any real world

scenario using JCA the resource adapter will be available from the middleware vendor, and its actual working should be hidden from a client developer.

Defining the Resource Adapter

Below is a resource adapter deployment descriptor that is bundled within the .rar file. The important elements in this XML are the following tags:

<managedconnectionfactory-class> This class will be the class that the application server interacts with to match requests to connections or to create new connections when required.

<connectionfactory-interface> This is the interface that the above class implements.

<connectionfactory-impl-class> This is the factory class that allows an application component to get a connection to the EIS. This class will be used by the managedconnectionfactory-class defined above to get the actual connection, thus handing over responsibility to the application server for processes such as connection pooling. An object of this type will be returned from the application when a component does a JNDI lookup on the connector component.

<connection-interface> This is the interface that the connection class implements. It must contain a getConnection() method.

<connection-impl-class> This is the class that provides connectivity to the EIS. This is got from the connectionfactory implementation class.

The complete descriptor follows:

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE connector PUBLIC "-//Sun Microsystems, Inc.//DTD Connector 1.0//EN"
'http://java.sun.com/j2ee/dtds/connector_1_0.dtd'>

<connector>

    <display-name>Some JCA</display-name>

    <vendor-name>Some Vendor</vendor-name>

    <spec-version>1.0</spec-version>

    <eis-type>EIS definition</eis-type>

    <version>1.0</version>

    <resourceadapter>

        <managedconnectionfactory-
class>com.bankframe.jca.samplefileadapter.SampleManagedConnectionFactory</managed
connectionfactory-class>

        <connectionfactory-
interface>javax.resource.cci.ConnectionFactory</connectionfactory-interface>
```

```

        <connectionfactory-impl-
class>com.bankframe.jca.samplefileadapter.SampleConnectionFactory</connectionfact
ory-impl-class>

        <connection-interface>javax.resource.cci.Connection</connection-
interface>

        <connection-impl-
class>com.bankframe.jca.samplefileadapter.SampleConnection</connection-impl-
class>

        <transaction-support>NoTransaction</transaction-support>

        <authentication-mechanism>

            <authentication-mechanism-type>BasicPassword</authentication-
mechanism-type>

            <credential-
interface>javax.resource.security.PasswordCredential</credential-interface>

            </authentication-mechanism>

            <reauthentication-support>false</reauthentication-support>

        </resourceadapter>

    </connector>

```

Interacting with the Resource Adapter

Since the adapter is a standard J2EE component it can be found via a JNDI lookup. When a JNDI lookup is performed on the resource adapter a reference to the [ConnectionFactory](#) class is returned. Using this class the application component, for example, the Financial Process Integrator bean, can call the [getConnection\(\)](#) method on the [ConnectionFactory](#) object. This will return an object with which the Financial Process Integrator can send requests to the EIS and receive responses. It should be noted that JCA supports asynchronous communication only.

For the sake of simplicity the sample resource adapter can be sent requests and receive responses in the form of [DataPackets](#). This is within the scope of the JCA specification as it doesn't restrict the resource adapter vendor to follow any specific interface. Rather it specifies that any interfaces that are used must contain at least a specific method, such as [getConnection\(\)](#), in the case of the [ConnectionFactory](#) class.

In order for the FPI to support a specific JCA adapter a data formatter class will have to be developed to format the data between [DataPackets](#) and the correctly formatted request object needed to interact with the EIS through the resource adapter. The sample JCA adapter processes only [DataPackets](#). This negates the need for a data formatter, as the [DataPacket](#) request can be passed to the resource adapter, which will return a [DataPacket](#) response.

To demonstrate the use of JCA from an application component there is a JSP that will perform a JNDI lookup for the resource adapter and then send a request in `DataPacket` format, wait for a response and then display the response. This example demonstrates the core functionality of JCA.

Store and Forward

The Financial Process Integrator's Store and Forward framework provides the means to store transactions, for example, in the event of a host going offline, in order to forward them to the host at a later time. The `HostStatusMonitor` periodically reads the `STORE_TRANSACTION` table. If records are found it attempts to forward the first record to the HOST. If a successful response is retrieved it means the HOST has come back online and a `FORWARD_ALL` request is sent to the `ForwardTransactionServlet`. The `com.bankframe.ei.txnhandler.storeandforward.impl.forwardtransaction.ForwardingThread` is then started. This thread forwards the rest of the stored transactions.

The store and forward implementation supports one host (destination). If support for multiple hosts is required the store and forward functionality needs to be extended and customized.

The store and forward system operates between the Siebel Retail Finance mid-tier (that is, the Siebel Financial Components) and the host. The Store and Forward system will only enable the storing of data for update to the host, it will not store data retrieved from the host.

Refer also to the sample file `storeandforward.sql` supplied in the MCA Services `Install` folder.

Determining if the Host is Offline

When a transaction fails to go to the host, the host is marked as offline and the transaction is stored for forwarding. The flow of execution is as follows:

- Each time a transaction is passed to the Financial Process Integrator it will attempt to send it to the host.
- The Financial Process Integrator will check with the host that it received the transaction.
- If the host did not receive the transaction or the host cannot be contacted then the transaction is stored for later forwarding and the host status will be set to offline.
- When a host is marked as offline it will remain marked as such for a specified configurable period. During that specified period no further attempts will be made to send transactions to that host; all transactions will instead be stored (except for transactions that are not permitted to be stored, these instead will result in an exception being thrown). This time period is configurable.
- When the time period has expired the forwarding mechanism will try to send the first entry on the queue to the host.
- If the first entry is forwarded successfully then the host is determined to be back online. The host status will be set to read-only and the forwarding thread will commence forwarding all stored transactions in batches. This batch figure will be configurable.
- If the first entry is not forwarded successfully then the forwarder will wait for the time period mentioned above, and then attempt to forward the first entry again. It will repeat this process until the host comes back online.
- When the store has been emptied of stored transactions the host will be marked online.

- When the host is forwarding the transactions those which are completed successfully will be added to the [SUCCESSFUL_TRANSACTION](#) table while those transactions which return an error from the host will be added to the [ERROR_TRANSACTION](#) table.

Host Status

The host has three states; these are as follows:

- **ON_LINE.** When the host status is set to [ON_LINE](#) all transactions are processed normally.
- **OFF_LINE.** When the host status is set to [OFF_LINE](#) any read transactions will throw an exception while write transactions will be stored to be forwarded later.
- **FORCE_OFF_LINE.** When the host is set to [FORCE_OFF_LINE](#) any read transactions will throw an exception while write transactions will be stored to be forwarded later. This ensures that when the host is set offline no attempts will be made to check if the host is back online until it has been set to online.

Host Operation types

The Financial Process Integrator Meta data must identify which transactions are read transactions and which transactions are write transactions.

Read transactions

- Read transactions cannot be carried out when the host is offline.
- Read transactions should not be stored if the host is offline, an exception should be thrown if an attempt is made to carry out a read transaction when the host is offline.
- Read transactions should become available as soon as the host comes back online.

Write transactions

- Write transactions cannot be carried out when the host is offline, but it is permissible to store some kinds of write transactions when the host is offline, and forward them when the host is back online.

Destination Entity Bean

To determine which Siebel Retail Finance Connector to instantiate and which Connector properties to use the [Destination](#) entity bean is queried. The [DESTINATION](#) table has been extended to include a new field; [HOST_STATUS](#), which is used by the Financial Components and the Persister to check if the host is online.

DestinationEjbMap Entity Bean

The Financial Components will need to know if the host is online or offline so they can apply the appropriate business logic. In order to do this it must be able to match the EJB and method called to the host destination, to do this it uses the [DESTINATION_EJB_MAP](#) table. Using the method name and

the `JNDI` name the `isHostOnline()` method in the `StoreAndForwardUtils` class retrieves the host destination. The `DESTINATION_EJB_MAP` table also contains details of the host operation type; whether the transaction is read or write, and a setting for backwards compatibility. When current versions of existing Financial Components are updated to add Store and Forward functionality they must be guaranteed to be able to be configured to work exactly as they used to work, that is, any new version of a Financial Component with no change apart from support for store and forward behavior must continue to work identically to the older version. This means the call to the Financial Process Integrator to determine if the host is online must always return true (even if the host is not online), to assure the online business logic is always invoked. This is done by setting the `ALWAYS_ONLINE` field to `Y`. The `STOREABLE` field is used to check if a transaction, that was initiated when the host was online but now encounters an offline host, should be stored, or if a `HostOfflineException` should be thrown instead. The `DestinationEjbMap` solution set layer is located in the `com.bankframe.ei.txnhandler.destinationejbmap` package and its implementation is in the `com.bankframe.ei.txnhandler.impl.destinationejbmap` package.

Table 29. DESTINATION_EJB_MAP table

EJB_NAME	EJB_OPERATION	DESTINATION_ID	OPERATION_TYPE	STOREABLE	ALWAYS_ONLINE
eontec.bp. retail. customersearch	retrieveCustomerDetailsBy AccountNumberAndBranchCode	C0004	READ	N	N

Store and Forward Classes and Package Structure

The Store and Forward solution is located in the `com.bankframe.ei.txnhandler.storeandforward` package and its implementation is in the `com.bankframe.ei.txnhandler.storeandforward.impl` package.

StoreAndForwardConstants

The Constants class for Store and Forward is located in the `com.bankframe.ei.txnhandler.storeandforward` package.

StoreAndForwardUtils

This class provides utility methods for allowing Financial Processes to use the store and forward features of the Financial Process Integrator and is located in the `com.bankframe.ei.txnhandler.storeandforward` package.

Table 30. StoreAndForwardUtils Methods

Method	Description
<code>isHostOnline(String ejbName, String</code>	This method takes two <code>Strings</code> , containing the name of the calling EJB and the name of the method, and determines if the host(s) used by the specified transaction is/are online.

Method	Description
<code>ejbOperation()</code>	
<code>isHostOnline(String ejbName, String ejbOperation, String companyCode)</code>	As above except it also takes a <code>String</code> containing the company code.
<code>setOffline()</code>	This method is used to force the host offline by setting the <code>hostStatus</code> to <code>FORCE_OFF_LINE</code> .
<code>setOnline()</code>	This method is used to update the host destinations to <code>ON_LINE</code> .
<code>transactionStoreable(String ejbName, String ejbOperation)</code>	This method determines if the specified transaction can be stored if the host goes offline, after it was initiated online.

isHostOnline() methods

This method allows the Financial Components to ascertain the host status when initiating a transaction in order to use the correct set of business rules as often differing rules will apply to online and offline transactions. In order to check the host status the `isHostOnline()` method is passed the name of the calling EJB and the name of the method being called. Using these values the method performs a look up on the `DESTINATION_EJB_MAP` table to get the host(s) destination(s) for the transaction as well as the transaction type. The method then performs the following checks:

- If the `ALWAYS_ONLINE` value is set to `Y` then `true` is returned
- If the `transactionHandler.storeAndForward.status` setting in the `BankframeResource.Properties` is set `OFF_LINE` and the operation type is `WRITE` then `false` is returned or if the operation type is `READ` a `HostOfflineException` is thrown
- If the `DESTINATION hostStatus` is `ON_LINE` `true` is returned
- If the `DESTINATION hostStatus` is `OFF_LINE` and the operation type is `WRITE` `false` is returned or if the operation type is `READ` a `HostOfflineException` is thrown
- If the `DESTINATION hostStatus` is `READ_ONLY` and the operation type is `WRITE` `false` is returned or if the operation type is `READ` `true` is returned

InternalStoreAndForwardUtils

This class provides utility methods for use by the Store and Forward features of the Financial Process Integrator and is located in the `com.bankframe.ei.txnhandler.storeandforward.impl` package.

Table 31. InternalStoreAndForwardUtils Methods

Method	Description
--------	-------------

Method	Description
<code>addToStore(DataPacket txnData)</code>	This method takes a <code>DataPacket</code> of request data and adds it to the store using the <code>StoreQueueBean</code> .
<code>convertSortedSetToString(SortedSet set)</code>	This method is a convenience method to convert a sorted <code>set</code> to a <code>String</code> that can be passed over HTTP using the channel management API. This is only to be used by Store and Forward because it assumes that the objects in the <code>set</code> are all of type <code>Integer</code> .
<code>convertStringToSortedSet(String string)</code>	This method is a convenience method to convert a <code>String</code> back to a sorted <code>set</code> . This is only to be used by Store and Forward because it assumes that the objects in the set are all of type <code>Integer</code> .
<code>getNextSequenceNo(String sequencePk)</code>	This method takes a <code>String</code> containing a primary key value to retrieve the next sequence number from the <code>SequenceGeneratorBean</code> . It does this by getting the current sequence number value and incrementing it by one then updating the table with the new value. Returns an <code>int</code> .
<code>hostDestinationStatus()</code>	This method checks to see if any of the host destinations in the <code>DESTINATION</code> table have been set to <code>OFF_LINE</code> or <code>FORCE_OFF_LINE</code> , if so it returns same, otherwise it returns <code>ON_LINE</code> . Returns a <code>String</code> containing the host status. (This method only checks the destination table).
<code>hostOnline()</code>	This method is used to determine the host status. It returns <code>true</code> if the host is online or <code>false</code> if it is offline.
<code>resetSequenceNo(String sequencePk)</code>	This method is used to reset the sequence number on the <code>SequenceGeneratorBean</code> initializing it back to 0.
<code>setAllDestinations(String status)</code>	This method takes a <code>String</code> containing a <code>status</code> to update all the host destinations with.
<code>updateDestination(String txnCode, String txnType)</code>	This method is used to update the host destination to <code>OFF_LINE</code> when a <code>HostConnectivityException</code> is encountered.

StoreTransactionBean

The host transactions are stored in a database table called `STORE_TRANSACTION` which is mapped by the `StoreTransactionBean`. The implementation of this bean is located in the package `com.bankframe.ei.txnhandler.storeandforward.impl.storetransaction`.

The request `DataPacket` is converted to a string to be stored using the `DPTPCodec` which is also used to convert it back into a `DataPacket`.

Table 32. STORE_TRANSACTION Table

SEQUENCE_NO	TIMESTAMP	REQUEST_TRANSACTION	BATCHED_FOR_FORWARD
Sequence number of the transaction.	Timestamp when the transaction is added to the store.	A string containing the request transaction details.	Boolean value which indicates if the transaction has already been added to a forwarding batch.

StoreQueueBean

This session bean is responsible for processing the transactions contained in the store. The implementation of this bean is located in the package

`com.bankframe.ei.txnhandler.storeandforward.impl.storequeue.`

Table 33. StoreQueueBean Methods

Method	Description
<code>addTransactionToCompleted(int sequenceNo)</code>	This method removes the transaction from the store queue and adds the transaction to the successful queue, with the given sequence number.
<code>addTransactionToError(int sequenceNo)</code>	This method removes the transaction from the store queue and adds it to the error queue, with the given sequence number.
<code>createStoredTransaction(Vector request)</code>	This method adds a new transaction to the store queue.
<code>findAllErrorTransactions()</code>	This method will find all the transactions on the error queue.
<code>findAllSuccessfulTransactions()</code>	This method will find all the transactions on the successful queue.
<code>isStoreEmpty()</code>	This method will determine if the store has transactions in it.
<code>removeTransactionFromError(int sequenceNo)</code>	This method removes the transaction from the error queue with the given <code>sequenceNo</code> .
<code>removeTransactionFromSuccessful(int sequenceNo)</code>	This method removes the transaction from the successful queue with the given <code>sequenceNo</code> .
<code>findAllStoredTransactions()</code>	This method will find all the transactions on the store queue.
<code>findNextStoredTransaction()</code>	This method will return the transaction at the head of the store queue.

Method	Description
<code>findStoredTransactionBySequenceNo(int sequenceNo)</code>	This method performs a lookup on the Store queue by <code>sequenceNo</code> .
<code>findStoredTransactionsInTimePeriod(long startTime, long endTime)</code>	This method performs a lookup on the store queue for a specified time period.
<code>nextStoredTransactionBatch()</code>	This method will returns a <code>Vector</code> containing a "-" delimited String of Sequence Numbers to be forwarded in the batch. This method also updates the <code>BATCHED_FOR_FORWARD</code> flag on the <code>STORE_TRANSACTION</code> from false to true to prevent the transaction from being added to any additional batches.

CompletedForwardTransactionBean

The completed host transactions are stored in a database table mapped by the `CompletedForwardTransactionBean`. There are two implementations of this bean located in the packages:

`com.bankframe.ei.txnhandler.storeandforward.completedforwardtransaction.impl.successfulltransaction` and
`com.bankframe.ei.txnhandler.storeandforward.completedforwardtransaction.impl.errortransaction`

SuccessfulTransactionBean

This entity maps to the `SUCCESSFUL_TRANSACTION` database and is used to record successfully forwarded transactions.

Table 34. SUCCESSFUL_TRANSACTION Table

SEQUENCE_NO	STORED_TIMESTAMP	COMPLETED_TIMESTAMP	REQUEST_TRANSACTION
Sequence number of the transaction.	Timestamp when the transaction is added to the store.	Timestamp when the transaction was forwarded successfully to the host.	A string containing the request transaction details.

ErrorTransactionBean

This entity maps to the `ERROR_TRANSACTION` database and is used to record host transactions which return a `ProcessingErrorException` when forwarded to the host.

Table 35. ERROR_TRANSACTION Table

SEQUENCE_NO	STORED_TIMESTAMP	ERROR_TIMESTAMP	REQUEST_TRANSACTION
Sequence number of the transaction.	Timestamp when the transaction is added to the store.	Timestamp when the transaction was forwarded erroneously to the host.	A string containing the request transaction details.

ForwardTransactionBean

This session bean is responsible for coordinating the forwarding of the stored host transactions. It is responsible for initiating the host status monitor and once the host is back online starting a thread to forward all the transactions.

Table 36. ForwardTransactionBean Methods

Method	Description
<code>forwardAll(String threadName)</code>	This method takes a String containing the name to call the Forwarding thread. It is used to forward all the transactions to the host. It will terminate when the queue is empty or if the queue goes offline.
<code>forwardAll(String threadName, int rate)</code>	This method takes a String containing the name to call the Forwarding thread and an int value which is the time interval to wait between each batch of transactions it forwards to the host. It will terminate when the queue is empty or if the queue goes offline.
<code>forwardSingle(String threadName, int sequenceNumber)</code>	This method takes a String containing the name to call the Forwarding thread. It will forward an individual request identified by the sequenceNumber from the queue.
<code>forwardSubset(String threadName, SortedSet transactions, int rate)</code>	This method takes a String containing the name to call the Forwarding thread. It will forward a SortedSet of stored transactions to the host in batches using the given time interval.
<code>setMonitorStatus(int rate)</code>	This method will set the status of the host monitor. This method assigns the rate parameter as the number of milliseconds to delay between each try to forward a request to the store. If this is set to -1 then the monitor is suspended.

ForwardOperationsBean

This session bean is responsible for controlling the rate at which transactions are forwarded to the host. It contains the following methods:

Table 37. ForwardOperationsBean Methods

Method	Description
<code>forwardNextRequest()</code>	This method will try and forward the request transaction at the head of the store queue. When the host is offline this method is used to check if it has gone back online by sending the request to the host and checking if it has been successfully sent.
<code>forwardRequest(int sequenceNumber)</code>	This method will try and forward a transaction by <code>sequenceNumber</code> .
<code>isStoreEmpty()</code>	This method will test if there are any requests on the store.
<code>updateDestination(String status)</code>	This method will amend the online/offline status of the destination associated with the transaction at the head of the store queue.

HostStatusMonitor

This thread class monitors the connection to the host system. It is used with the store class to determine whether requests in the store can be released to the host system. Every n seconds the thread will attempt to send a request to the host system. This will only happen if the store is non-empty. The class has the following constructors:

Table 38. HostStatusMonitor Constructors

Constructor	Description
<code>HostStatusMonitor()</code>	Default <code>HostStatusMonitor</code> constructor. It reads the <code>BankframeResource.properties</code> file for the monitor delay value.
<code>HostStatusMonitor(int delay)</code>	<code>HostStatusMonitor</code> constructor. This constructor takes an <code>int</code> value for the monitor delay.

The `HostStatusMonitor` class has the following methods:

Table 39. HostStatusMonitor Methods

Method	Description
<code>run()</code>	This method will check if the store is empty every n seconds. If it is and the host status is currently offline, then it tries to send a request from the store to the host system. If this request is successful then the online attribute of the destination entity corresponding to that host is set to true.
<code>setDelay(int newDelay)</code>	This method sets the time that the thread waits between checking the host status.

Method	Description
<code>start()</code>	This method starts the monitor thread at the lowest priority.
<code>stop()</code>	This method will shut down the thread.

ForwardingThread

This thread class will attempt to send a request to the host system. This class has the following constructors:

Table 40. ForwardingThread Constructors

Constructor	Description
<code>ForwardingThread()</code>	Forwarding thread constructor. This constructor reads the delay time from the <code>BankFrameResource.properties</code> file and is set to forward all transactions in the store.
<code>ForwardingThread(int delay)</code>	Forwarding thread constructor. This constructor takes the delay time from the passed parameter and is set to forward all transactions in the store.
<code>ForwardingThread(SortedSet list)</code>	Forwarding thread constructor. This constructor takes the delay time from the <code>BankframeResource.properties</code> file and is set to forward a passed subset of transactions in the store.
<code>ForwardingThread(SortedSet list, int delay)</code>	Forwarding thread constructor. This constructor takes the delay time from the passed parameter and is set to forward a passed subset of transactions in the store.

The ForwardingThread class has the following methods:

Table 41. ForwardingThread Methods

Method	Description
<code>forwardAll(ForwardOperations operations)</code>	This method will forward all the transactions in the store delaying for the specified delay time between each forward.
<code>forwardSubset(ForwardOperations operations)</code>	This method will forward a subset of the transactions in the store, delaying for the specified time between each forward.
<code>run()</code>	This method forwards transactions from the store.
<code>start()</code>	This method starts the forwarding thread at the lowest priority.

Forcing the Host Online or Offline

It must be possible to force the status of a host to online or offline. This is required for the following reasons:

- To test the store and forward functionality. Since a host is not available for testing, it must be possible to manually force the host online or offline.
- For maintenance reasons. The Financial Institution may want to restrict access to certain hosts to carry out maintenance on the host. The Financial Institution will want to be able to do this in an orderly manner.

The forwarding process should not be invoked and transactions should not attempt to be sent until the host has been forced back online. There are two `set...()` methods in the `StoreAndForwardUtils` class for setting the host either offline or online. The `setOffline()` method updates all the host destinations with a `hostStatus` of `FORCE_OFF_LINE`, this will ensure that the forwarding process will not be invoked until the `setOnline()` method has been used to set all the `hostStatus` back to `ON_LINE`.

Store and Forward Processing Exceptions

To apply the appropriate business logic the Financial Component must determine at the start of execution of the Financial Component whether the host is online or offline. Three new exception classes that extend the `ProcessingErrorException` class were added to MCA for Store and Forward:

HostConnectivityException

This class is located in the `com.bankframe.ei.txnhandler` package and is thrown when the Financial Process Integrator fails to connect to the host.

HostOfflineException

This class is located in the `com.bankframe.ei.txnhandler` package. There are two instances when this exception will be thrown:

- At the start of execution the host is determined to be online, but when the Financial Process Integrator attempts to post the transaction the host is offline. In this case the Financial Component will have applied the 'online' business rules, but the host is offline, however online transactions should never be stored.
- When the host is offline and a read transaction is attempted against the host.

HostProcessingErrorException

This class is located in the `com.bankframe.ei.txnhandler` package and is thrown when the host returns an error response.

Configuring Store and Forward Properties

The following transactionHandler.storeAndForward settings can be configured in the BankframeResource.properties file:

- **forwardingDelay.** This setting is used by the default constructor of the [ForwardingThread](#) to set the time interval, in milliseconds, between batches being sent to the host.
- **hostStatusDelay.** This setting is used by the default constructor of the [HostStatusMonitor](#) to set the time interval, in milliseconds, to wait between checks on the host status.
- **storeAndForward.url.** This setting is used to specify the URL of the [ForwardTransactionServlet](#). The URL should be specified in the format `http://localhost:7001/ForwardTransactionServlet`.
- **startHostMonitorAutomatically.** This setting is used to specify whether or not the [HostStatusMonitor](#) starts up automatically when the App server is started or not. It can have a setting of either `true` or `false`.
- **nextTransactionBatchAmount.** This setting is used to specify the amount of transactions the [ForwardingThread](#) is to forward in a batch.
- **storeAndForward.client.** This setting specifies the client type to use for [HostStatusMonitor](#). The value should be of the format `com.bankframe.ei.txnhandler.storeandforward.StoreAndForwardHttpClient`.

Configuring Deployment Descriptors for Store and Forward

This topic details the deployment descriptor configurations required to enable store and forward.

To configure the WebLogic Deployment Descriptor for Store and Forward

- Set the weblogic-ejb-jar value to `<concurrency-strategy>Database</concurrency-strategy>`

This change allows updates to be persisted.

To configure the WebSphere Deployment Descriptor for Store and Forward

- Modify the select statement for the findNextTransactionBatch finder on the StoreTransaction EJB as follows:

```
select object(o) from StoreTransactionCMPBean o where o.sequenceNo >= ?1 and
o.sequenceNo <= ?2
```

Implementing Store and Forward

Familiarity with the Financial Process Integrator and EJB lifecycle are required before attempting a Store and Forward implementation.

StoreAndForwardPersister

This persister class extends from the `TxnPersister` class. The class overwrites the `TxnPersisters processTxnRequest()` and the `amend()` method.

`processTxnRequest(EBMPEntity entityBean, DataPacket txnData, String cachePolicy)`

This protected method is called by the `find()` method. It is responsible for passing the transaction details to the Financial Process Integrator, receiving the response, placing it in the cache and returning an enumeration of primary keys. The `StoreAndForwardPersister` version also checks the host status against the host status when the transaction was initiated, this is so the persister will know whether to store the transaction, send it to the host or throw an exception.

```
protected Enumeration processTxnRequest(EBMPEntity entityBean, DataPacket txnData,
String cachePolicy) throws ProcessingErrorException {
    try {
        Vector entityPk = new Vector();
        String txnCode = txnData.getString(TransactionHandlerConstants.TXN_CODE);
        String hostStatus =
            txnData.getString(StoreAndForwardConstants.HOST_ONLINE_STATUS);

        if (StoreAndForwardUtils.hostOnline()) {
            if ((txnCode == null) ||
                txnCode.equalsIgnoreCase(TransactionHandlerConstants.FIELD_NA)) {
                // do nothing
            }
            else {
                //Get an instance of the transaction handler and send the transaction //data to the
                processFindRequest() method.
                TransactionHandler transactionHandler = this.getTxnHandler();
                try {
                    map = transactionHandler.processFindRequest(txnData);
                }
                catch (HostProcessingErrorException hpex) {
                    throw new ProcessingErrorException(hpex);
                }
                catch (HostConnectivityException hcex) {
                    BankFrameLog.log(BankFrameLog.DEBUG,
                        BankFrameLogConstants.TXNHANDLER_SUBSYSTEM, "Store Persister::processTxnRequest::
                        HostConnectivityException");
                    StoreAndForwardUtils.updateDestination(txnCode,
                        txnData.getString(TransactionHandlerConstants.TXN_TYPE),
                        StoreAndForwardConstants.OFF_LINE);
                    throw new ProcessingErrorException(new
                        BankFrameMessage(HOST_OFFLINE_EXCEPTION));
                }
            }
        }
    }
}
```

```

        boolean persistant;

//Before caching the data check to see if it is persistent or not. //Persistent
data will be written to a database as well as to memory.

        if
(cachePolicy.equalsIgnoreCase(TxnPersisterConstants.CACHE_PERSISTENT)) {
            persistant = true;
        }
        else if
(cachePolicy.equalsIgnoreCase(TxnPersisterConstants.CACHE_NON_PERSISTENT) ||
cachePolicy.equalsIgnoreCase(TxnPersisterConstants.NOT_CACHED)) {
            persistant = false;
        }
        else {
            //throw an exception
        }

//get the timeout value for the data and then store it in the cache.
        long timeOutValue = new
Long(txnData.getString(PersisterTxnMapConstants.TIME_OUT_VALUE)).longValue();
        this.storeInCache(map, timeOutValue, persistant);

//Process the keys of the map returned from the transaction handler to //return an
enumeration of primary keys.
        Set keys = map.keySet();
        Enumeration enum = Collections.enumeration(keys);
        while (enum.hasMoreElements()) {
            EPrimaryKey pk = entityBean.createPrimaryKey((DataPacket)
enum.nextElement());
            if (pk != null) {
                entityPk.addElement(pk);
            }
        }
        return new IteratorEnumeration(entityPk.iterator());
    }
    else {
        BankFrameLog.log(BankFrameLog.DEBUG,
BankFrameLogConstants.TXNHANDLER_SUBSYSTEM, "TxnPersister::processTxnRequest::
offline");

        throw new ProcessingErrorException(new
BankFrameMessage(HOST_OFFLINE_EXCEPTION));
    } catch (CreateException ce) {
        throw new ProcessingErrorException(ce);
    }

```

```

    }
    catch (RemoteException re) {
        throw new ProcessingErrorException(re);
    }
}

```

amend(EBMPEntity entityBean, String methodName, DataPacket data, Vector primaryKeys, boolean removeOperation)

The protected `amend()` method is called by the persister's `amend...()` method. The `amend()` method checks if the transaction policy is set to `CACHE_ONLY`, if it is then it will only update the cache, otherwise it adds the transaction code and the transaction type to a `DataPacket` containing the entity bean's update attributes and sends the `DataPacket` to the Financial Process Integrator. It also takes a `boolean` value which indicates if a remove operation is to be carried out on the host or from the cache. The `amend()` method is used for updating some or all of an entity's attributes. The `StoreAndForwardPersister` version also checks the current host status against the host status when the transaction was initiated, this is so the persister will know whether to store the transaction, send it to the host or throw an exception.

`protected void amend(EBMPEntity entityBean, String methodName, DataPacket data, Vector primaryKeys, boolean removeOperation) throws ProcessingErrorException, HostOfflineException, HostConnectivityException {`

```

    //DataPacket of data to be updated on the host
    DataPacket update = new DataPacket(data.DATA_PACKET_NAME);
    //Get txnCode and txnType from PERSISTER_TXN_MAP
    DataPacket amendData = this.mapTxn(entityBean.getEntityName(), methodName);
    String txnCode = amendData.getString(TransactionHandlerConstants.TXN_CODE);
    String txnType = amendData.getString(TransactionHandlerConstants.TXN_TYPE);
    long timeOutValue = new
Long(amendData.getString(PersisterTxnMapConstants.TIME_OUT_VALUE)).longValue();
    //the host status when the transaction was initiated
    String hostTransactionStatus =
data.getString(StoreAndForwardConstants.HOST_ONLINE_STATUS);
    try {

        update.append(update, data);
        //Add txnCode and txnType
        update.put(TransactionHandlerConstants.TXN_CODE, txnCode);
        update.put(TransactionHandlerConstants.TXN_TYPE, txnType);

        //check the host online status
        String hostOnlineStatus = StoreAndForwardUtils.hostOnline();
        boolean storeable = StoreAndForwardUtils.transactionStoreable(txnCode,
txnType);
        //if the host is offline and an offline transaction was initiated store the
transaction
        if (hostOnlineStatus == StoreAndForwardConstants.OFF_LINE &&
hostTransactionStatus == StoreAndForwardConstants.OFF_LINE) {
            StoreAndForwardUtils.addToStore(update);
        }
        //if an online transaction was initiated but the host is offline
        else if (hostOnlineStatus == StoreAndForwardConstants.OFF_LINE &&
hostTransactionStatus == StoreAndForwardConstants.ON_LINE) {
            throw new HostOfflineException(new
BankFrameMessage(HOST_OFFLINE_EXCEPTION));

```

```

    }
    //otherwise forward the transaction to the host
    else {
        if (getIgnoreHost(txnCode) == false) {

            TransactionHandler transactionHandler = this.getTxnHandler();

            try {
                transactionHandler.processRequest(update);
            }
            catch (HostProcessingErrorException hpex) {
                throw new ProcessingErrorException(hpex);
            }
            catch (HostConnectivityException hcex) {
                StoreAndForwardUtils.updateDestination(txnCode, txnType,
StoreAndForwardConstants.OFF_LINE);
                throw new HostConnectivityException(new
BankFrameMessage(HOST_CONNECTIVITY_EXCEPTION));
            }
        }

        if (removeOperation || getRemoveFromCache()) {
            this.removeFromCache(primaryKeys);
        }
        else {
            //put data into a map (same data used for each primary key):
            Map entityMap = new HashMap();
            for (int index = 0; index < primaryKeys.size(); index++) {
                entityMap.put(primaryKeys.elementAt(index), data);
            }
            String cachePolicy =
amendData.getString(PersisterTxnMapConstants.CACHE_POLICY);
            boolean bCachePolicy =
(cachePolicy.equalsIgnoreCase(TxnPersisterConstants.CACHE_PERSISTENT)) ? true : false;
            this.storeInCache(entityMap, timeOutValue, bCachePolicy);
        }
    }
}
catch (CreateException ce) {
    throw new ProcessingErrorException(ce);
}
catch (RemoteException re) {
    throw new ProcessingErrorException(re);
}
}
}

```

Teller Example of Store and Forward

One of the Financial Components of Teller to be enhanced with Store and Forward is Deposit. The changes to the deposit component are as follows:

TransactionDetails

The BMP version of this bean was written implementing the [com.bankframe.ejb.bmp.EBMPEntity](#) interface, for details on this please refer to the Persister documentation. A new variable [hostOnLineStatus](#) was added to the BMP class to pass along the host status at the time the

transaction was initiated. This variable is used to determine if the transaction should be processed or if a `HostOfflineException` should be thrown, depending on the host status.

IsSystemAvailabilityBean

This session bean is used to interact with the `StoreAndForwardUtils` class to ascertain the host status. It contains the following two methods:

- `imIsHostOnline(String sessionName, String processName, String companyCode)` this method is used to check if the Host is offline or online.
- `imIsTransactionStoreable(String sessionName, String processName)` this method is used to check whether or not a transaction can be stored.

IsMakeDeposit

This has been changed to throw new transaction handler exceptions: `HostConnectivityException` and `HostOfflineException`.

MakeDeposit

The `makeDepositBC` method was updated as follows: to process an online transaction when `true` is returned from the `imIsHostOnline()` method and to process an offline transaction when `false` is returned from the `imIsHostOnline()` method. One of the requirements for Store and Forward is the status of the host at the time the transaction was initiated. If the host was online when the transaction was started but has subsequently gone offline either a `HostConnectivityException` or a `HostOfflineException` will be thrown. In the example below these exceptions are caught and if the transaction is storeable then an offline transaction is sent to the host, otherwise the exception is re-thrown.

```
public Vector makeDepositBC(FinancialTransactionCommonAttributesVO
financialTransactionCommonAttributesVO, FinancialTransactionDestinationAccountVO
financialTransactionDestinationAccountVO, Vector
financialTransactionNegotiableInstrumentVOVector) throws
ProcessingErrorException, ValidationException, HostOfflineException,
HostConnectivityException {

    Vector batchStateMessageVector = new Vector();

    try {

        //check the host status

        this.online =
this.getIsSystemAvailability().imIsHostOnline(MakeDepositHome.JNDI_LOOKUP_NAME,
"makeDepositBC",
financialTransactionCommonAttributesVO.getCompanyCode()).booleanValue();

        //if the host is online try to send an online request

        if (online) {
```

```

        this.getUserAdministration().imIsUserValidForOperation(
financialTransactionCommonAttributesVO.getCompanyCode(),
financialTransactionCommonAttributesVO.getUserId(),
com.bankframe.bfa.Constants.getValueInList(0,
TellerConstantsKeysImpl.TASK_ID_MAKE_DEPOSIT_ONLINE).toString(),
DataTypeConvertor.getDouble(com.bankframe.bfa.Constants.getText(TellerConstantsKe
ysImpl.DEFAULT_LIMIT_VALUE_TEXT)));

        try {

            batchStateMessageVector =
this.getIsMakeDeposit().imMakeOnlineDepositBC(financialTransactionCommonAttribute
sVO, financialTransactionDestinationAccountVO,
financialTransactionNegotiableInstrumentVOVector, "ON_LINE");

        }

        //if a HostConnectivityException is returned then check if
//the transaction is storeable

        catch (HostConnectivityException hex) {

            if
(this.getIsSystemAvailability().imIsTransactionStoreable(MakeDepositHome.JNDI_LOO
KUP_NAME, "makeDepositBC").booleanValue()) {

this.getUserAdministration().imIsUserValidForOperation(financialTransactionCommon
AttributesVO.getCompanyCode(),
financialTransactionCommonAttributesVO.getUserId(),
com.bankframe.bfa.Constants.getValueInList(0,
TellerConstantsKeysImpl.TASK_ID_MAKE_DEPOSIT_OFFLINE).toString(),
DataTypeConvertor.getDouble(com.bankframe.bfa.Constants.getText(TellerConstantsKe
ysImpl.DEFAULT_LIMIT_VALUE_TEXT)));

            batchStateMessageVector =
this.getIsMakeDeposit().imMakeOfflineDepositBC(financialTransactionCommonAttribut
esVO, financialTransactionDestinationAccountVO,
financialTransactionNegotiableInstrumentVOVector, "OFF_LINE");

        }

        else

            throw new HostConnectivityException(hex);

```

```

    }

    //if a HostOfflineException is returned then check if the
    //transaction is storeable

    catch (HostOfflineException hex) {

        if
        (this.getIsSystemAvailability().imIsTransactionStoreable(MakeDepositHome.JNDI_LOO
        KUP_NAME, "makeDepositBC").booleanValue()) {

            this.getUserAdministration().imIsUserValidForOperation(financialTransactionCommon
            AttributesVO.getCompanyCode(),
            financialTransactionCommonAttributesVO.getUserId(),
            com.bankframe.bfa.Constants.getValueInList(0,
            TellerConstantsKeysImpl.TASK_ID_MAKE_DEPOSIT_OFFLINE).toString(),
            DataTypeConvertor.getDouble(com.bankframe.bfa.Constants.getText(TellerConstantsKe
            ysImpl.DEFAULT_LIMIT_VALUE_TEXT)));

            batchStateMessageVector =
            this.getIsMakeDeposit().imMakeOfflineDepositBC(financialTransactionCommonAttribut
            esVO, financialTransactionDestinationAccountVO,
            financialTransactionNegotiableInstrumentVOVector, "OFF_LINE");

        }

        else

            throw new HostOfflineException(hex);

    }

    //if the host is offline send an offline request

    else if (!online) {

        this.getUserAdministration().imIsUserValidForOperation(financialTransactionCommon
        AttributesVO.getCompanyCode(),
        financialTransactionCommonAttributesVO.getUserId(),
        com.bankframe.bfa.Constants.getValueInList(0,
        TellerConstantsKeysImpl.TASK_ID_MAKE_DEPOSIT_OFFLINE).toString(),
        DataTypeConvertor.getDouble(com.bankframe.bfa.Constants.getText(TellerConstantsKe
        ysImpl.DEFAULT_LIMIT_VALUE_TEXT)));
    }

```



```

        batchStateMessageVector =
this.getIsMakeDeposit().imMakeOfflineDepositBC(financialTransactionCommonAttribut
esVO, financialTransactionDestinationAccountVO,
financialTransactionNegotiableInstrumentVOVector, "OFF_LINE");

    }

}

catch (RemoteException remoteException) {

    //throw new
ProcessingErrorException(TellerErrorNumberImpl.REMOTE_EXCEPTION_NUMBER, new
String[] { "MakeDeposit", "makeDepositBC" });

    throw new ProcessingErrorException(remoteException);

}

return batchStateMessageVector;

}

```

MaintainFinancialTransaction

Has been changed to throw new transaction handler exceptions; [HostConnectivityException](#) and [HostOfflineException](#).

About Branch Teller Offline Transaction Processing

This topic provides an overview of Branch Teller Offline Transaction Processing. A transaction is written to the store is if the host is offline or if a [HostConnectivityException](#) is encountered. The session EJBs should be coded as per the MakeDeposit example.

Processing a Timeout between SRF and the Host System

If there is a timeout between SRF and the host system it should be detected when the connector tries to post the transaction. It should throw a [ProcessingErrorException](#) which is converted to a [HostConnectivityException](#). If you are using the [StoreAndForwardPersister](#) then this [HostConnectivityException](#) is caught and it marks the host as offline. You need to use the [StoreAndForwardPersister](#) if you want to use level two offline support. If a new connector is developed it should throw an exception if it cannot post the transaction correctly.

Store and Forward Mid-Tier Processing when the Host is Offline

The host will be marked as offline by the [StoreAndForwardPersister](#). The mid-tier session has to be written to take a different path depending on whether the host is offline or online. The MakeDeposit session in the MCA Services Developer guide provides a good example.

Processing Failed Stored Transactions

If a stored transaction fails it is written to an error queue. The custom implementation will determine how the transaction in the error queue is handled.

Financial Process Integrator Sample Implementations

Extracting the Source Code for the FPI Examples

The source code for the FPI examples referred to in this topic is provided on the software CD for reference as follows:

- If you have licensed the SRF banking application the FPI example files are available on the Common Software Resources CD, typically located at [siebel\Common\mcaresources\examples.jar](#).
- If you have licensed Siebel Foundation Services the FPI example files are available on the Foundation Services CD, typically located at [FoundationServices\examples\examples.jar](#)

This source code is for reference purposes only and should not be amended.

Launching the FPI Examples

To launch the Financial Process Integrator examples, the application server must be configured and the EAR must be deployed. For a banking application deployment refer to the banking application installation guide. For a Foundation Services deployment refer to the MCA Services installation guide.

The Financial Process Integrator examples can be launched from the following URL:

http://localhost:<port_number>/BankFrameMCA/CustomSearchServlet

The CustomerSearch Example

This example illustrates how the Financial Process Integrator works using two entity beans and a session bean:

Name	EJB Type	Description
Address	Entity	Models the common attributes of a postal address
Customer	Entity	Models the name attributes of a customer
CustomerSearch	Session	Searches for Customer instances and their associated Address instances Allows Customer and Address details to be amended

These examples illustrate the following:

- How entity beans interact with the persister
- How the persister interacts with the Financial Process Integrator
- How to configure the Financial Process Integrator metadata
- How to configure the Financial Process Integrator routes and destinations

Scope

It is assumed that the reader is familiar with the best practices for modeling entity beans and session beans.

The Address Entity EJB

The `Address` entity has the following attributes:

Attribute	Description
<code>ownerId</code>	The ID of the entity that the address belongs to
<code>addressLine1</code>	The first line of the address
<code>addressLine2</code>	The second line of the address
<code>addressLine3</code>	The third line of the address
<code>addressLine4</code>	The fourth line of the address
<code>country</code>	The country of the address
<code>postcode</code>	The postal code

The Customer Entity EJB

The `Customer` entity has the following attributes:

Attribute	Description
<code>ownerId</code>	The customer's unique ID number
<code>title</code>	The customer's formal title
<code>firstName</code>	The customer's first name
<code>lastName</code>	The customer's last name

Relationship between the Customer and Address Entity EJBs

Every [Customer](#) entity must have an associated [Address](#) entity. This means that a [Customer](#) entity cannot exist without having a corresponding [Address](#) entity. The existence of an [Address](#) entity is dependent on the existence of a [Customer](#) entity. Each [Customer](#) entity has a unique [ownerId](#) attribute. For each [Customer](#) entity there will be a corresponding [Address](#) entity whose [ownerId](#) is equal to the Customer's [ownerId](#) attribute.

The CustomerSearch Session EJB

The [CustomerSearch](#) session EJB must be able to:

- Find a Customer by [ownerId](#)
- Find one or more [Customers](#) by last name
- Find one or more Customers by first name. This finder method uses cache indexing as there is no corresponding host transaction to do a lookup by customer first name. The example configuration has the response from find by last name being indexed by first name.
- Amend [Customer](#) details, including [Address](#) details.

Interfacing the Entities with the Financial Process Integrator

This topic describes how the [Address](#) entity bean has been modelled, concentrating on issues relevant to connecting the entity bean to the Financial Process Integrator.

com.bankframe.examples.impl.address.AddressBMPBean

This class is the Bean Managed Persistence (BMP) implementation of the [Address](#) entity bean.

This class must persist its attributes to/from the host system.

EBMPEntity Methods

As described previously all BMP entity beans must implement the

[com.bankframe.ejb.bmp.EBMPEntity](#) interface. This topic describes how [AddressBMPBean](#) implements each of the methods defined in the [EBMPEntity](#) interface.

createPrimaryKey()

```
public EPrimaryKey createPrimaryKey(DataPacket dp) throws
ProcessingErrorException {

    if ( dp.getName().equals("ADDRESS") ) {

        return new AddressPK(dp.getString("OWNER_ID"));

    } else {

        return null;

    }
}
```

```
}
```

This method must create an instance of the entity bean's primary key type from the information in the supplied `DataPacket`. The method must check that the `DataPacket` being passed in is of the correct type, that is, that the `DataPacket` name matches the entity bean's name.

getEntityName()

```
public String getEntityName() {
    return AddressHome.JNDI_NAME;
}
```

This method must provide a `String` that uniquely identifies this *type* of entity bean. By convention this name must be the JNDI name of the entity bean.

getPersister()

```
public EPersister getPersister() {
    try {
        return EPersisterFactory.getPersister(this.getEntityName());
    } catch ( ProcessingErrorException pex ) {
        BankFrameLog.log(BankFrameLog.WARN, "BANKFRAME.MCA", pex);
        throw new RuntimeException(pex.getMessage());
    }
}
```

This method must return an instance of the persister object to be used for persisting this entity bean. This method delegates the task of locating the persister to the `com.bankframe.ejb.bmp.EPersisterFactory` class. This enables the persister used by an entity bean to be changed without having to recompile or re-deploy the entity bean. Note that all BMP entity beans will use the exact code shown above.

getPrimaryKey()

```
public EPrimaryKey getPrimaryKey() {
    return (EPrimaryKey)this.ctx.getPrimaryKey();
}
```

This method must return an instance of this entity bean instance's primary key object. The primary key for each entity bean instance is stored in the entity bean's `EntityContext`, so this method just returns the primary key reference stored in the `EntityContext`. Note that all BMP entity beans will use the exact code shown above.

populate()

```

public void populate(DataPacket dp) {

    this.ownerId = dp.getString("OWNER_ID");

    this.addressLine1 = dp.getString("ADDRESS_LINE1");

    this.addressLine2 = dp.getString("ADDRESS_LINE2");

    this.addressLine3 = dp.getString("ADDRESS_LINE3");

    this.addressLine4 = dp.getString("ADDRESS_LINE4");

    this.country = dp.getString("COUNTRY");

    this.postCode = dp.getString("POST_CODE");

}

```

This method must populate the entity bean's attributes with the data retrieved from the supplied `DataPacket`.

AddressBMPBean Methods

`AddressBMPBean` must implement the methods required by the `javax.ejb.EntityBean` interface; this is described below.

ejbActivate(). This method is called by the EJB container when an entity bean instance is about to be used. In this method the entity bean should acquire any resources it requires. The `AddressBMPBean` does not need to acquire any resources so this method is empty.

ejbCreate(). This method is called by the EJB container when a new entity bean instance is being created. This method must create the corresponding data in the data-store, and return a primary key object for the new instance.

```

public AddressPK ejbCreate(String ownerId,String addressLine1, String
addressLine2, String addressLine3, String addressLine4,String
country,String postCode) throws CreateException,ValidationException,
ProcessingErrorException {

    super.create(ownerId,addressLine1,addressLine2,addressLine3,addressLine4,
country,postCode);

    return (AddressPK)this.getPersister().create(this);

}

```

This method first calls the super-classes' `create()` method to initialise the new instance. It then calls the persister's `create()` method, which takes care of creating the data on the host system. The persister's `create()` method also returns a primary key object for the new instance.

ejbLoad() . This method is called by the EJB container when the entity bean must refresh its attributes from the data-store.

```
public void ejbLoad() {
    try {
        this.getPersister().load(this);
    } catch ( ProcessingErrorException pex ) {
        BankFrameLog.log(BankFrameLog.WARN, "BANKFRAME.MCA",pex);
    }
}
```

This method calls the persister's `load()` method, which takes care of reading the data from the data-store, and calling the entity bean's `populate()` method to initialise the entity bean's attributes.

ejbPassivate(). This method is called by the EJB container when an entity bean instance is about to be de-activated. In this method the entity bean should release any resources it has been using. The `AddressBMPBean` does not use any resources, so this method is empty.

ejbPostCreate(). This method is called by the EJB container immediately after a new entity bean instance has been created.

```
public void ejbPostCreate(String ownerId,String addressLine1, String
addressLine2, String addressLine3, String addressLine4,String
country,String postCode) {
    setModified(false); // reset the modified status
}
```

This method sets the `modified` flag to `false`. All BMP entity beans will use the exact code shown above.

ejbRemove(). This method is called by the EJB container when the entity bean instance should be deleted from the data store.

```
public void ejbRemove() {
    try {
        this.getPersister().remove(this);
    } catch ( ProcessingErrorException pex ) {
        BankFrameLog.log(BankFrameLog.WARN, "BANKFRAME.MCA",pex);
    }
}
```

This method calls the persister's `remove()` method to remove the data from the data store. All BMP entity beans will use the exact code shown above.

ejbStore(). This method is called by the EJB container when the entity bean instance should be written to the data store.

```
public void ejbStore() {
    try {
        if ( this.modified == true ) {
            this.getPersister().store(this);
            this.setModified(false);
        }
    } catch ( ProcessingErrorException pex ) {
        BankFrameLog.log(BankFrameLog.WARN, "BANKFRAME.MCA",pex);
    }
}
```

This method calls the persister's `store()` method to actually store the data to the data store. This method will only call the persister's `store()` method if the `modified` flag is set to `true`. This is an optimisation to prevent unnecessary updates to the data store. It is imperative that all entity bean methods which modify an entity bean's attributes must set the `modified` flag to `true`.

All BMP entity beans will use the exact code shown above.

setEntityContext(). This method is called by the EJB Container when an entity bean is about to be used. The entity bean must store the supplied context.

```
public void setEntityContext(EntityContext newCtx) {
    this.ctx = newCtx;
    this.setModified(false);
}
```

All BMP entity beans will use the exact code shown above.

unsetEntityContext(). This method is called by the EJB Container when it is finished using an entity bean. The entity bean must null its context.

```
public void unsetEntityContext() {
    this.ctx = null;
}
```

All BMP entity beans will use the exact code shown above.

ejbFindByPrimaryKey(). This method is called by the EJB Container when a `findByPrimaryKey()` is invoked on the entity bean's home interface. It must verify that the specified entity bean instance exists in the data store. If it does exist then this method must return the supplied primary key, otherwise this method must throw a `javax.ejb.ObjectNotFoundException`

```
public AddressPK ejbFindByPrimaryKey(AddressPK primaryKey) throws
FinderException, ValidationException {

    try {

        this.validator.validateOwnerId(primaryKey.ownerId);

        Enumeration enum =
this.getPersister().find(this,"findByPrimaryKey",primaryKey.toDataPacket(
));

        if ( enum.hasMoreElements() == false ) {

            throw new
javax.ejb.ObjectNotFoundException(primaryKey.toDataPacket().toString());

        }

        return primaryKey;

    } catch ( ProcessingErrorException pex ) {

        throw new FinderException(pex.getMessage());

    }

}
```

This method first validates that the primary key object contains a legal value for the `ownerId`. It then calls the persister's `find()` method passing it the primary key object in `DataPacket` form. If the instance does exist then the `find()` method will return an `Enumeration` containing the entity's primary key object, otherwise it will return an empty `Enumeration`. If the `Enumeration` is empty then this method will throw a `javax.ejb.ObjectNotFoundException()`.

ejbFind<Method>() Methods. The `AddressBMPBean` has a number of `ejbFind<Method>()` methods, each of which has a corresponding method in the `AddressHome` interface. These methods must return an `Enumeration` of primary key objects that match the specified search criteria. If no matches are found then it must return an empty `Enumeration`.

Below is the code for the `ejbFindByPostCode()` method.

```
public Enumeration ejbFindByPostCode(String postCode) throws
FinderException, ValidationException {

    try {

        this.validator.validatePostCode(postCode);
```

```

        DataPacket dp = new DataPacket("FIND_BY_POST_CODE");

        dp.put("POST_CODE",postCode);

        return this.getPersister().find(this,"findByPostCode",dp);

    } catch ( ProcessingErrorException pex ) {

        throw new FinderException(pex.getMessage());

    }

}

```

The amend() Method

All Siebel Retail Finance entity beans have an `amend()` method which is used to modify the entity bean's attributes.

```

    public void amend(String addressLine1, String addressLine2, String
        addressLine3, String addressLine4,String country,String postCode) throws
        ValidationException {

        super.amend(addressLine1,addressLine2,addressLine3,addressLine4,country,p
            ostCode);

        this.setModified(true);

    }

```

This method calls its super-classes' `amend()` method to actually perform the amend, and then sets the `modified` flag to `true`.

CustomerBMPBean Methods

The `com.bankframe.examples.bo.impl.customer.CustomerBMPBean` has very similar methods to the `AddressBMPBean`. The only difference is some extra methods to handle the relationship between `Customer` Entities and `Address` Entities.

Modeling the Customer and Address Entity Relationship

The `Customer` entity is called a master entity because it has an associated entity (or dependent entity) that cannot exist by itself. An `Address` entity cannot exist without a corresponding `Customer` entity also existing.

In a real system, an `Address` entity could be associated with other types of entity other than a `Customer` entity; for example, an `Address` entity could be associated with a `BranchOffice` entity. This means that an `Address` entity cannot know which entity it is associated with.

The example host system has only one amend transaction, which must be used for amending both [Customer](#) and [Address](#) information. This transaction requires that all the attributes from the [Customer](#) entity and the [Address](#) entity be present in the transaction. Therefore, to generate the transaction the data from the [Customer](#) and [Address](#) entities must be merged.

When amending the [Customer](#) entity it is straightforward to locate the corresponding [Address](#) entity, and merge the two entities to produce the complete amend transaction.

However when amending the [Address](#) entity it cannot be determined which entity it is associated with. This means that the [Address](#) entity does not have enough information to create a complete amend transaction.

The solution is to add a new method to the [Customer](#) method called `amendAddress()`. This method is used when the [Address](#) details associated with a [Customer](#) must be updated. This method takes care of locating the [Address](#) associated with the [Customer](#) and calling the [Address](#)'s `amend()` method, and then merging the data from the two entities to create the complete amend transaction required by the host.

There is a second problem which must also be addressed: since data is cached when it is read from the host, old entries must be removed from the cache when entities are amended. When an address is amended, it must be removed from the cache and its associated [Customer](#) must also be removed from the cache.

To make sure this happens the [Customer](#) entity must implement the `com.bankframe.ejb.bmp.EBMPMasterEntity` interface, and must also be configured to use the `com.bankframe.ei.txnhandler.persister.MasterEntityPersister` persister. `MasterEntityPersister` extends `CacheIndexPersister` therefore the EJBs using it support cache indexing.

CustomerBean Methods

The `com.bankframe.examples.bo.impl.customer.CustomerBean` has the following methods to model the relationship between the [Customer](#) entity and the [Address](#) entity.

getAddress()

This method returns an instance of the [Address](#) entity associated with the [Customer](#) entity.

```
try {

    AddressHome home =
        (AddressHome)ObjectLookup.lookup(AddressHome.JNDI_NAME,AddressHome.class)
    ;

    return home.findByPrimaryKey(new AddressPK(this.ownerId));

} catch ( javax.ejb.FinderException fex ) {

    BankFrameLog.log(BankFrameLog.WARN, "BANKFRAME.MCA",fex);

    throw ExceptionUtils.toProcessingErrorException(fex);

} catch ( ValidationException vex ) {
```

```

        BankFrameLog.log(BankFrameLog.WARN, "BANKFRAME.MCA",vex);

        throw ExceptionUtils.toProcessingErrorException(vex);

    }

```

This method does a `findByPrimaryKey()` to find the corresponding `Address` instance and returns the corresponding instance.

amendAddress()

This method amends the `Address` entity associated with the `Customer` entity. This method should be called rather than calling the `Address.amend()` method directly.

```

    public Address amendAddress(String addressLine1, String addressLine2,
        String addressLine3, String addressLine4,String country,String postCode)
        throws ProcessingErrorException, ValidationException, RemoteException {

        Address address = this.getAddress();

        address.amend(addressLine1,addressLine2,addressLine3,addressLine4,country
        ,postCode);

        this.setModified(true);

        return address;

    }

```

This method sets the `modified` flag to `true` so that the `Customer` entity data is stored to the host. This will cause the `Address` entity data to be stored as well.

CustomerBMPBean Methods

The `CustomerBMPBean` class has very similar methods to the `AddressBMPBean`. The only extra methods are for managing the `Customer-Address` relationship.

Since `CustomerBMPBean` is a master entity, it must implement the `com.bankframe.ejb.bmp.EBMPMasterEntity` interface. The `EBMPMasterEntity` interface extends the `com.bankframe.ejb.bmp.EBMPEntity` interface, adding the following method:

```

    public Vector getDependentEntities() throws ProcessingErrorException,
        RemoteException ;

```

This method must return a `Vector` of `com.bankframe.ejb.bmp.EEntity` instances, where each instance is a dependent entity of the master entity.

Below is `CustomerBMPBean`'s implementation of this method:

```

public Vector getDependentEntities() throws ProcessingErrorException,
RemoteException {

    Vector dependents = new Vector(1);

    dependents.add(this.getAddress());

    return dependents;

}

```

This method calls the `getAddress()` method and adds the returned instance to the `Vector` of dependent instances.

Configuring the PERSISTER_TXN_MAP Table for CustomerSearch

The `PERSISTER_TXN_MAP` database table must be correctly configured to connect the BMP (bean managed persistence) entity beans to the Financial Process Integrator. Table 42 below illustrates the data used to configure the `PERSISTER_TXN_MAP` table for the `Customer` and `Address` entities. These entities use different persister classes in their examples. Cache indexing is used in the `Customer` entity where `findByFirstName` looks up the cache populated by the `findByLastName` responses.

Table 42. `PERSISTER_TXN_MAP` Database

ENTITYNAME	METHODNAME	TXNCODE	TXNTYPE	CACHE POLICY	TIMEOUT VALUE	INDEX NAME
eontec.bankframe.examples.bo.customer	findByPrimaryKey	TESTFIND0001	TEST	memory	100000	
eontec.bankframe.examples.bo.customer	findByLastName	TESTFIND0002	TEST	none	100000	
eontec.bankframe.examples.bo.customer	findAll	TESTFIND0004	TEST	none	100000	
eontec.bankframe.examples.bo.customer	store	TESTAMND0001	TEST	none	100000	
eontec.bankframe.examples.bo.address	findByPrimaryKey	TESTFIND0001	TEST	memory	100000	

ENTITYNAME	METHODNAME	TXNCODE	TXNTYPE	CACHE POLICY	TIMEOUT VALUE	INDEX NAME
eontec.bankframe. examples.bo.address	store	NA	NA	memory	100000	
eontec.bankframe. examples.bo.customer	amendAddress	TESTAMND001	TEST	none	100000	
eontec.bankframe. examples.bo.customer	findByFirstName	NA	NA	memory	100000	CUSTOMER_FIRST_NAME_INDEX

This table maps entity names and method names to transaction codes and transaction types. Note that some entity name, method name pairs may be mapped to a special transaction code: 'NA'. The 'NA' value indicates that the specified method is not connected to the Financial Process Integrator. In the above example the `store()` method for the `Address` entity is marked 'NA' because the `Address` entity is unable to persist itself.

The `CACHEPOLICY` value specifies whether the results of the transaction are cacheable. If they are, then the `TIMEOUTVALUE` specifies the number of milliseconds the results should be cached. In cases where the cache response could be used by looking up a cache index, the timeout value should be set to make sure the data is still in the cache when it is needed. The `INDEX_NAME` value, when set, specifies the name of the cache index to do the data lookup on.

Configuring the Metadata Tables for CustomerSearch

The `REQUEST_TXN_LAYOUT`, `RESPONSE_TXN_LAYOUT`, `RESPONSE_META_DATA`, `RESPONSE_INDEX` and `INDEX_META_DATA` database tables must be correctly configured to map the `DataPackets` received from the persister to the host transaction fields, and vice versa. The `INDEX_META_DATA` table contains both the index structure and the name of the cache that it is indexed by. Since `CacheIndexPersister` extends `TxnPersister`, the cache name used is `txnPersister`.

NOTE: In the interests of clarity some of the columns in the tables have been omitted from the representation of the tables below. Consult the `txnsampleddata.sql` file supplied with MCA Services for the complete metadata.

Format of TESTFIND0001

Transaction `TESTFIND0001` corresponds to the `Customer` entity bean's `findByPrimaryKey()` method.

Table 43. `REQUEST_TXN_LAYOUT` has the following format:

Table 43. `REQUEST_TXN_LAYOUT`

FIELDNAME	DP_FIELD	LENGTH	SEQUENCE	Sample value
-----------	----------	--------	----------	--------------

FIELDNAME	DP_FIELD	LENGTH	SEQUENCE	Sample value
T-CODE	TXN_CODE	12	1	TESTFIND0001
T-RESTART-INDEX	RESTART_INDEX	4	2	0000
C-OWNER-ID	OWNER_ID	10	3	1234560010

Table 44. `RESPONSE_META_DATA` has the following format:

Table 44. `RESPONSE_META_DATA`

DP_NAME	DP_INDEX	DP_FIELD	TXN_FIELDNAME
HEADER	1	RECORD_COUNT	H-RECORDS
HEADER	1	RESTART_FLAG	H-RESTART
CUSTOMER	2	OWNER_ID	C-OWNER-ID
CUSTOMER	2	FIRST_NAME	C-FIRST-NAME
CUSTOMER	2	LAST_NAME	C-LAST-NAME
CUSTOMER	2	TITLE	C-TITLE
ADDRESS	3	POST_CODE	A-POST-CODE
ADDRESS	3	ADDRESS_LINE1	A-LINE-1
ADDRESS	3	ADDRESS_LINE2	A-LINE-2
ADDRESS	3	ADDRESS_LINE3	A-LINE-3
ADDRESS	3	ADDRESS_LINE4	A-LINE-4
ADDRESS	3	COUNTRY	A-COUNTRY

The response is parsed into three `DataPackets`:

- The header `DataPacket` which contains the header information
- The `Customer DataPacket` which contains the data for the `Customer` entity
- The `Address DataPacket` which contains the data for the `Address` entity associated with the `Customer` entity

Format of TESTFIND0002

Transaction `TESTFIND0002` corresponds to the `Customer` entity bean's `findByLastName()` method.

Table 45. REQUEST_TXN_LAYOUT, has the following format:

Table 45. REQUEST_TXN_LAYOUT

FIELDNAME	DP_FIELD	LENGTH	SEQUENCE	Sample value
T-CODE	TXN_CODE	12	1	TESTFIND0002
T-RESTART-INDEX	RESTART_INDEX	4	2	0000
C-LAST-NAME	LAST_NAME	20	3	Walsh

Table 46. RESPONSE_META_DATA, has the following format:

Table 46. RESPONSE_META_DATA

TXN_FIELDNAME	DP_NAME	DP_FIELD	DP_INDEX
H-RECORDS	HEADER	RECORD_COUNT	1
H-RESTART	HEADER	RESTART_FLAG	1
C-OWNER-ID	CUSTOMER	OWNER_ID	2
C-FIRST-NAME	CUSTOMER	FIRST_NAME	2
C-LAST-NAME	CUSTOMER	LAST_NAME	2
C-TITLE	CUSTOMER	TITLE	2
A-POST-CODE	ADDRESS	POST_CODE	3
A-LINE-1	ADDRESS	ADDRESS_LINE1	3
A-LINE-2	ADDRESS	ADDRESS_LINE2	3
A-LINE-3	ADDRESS	ADDRESS_LINE3	3
A-LINE-4	ADDRESS	ADDRESS_LINE4	3
A-COUNTRY	ADDRESS	COUNTRY	3
C-OWNER-ID	CUSTOMER	OWNER_ID	4
C-FIRST-	CUSTOMER	FIRST_NAME	4

TXN_FIELDNAME	DP_NAME	DP_FIELD	DP_INDEX
NAME			
C-LAST-NAME	CUSTOMER	LAST_NAME	4
C-TITLE	CUSTOMER	TITLE	4
A-POST-CODE	ADDRESS	POST_CODE	5
A-LINE-1	ADDRESS	ADDRESS_LINE1	5
A-LINE-2	ADDRESS	ADDRESS_LINE2	5
A-LINE-3	ADDRESS	ADDRESS_LINE3	5
A-LINE-4	ADDRESS	ADDRESS_LINE4	5
A-COUNTRY	ADDRESS	COUNTRY	5
C-OWNER-ID	CUSTOMER	OWNER_ID	6
C-FIRST-NAME	CUSTOMER	FIRST_NAME	6
C-LAST-NAME	CUSTOMER	LAST_NAME	6
C-TITLE	CUSTOMER	TITLE	6
A-POST-CODE	ADDRESS	POST_CODE	7
A-LINE-1	ADDRESS	ADDRESS_LINE1	7
A-LINE-2	ADDRESS	ADDRESS_LINE2	7
A-LINE-3	ADDRESS	ADDRESS_LINE3	7
A-LINE-4	ADDRESS	ADDRESS_LINE4	7
A-COUNTRY	ADDRESS	COUNTRY	7
C-OWNER-ID	CUSTOMER	OWNER_ID	8
C-FIRST-	CUSTOMER	FIRST_NAME	8

TXN_FIELDNAME	DP_NAME	DP_FIELD	DP_INDEX
NAME			
C-LAST-NAME	CUSTOMER	LAST_NAME	8
C-TITLE	CUSTOMER	TITLE	8
A-POST-CODE	ADDRESS	POST_CODE	9
A-LINE-1	ADDRESS	ADDRESS_LINE1	9
A-LINE-2	ADDRESS	ADDRESS_LINE2	9
A-LINE-3	ADDRESS	ADDRESS_LINE3	9
A-LINE-4	ADDRESS	ADDRESS_LINE4	9
A-COUNTRY	ADDRESS	COUNTRY	9

As you can see, the response is quite long! **TESTFIND003** is an example of a transaction that has repeating groups. The response may contain the data for zero or more **Customer** and **Address** entities, furthermore the host will only return four results at a time, so the transaction must be fired against the host multiple times to get the complete result set.

The **H-RECORDS** field in the response indicates how many records were returned by the host.

The **H-RESTART** field indicates whether there are more records to be retrieved from the host. If this field has a value of '1' then there are more results to be retrieved, otherwise there are no more results.

Since the host returns four results at a time the metadata for the **Customer** and **Address** entities must be repeated four times, with each entity instance being given a different entity occurrence value.

Format of TESTAMND0001

Transaction **TESTAMND0001** corresponds to the **Customer** entity's **store()** or **amendAddress()** methods. Both **store()** and **amendAddress()** must use this transaction to amend **Customer** and/or **Address** data because the host only provides a single transaction for amending **Customer** and **Address** attributes.

Table 47. **REQUEST_TXN_LAYOUT**, has the following format:

Table 47. **REQUEST_TXN_LAYOUT**

FIELDNAME	SEQUENCE	DP_FIELD	LENGTH
T-CODE	1	TXN_CODE	12

FIELDNAME	SEQUENCE	DP_FIELD	LENGTH
C-OWNER-ID	2	OWNER_ID	10
C-FIRST-NAME	3	FIRST_NAME	20
C-LAST-NAME	4	LAST_NAME	20
C-TITLE	5	TITLE	5
A-POST-CODE	6	POST_CODE	15
A-LINE-1	7	ADDRESS_LINE1	20
A-LINE-2	8	ADDRESS_LINE2	20
A-LINE-3	9	ADDRESS_LINE3	20
A-LINE-4	10	ADDRESS_LINE4	20
A-COUNTRY	11	COUNTRY	20

Table 48. `RESPONSE_META_DATA`, has the following format:

Table 48. `RESPONSE_META_DATA`

TXN_FIELDNAME	DP_INDEX	DP_NAME	DP_FIELD
H-STATUS	1	CUSTOMER	STATUS

The request transaction contains the transaction code and all the attributes of the `Customer` and `Address` entities. The response transaction contains a single field indicating if the amend operation succeeded. If the operation succeeds the field will contain 'OK', otherwise the field will contain 'ERROR'.

Configuring the TXN_ROUTE Table for CustomerSearch

The `TXN_ROUTE` table must be correctly configured to map requests to the correct connector and to specify which data formatter class to use. Table 49 below illustrates the data used to configure the `TXN_ROUTE` table:

Table 49. `TXN_ROUTE`

TXN_CODE	TXN_TYPE	DESTINATION_ID	DATAFORMAT
TESTFIND0001	TEST	C001	<code>com.bankframe.examples.txnhandler.</code>

TXN_CODE	TXN_TYPE	DESTINATION_ID	DATAFORMAT
			<code>dataformat.testcustomer.TestCustomerDataFormat</code>
<code>TESTFIND0002</code>	<code>TEST</code>	<code>C001</code>	<code>com.bankframe.examples.txnhandler. dataformat.testcustomer.TestCustomerDataFormat</code>
<code>TESTFIND0004</code>	<code>TEST</code>	<code>C001</code>	<code>com.bankframe.examples.txnhandler. dataformat.testcustomer.TestCustomerDataFormat</code>
<code>TESTAMND0001</code>	<code>TEST</code>	<code>C001</code>	<code>com.bankframe.examples.txnhandler. dataformat.testcustomer.TestCustomerDataFormat</code>

In all cases the data formatter class used is:

```
com.bankframe.examples.txnhandler.dataformat.testcustomer.TestCustomerDataFormat
```

Similarly all transactions use the same destination: `C001`

Configuring the DESTINATION Table for CustomerSearch

Table 50 must be configured to specify which connector to use for communicating with the host system:

Table 50. `DESTINATION`

DESTINATION_ID	CONNECTOR_FACTORY_CLASSNAME	CONNECTOR_PROPERTIES
<code>C001</code>	<code>com.bankframe.examples.txnhandler. connector.testcustomer. TestCustomerConnectionFactory</code>	<code>offlineMode=disable</code>

Where the connector is defined in the `BankframeResource.properties` file.

Configuring the RESPONSE_INDEX Table for CustomerSearch

The `RESPONSE_INDEX` table must be configured as follows so that responses from the host system are indexed when cached:

Table 51. `RESPONSE_INDEX`

TXN_CODE	TXN_TYPE	INDEX_NAME
<code>TESTFIND0002</code>	<code>TEST</code>	<code>CUSTOMER_FIRST_NAME_INDEX</code>

The configuration shown above in the [RESPONSE-INDEX](#) table specifies that the response to [TESTFIND0002](#) ([findByLastName](#)) will be indexed. In this example the response [CUSTOMER DataPacket](#) will be indexed by the first name attribute.

Configuring the INDEX_META_DATA Table for CustomerSearch

The [INDEX_META_DATA](#) table must be configured as follows to specify the index structures:

Table 52. [INDEX_META_DATA](#)

INDEX_NAME	CACHE_NAME	DP_NAME	DP_FIELD
CUSTOMER_FIRST_NAME_INDEX	txnPersister	CUSTOMER	FIRST_NAME

The [CacheIndexer](#) class for managing the index is defined in the [BankframeResource.properties](#) file under the [cache.index.<INDEX_NAME>](#) key. If a specific [CacheIndexer](#) is not defined the default [CacheIndexer](#) is used. The default [CacheIndexer](#) gets its index structure from the [INDEX_META_DATA](#) table. This table contains the name of the cache to index as well as the [DataPacket](#) name and the fields to index by. The configuration shown above in the [INDEX_META_DATA](#) table will index the [FIRST_NAME](#) attribute in the [CUSTOMER DataPacket](#).

Configuring the Cobol Test Connector for CustomerSearch

The [Account](#) example uses the Cobol Test Connector:

```
com.bankframe.examples.txnhandler.connector.coboltest.*
```

The Cobol Test Connector generates Cobol binary data from a specified Cobol copybook file and returns the data to the Financial Process Integrator. This can be used to test the Financial Process Integrator metadata and entity bean's design for a simulated host system.

The Cobol test Connector key [transactionHandler.connector.CobolTestConnector.*](#) is configured in [BankframeResource.properties](#).

Table 53. CobolTestConnector Properties

Setting	Description
midfile	Specifies the path of the cobol copybook that defines the format of the data request to the host system.
modfile	Specifies the path of the cobol copybook that defines the format of the cobol data response from the host system.
cobol.numbtype	Specifies the format of the numbers in the created cobol data; COMP-3 , COMP , X , STD
cobol.texttype	Specifies the format of text created in the cobol data; ASCII , EBCDIC
midfile.debug	Specifies if debug information is displayed while host request is

Setting	Description
	being processed; <code>TRUE</code> , <code>FALSE</code>
<code>modfile.debug</code>	Specifies if debug information is displayed while the host response is being processed; <code>TRUE</code> , <code>FALSE</code>
<code>modfile.fillfield.<field name>=<value></code>	Specifies a specific value, <code><value></code> , for the field called <code><field name></code> in the host response, to simulate an error response.

Configuring the CustomerSearch Example

If your application server is installed in a folder other than the default location defined in the deployment guide and you wish to use the CustomerSearch example then the following changes must be made:

- Edit the `BankframeResource.properties` file and locate the `transactionHandler.test.customerData` setting.
- Change the value of this setting to point to the correct location of the `TestCustomerData.properties` (`TestCustomerData.properties` will be located in the same folder as `BankframeResource.properties`).
- Edit the `TestCustomerData.properties` file and locate the `this.absolutePath` setting.
- Change the value of this setting to point to the correct location of the `TestCustomerData.properties`.

The AccountSearch Example

This example illustrates how the Financial Process Integrator works using the Account Entity EJB and the AccountSearch Session EJB.

Name	EJB Type	Description
<code>Account</code>	Entity	Models the common attributes of an account
<code>AccountSearch</code>	Session	Searches for <code>Account</code> instances

These examples aim to show:

- How an entity bean interacts with the persister
- How the persister interacts with the Financial Process Integrator
- How to configure the Financial Process Integrator metadata
- How to configure the Financial Process Integrator routes and destinations
- How to configure the example Cobol Test Connector

The Account Entity EJB

The [Account](#) entity has the following attributes:

Attribute	Description
cardNumber	The customers card number
accountNumber	The account number
accountName	The account name

The AccountSearch Session EJB

The [AccountSearch](#) session bean must be able to find all the [Account](#) entities.

Interfacing the Account Entities with the Financial Process Integrator

The modelling of the [Account](#) entity bean is described below. This topic focuses on issues relevant to connecting the entity bean to the Financial Process Integrator.

`com.bankframe.examples.impl.account.AccountBMPBean`

This class is the Bean Managed Persistence (BMP) implementation of the [Account](#) entity bean.

This class must persist its attributes to/from the host system.

EBMPEntity Methods

As described previously all BMP entity beans must implement the `com.bankframe.ejb.bmp.EBMPEntity` interface. This is achieved in a similar manner to the `CustomerBMPBean` example described previously. The implementation of the methods defined in the `EBMPEntity` interface by `AddressBMPBean` is described below.

Configuring the PERSISTER_TXN_MAP Table for AccountSearch

The `PERSISTER_TXN_MAP` database table must be correctly configured to connect the BMP entity beans to the Financial Process Integrator. Table 54 below illustrates the data used to configure the `PERSISTER_TXN_MAP` table for the [Account](#) entity:

Table 54. `PERSISTER_TXN_MAP` Database

ENTITYNAME	METHODNAME	TXNCODE	TXNTYPE	CACHE POLICY	TIMEOUT VALUE
<code>eontec.bankframe.examples.bo.account</code>	<code>findAll</code>	<code>ACCOUNTFIND</code>	<code>TEST</code>	<code>none</code>	<code>0</code>

This table maps entity names and method names to transaction codes and transaction types. The [CACHEPOLICY](#) value specifies whether the results of the transaction are cacheable. If they are then the [TIMEOUTVALUE](#) specifies how many milliseconds the results should be cached for.

Configuring the Metadata Tables for AccountSearch

The [REQUEST_TXN_LAYOUT](#), [RESPONSE_TXN_LAYOUT](#) and [RESPONSE_META_DATA](#) database tables must be correctly configured to map the [DataPackets](#) received from the persister to the host transaction fields, and vice versa.

NOTE: In the interests of clarity some of the columns in the tables have been omitted from the tables below. Consult the [txnsampledatab.sql](#) file supplied with MCA Services for the complete metadata.

Format of ACCOUNTFIND

Transaction [ACCOUNTFIND](#) corresponds to the [Account](#) entity's [findAll\(\)](#) method. Table 55. [REQUEST_TXN_LAYOUT](#) has the following format:

Table 55. [REQUEST_TXN_LAYOUT](#)

FIELDNAME	DP_FIELD	LENGTH	SEQUENCE	Sample value
T-CODE	TXN_CODE	12	1	TESTFIND0001

Table 56. [RESPONSE_META_DATA](#), has the following format:

Table 56. [RESPONSE_META_DATA](#)

TXN_FIELDNAME	DP_NAME	DP_FIELD	DP_INDEX
CARD-NUMBER	ACCOUNT	CARD_NUMBER	1
ACCOUNT-NUMBER	ACCOUNT	ACCOUNT_NUMBER	1
ACCOUNT-NAME	ACCOUNT	ACCOUNT_NAME	1
CARD-NUMBER	ACCOUNT	CARD_NUMBER	2
ACCOUNT-NUMBER	ACCOUNT	ACCOUNT_NUMBER	2
ACCOUNT-NAME	ACCOUNT	ACCOUNT_NAME	2
CARD-NUMBER	ACCOUNT	CARD_NUMBER	3
ACCOUNT-NUMBER	ACCOUNT	ACCOUNT_NUMBER	3

TXN_FIELDNAME	DP_NAME	DP_FIELD	DP_INDEX
ACCOUNT-NAME	ACCOUNT	ACCOUNT_NAME	3
CARD-NUMBER	ACCOUNT	CARD_NUMBER	4
ACCOUNT-NUMBER	ACCOUNT	ACCOUNT_NUMBER	4
ACCOUNT-NAME	ACCOUNT	ACCOUNT_NAME	4
...

The response is parsed into ten [Account DataPackets](#) which contain the data for the [Account](#) entities.

Configuring the TXN_ROUTE Table for AccountSearch

The [TXN_ROUTE](#) table must be correctly configured to map requests to the correct connector and to specify which data formatter class to use. Table 57 below illustrates the data used to configure the [TXN_ROUTE](#) table:

Table 57. [TXN_ROUTE](#)

DESTINATION_ID	CONNECTOR_FACTORY_CLASSNAME	CONNECTOR_PROPERTIES
C002	<code>com.bankframe.examples.txnhandler.connector.coboltest.CobolTestConnectionFactory</code>	<code>offlineMode=disable</code>

In all cases the data formatter class used is:

```
com.bankframe.examples.txnhandler.dataformat.testaccount.TestAccountDataFormat
```

This data-format class is derived from

```
com.bankframe.ei.txnhandler.dataformat.basic.BasicDataFormat
```

Similarly all transactions use the same destination: [C002](#)

Configuring the DESTINATION Table for AccountSearch

Table 58. [DESTINATION](#), must be correctly configured to specify the correct connector to use for communicating with the host system:

Table 58. DESTINATION

DESTINATION_ID	CONNECTOR_FACTORY_CLASSNAME	CONNECTOR_PROPERTIES
C002	<code>com.bankframe.examples.txnhandler.connector.coboltest.CobolTestConnectionFactory</code>	<code>offlineMode=disable</code>

The settings for this connector are defined in `BankframeResource.properties`.

The `Account` example uses the following Cobol copybook, `modAccountTestCobol.txt`, to define the request to the host system from the Financial Process Integrator:

```
000400 01 MAIN-ACCOUNTFIND.

001400* INPUT DATA

001500 03 ACCOUNTFIND-DATA.

001600 05 T-CODE PIC X(5).
```

The Cobol Test Connector parses the Cobol data request generated by the Financial Process Integrator using this Cobol copybook. This parsing of Cobol data from the Financial Process Integrator is used to test the design of the request transaction fields in the Financial Process Integrator metadata. The `BankframeResource.properties` property `transactionHandler.connector.CobolTestConnector.cobol.debug` is set to `TRUE` to view the results of the parsing.

The `Account` example uses the following Cobol copybook, `midAccountTestCobol.txt`, to define the response from the host system to the Financial Process Integrator:

```
000400 01 MAIN-ACCOUNTFIND.

000410 010 ERROR-FLAG PIC X(5).

000420 010 ERROR-TYPE PIC X(20).

000430 010 FILLER PIC X(5).

000450* FOLLOWING IS A REPEATING FIELD, USED IN EACH OF

000500* THE FOLLOWING ENTITIES:

001300 010 CARD-NUMBER PIC 9(5).

001400* EACH OCCURANCE OF THIS GROUP MAPS TO AN INSTANCE OF AN ENTITY:

001500 05 ACCOUNT-INFO OCCURS 10 TIMES.

001700 010 ACCOUNT-NUMBER PIC 9(5).

001800 010 ACCOUNT-NAME PIC X(10).
```

```

001850*****
001900* APPENDING HOST-SYSTEM ERROR COBOL COPYBOOK HERE
002000* SO IT CAN BE USED BY TXN HANDLER SAMPLE META-DATA
002010* WHEN AN ERROR IS BEING SIMULATED:
002015*****
002020      05 HOST-SYSTEM-ERROR.
002030      010 ERROR-CODE          PIC 9(5).
002040      010 ERROR-MESSAGE      PIC X(30).

```

The Cobol Test Connector generates the Cobol binary data host system response that is expected by the Financial Process Integrator for the transaction being tested. This is used to test the design of the response transaction fields in the Financial Process Integrator metadata.

The Cobol Test Connector generates values for the transaction fields in the response by one of the following three methods in this order:

- The `BankframeResource.properties` key `transactionHandler.connector.CobolTestConnector.modfile.fillfield.<field name>=<value>` can be used to generate a specific value for transaction fields in the host system response.
- Field names in the mod Cobol copybook file that match field names in the mid Cobol copybook file result in the response transaction field taking the value of that request field when the transaction is being processed. The full group name is not used for comparing request and response field names, only the transaction field name, that is, if the sample `modAccountTestCobol.txt`, and hence response, above had a transaction field called T-CODE it would use the value of the T-CODE given in the request transaction, defined by `midAccountTestCobol.txt`.
- A unique sample text is generated for each field in the host response. Text field values are in the format `A1, A2 ... A<N>`. Numeric fields use digits.

Handling Complex Amend and Find Operations

In some cases, it may be necessary to invoke amend or find operations directly from a session bean, rather than via the `amend()` or `findByXXX()` methods of an entity bean, for example:

- If the data to be amended is not modelled as an entity bean
- If the data from many entities need to be merged, and these entities cannot be modelled using a master-dependent relationship

To facilitate these cases a class called: `com.bankframe.ei.txnhandler.broker.TxnHandlerBroker` is provided:

TxnHandlerBroker

The Financial Process Integrator Broker provides an `amend()` and `find()` interface into the Financial Process Integrator, that is not dependant on mapping entity beans to host transactions. To provide as flexible a framework as possible, interfaces are provided to allow behaviour to be customised at various stages of the broker's operation. Default implementations of these interfaces are provided with the MCA. This can be extended to provide specific behaviour for a host transaction request and the caching of data.

HostTransactionObject and HostTransactionObjectFactory

The `HostTransactionObject` is used to hold data and vector of primary keys to be used by the `TxnHandlerPersister` when performing either a find or amend operation. The `HostTransactionObjectFactory` is used to create `HostTransactionObjects` from values in a `HashMap`. The factory inspects the type of object in the map and determines how the `DataPacket` of data and `Vector` of primary keys will be created. The `getHostTransactionObject` method can be overridden to provide different behaviour for a specific EJB or method name.

Amend operations

There are two static amend methods in `TxnHandlerBroker`. Both take EJB name and method name as parameters. However, one also takes a `DataPacket` with amend data and `Vector` of `DataPackets` representing the primary keys for data to be stored, or removed from the cache used by the `TxnPersister`. The other amend method takes a `HashMap` of objects that the broker will pass to a `HostTransactionObjectFactory` to get the amend data and vector of primary keys to pass to the former amend method. When performing an amend the `TxnHandlerBroker` will also check the `transactionHandler.broker.removeFromCacheOperation.<ejb name>.<method name> boolean` property to pass to the persister. If none specified, the `transactionHandler.broker.removeFromCacheOperation.default` will be used. The persister will determine what behaviour will be implemented to remove or updated the persisters cache. The amend methods will return the `Vector` of `DataPackets` returned by the persister.

Find operations

Similar to amend, there are two static find methods in `TxnHandlerBroker`. Both take ejb name and method name as parameters. However, one also takes a `DataPacket` with finder data to be used by the `TxnPersister`. The other amend method takes a `HashMap` of objects that the broker will pass to a `HostTransactionObjectFactory` to get the amend data to pass to the former find method. The find methods will return the `Vector` of `DataPackets` returned by the persister.

Handling Create and Remove Operations

In the EJB model new data is created by calling the `create()` method of an entity bean's home interface, similarly data is deleted by calling the home's `remove()` method. It is assumed that these operations are carried out synchronously and immediately.

In many banking environments create and remove operations may not be performed immediately, instead they may be batched up to be performed only once per day. For example creation of new customer bank account's are usually performed as a batch operation carried out after the close of business.

Create and remove operations which are not carried out immediately should be implemented using a session bean which calls the `TxnHandlerBroker.amend()` method.

Create and remove operations which are carried out immediately should be implemented by defining the appropriate operations in the `PERSISTER_TXN_MAP` table, and the correct metadata in the `TXN_FIELD` table.

Immediate create operation example

The example below illustrates how to configure a create operation for the `Customer` entity (assuming the create is carried out immediately by the host).

Configuring the PERSISTER_TXN_MAP table

Table 59. `PERSISTER_TXN_MAP`, should have the following entry:

Table 59. `PERSISTER_TXN_MAP`

Entity Name	Method Name	Transaction Code	Transaction Type	Cache Policy	Time out value
eontec.bankframe. examples. bo.customer	create	TESTCREA0001	TEST	none	0

Configuring the TXN_FIELD table

Table 60. `TXN_FIELD`, should have the following data:

Table 60. `TXN_FIELD`

Field Name	Sequence	DataPacket Field Name	Length
T-CODE	1	TXN_CODE	12
C-OWNER-ID	2	OWNER_ID	10
C-FIRST-NAME	3	FIRST_NAME	20
C-LAST-NAME	4	LAST_NAME	20
C-TITLE	5	TITLE	5
A-POST-CODE	6	POST_CODE	15
A-LINE-1	7	ADDRESS_LINE1	20
A-LINE-2	8	ADDRESS_LINE2	20
A-LINE-3	9	ADDRESS_LINE3	20
A-LINE-4	10	ADDRESS_LINE4	20

Field Name	Sequence	DataPacket Field Name	Length
A-COUNTRY	11	COUNTRY	20
H-STATUS	1	STATUS	5

Sample Data Formatter Implementation

The data formatter class is responsible for interpreting the metadata and using it to transform the request data into the format that the host system understands, and conversely to transform the response data into a format the Financial Process Integrator can understand.

The [Customer](#) and [Address](#) examples above require a custom data formatter class which is implemented by:

```
com.bankframe.examples.txnhandler.dataformat.testcustomer.TestCustomerDataFormat
```

This class extends the [com.bankframe.ei.txnhandler.dataformat.basic.BasicDataFormat](#) class. The [BasicDataFormat](#) class provides a number of methods that can be overridden these are described below:

checkIfMoreToRequest()

This method is called by [BasicDataFormat](#) after the response from the host has been parsed into [DataPackets](#). Its purpose is to determine if the complete result set has been received from the host, if not then another request transaction must be sent to the host to get more results. Below is the code for the [TestCustomerDataFormat](#) implementation of this method:

```
protected boolean checkIfMoreToRequest(DataPacket txnRequest, Vector
responseData) throws ProcessingErrorException {

    DataPacket header = (DataPacket)responseData.elementAt(0);

    if ( header != null) {

        String restartIndexString    = txnRequest.getString("RESTART_INDEX");

        String recordCountString     = header.getString("RECORD_COUNT");

        String restartFlagString     = header.getString("RESTART_FLAG");

        if(recordCountString == null || restartFlagString == null ) {

            return false;

        }

        int recordCount = Integer.parseInt(recordCountString);

        int continueFlag = Integer.parseInt(restartFlagString);

        int restartIndex = 0;
```

```

    if ( restartIndexString != null ) {

        restartIndex = Integer.parseInt(restartIndexString);

        } if(continueFlag == 1) {

            txnRequest.put("RESTART_INDEX", Integer.toString(restartIndex +
recordCount));

            return true;

        }

    }

    return false;

}

```

This method carries out the following steps:

- Extracts the header `DataPacket` from the response `DataPackets`
- Extracts the restart index from the request `DataPacket`
- Extracts the record count value from the header `DataPacket`
- Extracts the restart flag from the header `DataPacket`
- If the restart flag is equal to '1' then modify the request `DataPacket` to request the next set of results and return `true`
- Otherwise return `false`

checkIfNoEntitiesFound()

This method is called by `BasicDataFormat` after the response from the host has been parsed into `DataPackets`. Its purpose is to determine if the response received from the host does not contain any entity data. Below is the code for the `TestCustomerDataFormat` implementation of this method:

```

protected boolean checkIfNoEntitiesFound(Vector responseData) throws
ProcessingErrorException {

    if(super.checkIfNoEntitiesFound(responseData)) {

        return true;

    }

    if(responseData.size() == 1) {

        DataPacket header = (DataPacket)responseData.elementAt(0);

        int recordCount = Integer.parseInt(header.getString("RECORD_COUNT"));

        if(recordCount == 0) {

```

```

        return true;
    }

    }

    return false;
}

```

This method carries out the following steps:

- Call the super-classes' `checkIfNoEntitiesFound()` method to check that the response data `Vector` is not empty or `null`.
- Check if the response data contains only a single `DataPacket`.
- If it does then assume the `DataPacket` is the header `DataPacket`, and check the record count value.
- If the record count is zero return `true` otherwise return `false`.

postProcessResponseData()

This method is called by `BasicDataFormat` after the response from the host has been parsed into `DataPackets`. Its purpose is to carry out any extra processing that may be necessary on the response `DataPackets`.

5 Enterprise Services

This chapter covers the required and optional enterprise services and frameworks that are provided with MCA. It includes the following topics:

- About Enterprise Services
- Security Provider Framework
- User Authentication
- Session Management
- Access Control
- Routing
- Remote Notification
- Internationalization
- Logging
- Audit
- Timing Points
- Mail
- Ping
- LDAP Connectivity
- Data Validation
- Peripherals Support
- Printing Framework
- Caching Framework
- Dynamic Configuration

About Enterprise Services

MCA Services includes a set of services and frameworks to provide ancillary functionality such as routing, access control, dynamic configuration and logging. These services are divided into required services, which are necessary for MCA to function correctly, and optional services, which are not required for MCA to function.

The following enterprise services are required for MCA to function correctly:

- Routing
- User Authentication

- Session Management
- Access Control
- Internationalization
- Dynamic Configuration

The following enterprise services are optional:

- Logging
- Audit
- Timing Points
- Mail
- Ping
- LDAP Connectivity
- Data Validation
- Peripherals Support
- Printing Framework
- Caching Framework

Security Provider Framework

MCA Services provides a customizable Security Provider Framework for ensuring that access to Financial Components is limited to authorized users. As part of the processing of a client request, the MCA request router dispatches the request to the specified Security Provider. The MCA Services Security Provider Framework consists of a [NullBankFrameSecurityProvider](#) and a [DefaultBankFrameSecurityProvider](#). The Security Provider framework also enables the implementation of custom security providers. The [NullBankFrameSecurityProvider](#) is used to turn off security. The [DefaultBankFrameSecurityProvider](#) encompasses the following processes:

- **User Authentication.** The client sends a logon request [DataPacket](#), which contains the user's authentication details. The [DataPacket REQUEST_ID](#) must map to the User Authentication EJB Session bean. The logon request is passed to the User Authentication Bean, which determines if the user's credentials are correct.
- **Session Management.** If a client's user credentials are determined to be correct then a user session is created and the session ID is returned to the client. The client adds this session ID to each subsequent [DataPacket](#) it sends to MCA Services. This requirement makes sure that only authenticated users gain access. Each time MCA Services receives a request from a client it checks to ensure that the session ID is valid.
- **Access Control.** Before passing a [DataPacket](#) from a client to a Financial Component for processing the access control bean checks to ensure that the client has access to the Financial Component. Each [DataPacket](#) from the client will contain a unique session ID. This session ID

corresponds to an individual user. The user's access rights corresponding to the Session ID will be checked to ensure the user has access to the requested Financial Component.

Security Provider Framework Classes and Package Structure

The Security Provider Framework is located in the `com.bankframe.services.security` package. It consists of a security provider interface named `BankFrameSecurityProvider` and comes complete with two security provider implementations: `DefaultBankFrameSecurityProvider` and the `NullBankFrameSecurityProvider`.

The Security Provider interface (which all providers must implement) contains the method:

```
public Vector dispatch(Vector request, Route route) throws ProcessingErrorException,
RemoteException.
```

This method Takes a `Vector` of `DataPackets` (which is the original client request) and a `Route` object which the request router has determined is the correct route to match the client request's `REQUEST_ID`. It must verify that the specified request is permitted to be processed.

If this method returns `null` then it is assumed that the request be permitted. However, if this method returns a `Vector` of `DataPackets` then these will be returned to the client and the request will be considered to be processed. If a request is not to be permitted then a `ProcessingErrorException` (or subclass) will be thrown.

Configuring the Security Provider

The Security Provider for a solution runtime is configured using the `security.provider` key in the `BankframeResource.properties` configuration file.

The key takes a fully qualified class name of the required Security Provider implementation.

It is imperative that the configured Security Provider implementation fully implements the `com.bankframe.services.security.BankFrameSecurityProvider` interface as described above.

For example, if a solution wished to switch off security (that is, switch off user authentication, session management and access control) and allow all client requests to attempt processing then the included `NullBankFrameSecurityProvider` would be used and configured as follows:

```
security.provider=com.bankframe.services.security.NullBankFrameSecurityProvider
```

There is an example configuration of the Security Provider included in the default `BankframeResource.properties` file – which ships with MCA Services.

It is worth noting that the individual Security Providers are likely to require implementation specific configuration. For an example of this refer to the included `DefaultBankFrameSecurityProvider` which uses the following keys: `security.sessionMgmtJndiName` and `security.accessControlJndiName`.

DefaultBankFrameSecurityProvider

The Default Security Provider brings together and uses the User Authentication, Session Management and Access Control services described in later chapters and exposes them using the MCA Security Provider Framework.

Package

```
com.bankframe.services.security.DefaultBankFrameSecurityProvider
```

Configuring the Default Security Provider

Configuring the Default Security Provider requires the setting of the following keys in the `BankframeResource.properties` configuration file:

```
security.provider
security.sessionMgmtJndiName
security.accessControlJndiName
```

The `security.provider` key should be set to `com.bankframe.services.security.DefaultBankFrameSecurityProvider`. Both the `security.sessionMgmtJndiName` and `security.accessControlJndiName` keys should be set to the JNDI name of the Session Management EJB and Access Control EJB respectively.

For example,

```
security.provider=com.bankframe.services.security.DefaultBankFrameSecurityProvide
r
security.sessionMgmtJndiName=eontec.bankframe.EJBSessionManagement
security.accessControlJndiName=eontec.bankframe.EJBAccessControl
```

NullBankFrameSecurityProvider

The Null Security Provider allows all client requests to be processed, and is a means of turning off security.

Package

```
com.bankframe.services.security.NullBankFrameSecurityProvider
```

Configuring the Null Security Provider

To configure the Null Security Provider set the `security.provider` in the `BankframeResource.properties` configuration file to the `com.bankframe.services.security.NullBankFrameSecurityProvider` implementation.

For example:

```
security.provider=com.bankframe.services.security.NullBankFrameSecurityProvider
```

Caution should be observed if making this change on a production solution as it will effectively disable security.

Implementing a Security Provider

A custom security provider allows one to customize the implementation of security. To write a security provider you need to write a class which implements the `com.bankframe.services.security.BankFrameSecurityProvider` interface. This interface prescribes the `dispatch()` method that will be called by the MCA `RequestRouter`. When implementing your own Security Provider then any necessary logic can be inserted into `dispatch()` to determine if a particular client request may be permitted. There are three valid types of returns from this method:

null – Whenever a call to dispatch returns **null** this will be interpreted by the `RequestRouter` as having passed security and to be ready for processing.

Vector of DataPackets – Return a **Vector** if the security provider has fully processed the request. This **Vector** will then be returned to the client by the MCA `RequestRouter`. This case arises if the client requests to logon and the security provider can fully process this request and return a response to the client.

Method throws `ProcessingErrorException` – This exception should be thrown if you do not wish to continue processing a user's request, for example, if the user has failed security checks.

The following is a brief overview of how a simple security provider can be implemented and the code behind the `NullBankFrameSecurityProvider`.

```
public class NullBankFrameSecurityProvider implements BankFrameSecurityProvider {

    public Vector dispatch(Vector datapacket, Route route) throws
ProcessingErrorException, RemoteException {

        return null;

    }

}
```

As can be seen from this example any request and route passed into the `dispatch()` method will result in a return of null, therefore all client requests will continue to be processed.

User Authentication

User Authentication is part of the MCA Services Security Provider Framework. The MCA User Authentication framework provides a set of standard authentication mechanisms and provides a framework for implementing custom authentication mechanisms. User authentication is required to facilitate the session management and access control mechanisms.

Framework for Custom Authentication Mechanisms

In many scenarios a custom authentication mechanism will be needed to capture the data required to authenticate a user, or to integrate with an existing authentication mechanism. MCA provides an interface that custom authentication mechanisms must comply with. Authentication mechanisms that implement this interface can be plugged into MCA.

Standard Authentication Mechanisms

MCA provides two standard authentication mechanisms:

- Authenticating users against a database table
- Authenticating users against an LDAP repository

The Siebel Retail Finance Logon Process

Before a client can access MCA it must log on. A client achieves this by carrying out the following steps:

- Send a request for any free services that are required in carrying out user authentication (a 'free service' is an MCA Service or a Siebel Financial Component that is not session managed). For example a call may be made to the [GenerateRandomNumbers](#) service in order to decide which digits from a PIN code to prompt the user for.
- Send a request to the user authentication mechanism with the necessary data to authenticate the user.
- If the request is successful then the user authentication mechanism will return a response to the client, otherwise an exception [DataPacket](#) will be returned to the client.
- If the request is successful then the first returned [DataPacket](#) will contain the session ID of the user session that was created for the user. The client should store this session ID so that it can pass it back to MCA with all subsequent requests. See Session Management for more detail on this.

The Siebel Retail Finance Logoff Process

When a user is finished using the client application, then the MCA Session should be terminated. A client achieves this by carrying out the following steps:

- Send a logoff request with the session ID for the user's current session to the user authentication mechanism.
- If the request is successful then a response will be sent back to the client confirming the logoff request succeeded, otherwise an exception [DataPacket](#) will be returned to the client.
- If the logoff request is successful then the user session will be deleted. Therefore the client must establish another session before it can again use MCA Services.

com.bankframe.services.authentication package

The `com.bankframe.services.authentication` package defines the interfaces that all MCA User Authentication Mechanisms must comply with.

Table 61. Authentication Classes

Method	Description
<code>AuthenticationBean</code>	Abstract EJB session bean class that defines the methods that all authentication mechanisms must implement.
<code>AuthenticationException</code>	Exception class thrown when user authentication fails.
<code>Authentication</code>	Remote Interface that the authentication EJBs must extend.
<code>AuthenticationUtils</code>	Utility class that provides methods for simplifying interaction with MCA User Authentication Mechanisms.

com.bankframe.services.authentication.AuthenticationBean

The basic functionality that all authentication methods must provide is defined in the `com.bankframe.services.authentication.AuthenticationBean` class. This class defines two abstract methods:

- `processLogon(DataPacket data)`
- `processLogoff(DataPacket data)`

`AuthenticationBean` extends the `com.bankframe.ejb.ESessionBean` class. It provides a standard implementation of the required `processDataPacket()` method, which checks if the incoming request is a logon or a logoff request and passes the request on to `processLogon()` or `processLogoff()` as appropriate. This means that all MCA User Authentication Mechanisms are implicitly standard MCA Services.

processLogon(DataPacket data)

```
public abstract Vector processLogon(DataPacket data) throws
ProcessingErrorException;
```

This method is responsible for retrieving the authentication information from the `DataPacket` passed in and verifying that the information is correct. If the information is not correct then it should throw an `AuthenticationException`. If the information is correct it should return a `Vector` of `DataPackets`. The first `DataPacket` in the `Vector` must have a field named `com.bankframe.services.authentication.Authentication.USER_ID`. This field must have a `String` value that is the unique user ID for the authenticated user. The returned `Vector` of `DataPackets` will be passed back to the client.

processLogoff(DataPacket data)

```
public abstract Vector processLogoff(DataPacket data) throws
ProcessingErrorException;
```

This method is called whenever a user attempts to logoff. It allows the custom authentication mechanism to be notified when the user logs off, and to perform any clean ups that need to be carried out. If an error occurs then this method should throw a `ProcessingErrorException`, for example if the user is already logged off. If the logoff attempt is successful then a `Vector` of `DataPackets` with the logoff response is returned. This `Vector` of `DataPackets` will be passed back to the client.

com.bankframe.services.authentication.AuthenticationException

This exception class extends the `com.bankframe.ejb.ProcessingErrorException` class. It should be thrown by user authentication mechanisms whenever user authentication fails. When this exception class is converted to a `DataPacket`, the `DataPacket` name will be `AUTHENTICATION EXCEPTION`.

com.bankframe.services.authentication.Authentication

This remote interface extends the `com.bankframe.ejb.EsessionRemote` interface. All MCA Authentication Mechanisms' remote interfaces must extend this interface. It defines the following two methods:

- `public Vector processLogon(DataPacket data) throws AuthenticationException;`
- `public Vector processLogoff(DataPacket data) throws ProcessingErrorException;`

com.bankframe.services.authentication.AuthenticationUtils

This utility class provides static methods to simplify interaction with MCA Authentication Mechanisms. These methods are typically used by client applications to create `DataPackets` that are correctly formatted for making user authentication requests.

Table 62. AuthenticationUtils Methods

Method	Description
<code>public static void makeLogonPacket(DataPacket dp)</code>	Add the data to a <code>DataPacket</code> that identifies it as a logon request.
<code>Public static void makeLogoffPacket(DataPacket dp, String sessionId)</code>	Add the data to a <code>DataPacket</code> that identifies it as a logoff request.
<code>Public static boolean checkIsLogonPacket(DataPacket dp)</code>	Checks if a <code>DataPacket</code> is a logon request.
<code>Public static boolean checkIsLogoffPacket(DataPacket dp)</code>	Checks if a <code>DataPacket</code> is a logoff request.
<code>Public static String getUserId(DataPacket dp)</code>	Extracts the unique user ID from a <code>DataPacket</code> .
<code>Public static String</code>	Puts a user ID field into a <code>DataPacket</code> .

Method	Description
<code>putUserId(DataPacket dp, String userId)</code>	

Implementing a Custom Authentication Mechanism

The main task in developing a custom authentication mechanism is implementing the `processLogon()` and `processLogoff()` methods. Apart from that the process is identical to developing any EJB session bean.

The example below implements an MCA Authentication Mechanism that interfaces with an imaginary third party authentication mechanism defined as follows:

```
public class ThirdPartyAuthenticationMechanism {

    public static void logon(String userId,String password) throws
    ThirdPartyException;

    public static void logoff(String userId) throws ThirdPartyException;

}
```

Creating the Bean Implementation

The Bean Implementation Class

```
import com.bankframe.bo.DataPacket;

import com.bankframe.ejb.ProcessingErrorException;

import com.bankframe.services.authentication.AuthenticationBean;
import com.bankframe.services.authentication.AuthenticationException;
import com.bankframe.services.authentication.AuthenticationUtils;

public class SampleAuthenticationBean extends AuthenticationBean {

    private final static int LOGON_ERROR=10026;

    private final static int LOGOFF_ERROR=10027;

    public Vector processLogon(DataPacket data) throws AuthenticationException {

        String userId = null;

        try {
```

```

        userId = data.getString(SampleAuthentication.USER_ID);

        String password = data.getString(SampleAuthentication.PASSWORD);

        ThirdPartyAuthenticationMechanism.logon(userId,password);

        return this.getLogonResponse(userId);
    } catch ( ThirdPartyException ex ) {

        String[] params = new String[1];

        params[0] = userId;

        throw new AuthenticationException(this.LOGON_ERROR,params);
    }
}

public Vector processLogoff(DataPacket data) throws ProcessingErrorException {

    String userId = null;

    try {

        userId = data.getString(SampleAuthentication.USER_ID);

        ThirdPartyAuthenticationMechanism.logoff(userId);

        return this.getLogoffResponse(userId);
    } catch ( ThirdPartyException ex ) {

        String[] params = new String[1];

        params[0] = userId;

        throw new ProcessingErrorException(LOGOFF_ERROR,params);
    }
}

private Vector getLogonResponse(String userId) {

    Vector v = new Vector();

    DataPacket response = new DataPacket("LOGON RESULT");

    response.put(AuthenticationUtils.USER_ID,userId);

    v.addElement(response);

    return v;
}

```

```

    }

    private Vector getLogoffResponse(String userId) {

        Vector v = new Vector();

        DataPacket response = new DataPacket("LOGOFF RESULT");

        response.put(AuthenticationUtils.USER_ID,userId);

        v.addElement(response);

        return v;

    }

}

```

The Bean Implementation Code explanation

The `SampleAuthenticationBean` class extends the `com.bankframe.services.authentication.AuthenticationBean` class. It provides implementations of the two abstract methods: `processLogon()` and `processLogoff()`.

SampleAuthenticationBean.processLogon()

This method carries out the following steps:

- Extract the user ID and password from the incoming `DataPacket`.
- Attempt to authenticate the user by invoking `ThirdPartyAuthenticationMechanism.logon()`.
- If the authentication is successful then produce a response to be sent back to the client by calling the `getLogonResponse()` method.
- If the authentication fails then a `ThirdPartyException` is raised. This is caught and a `AuthenticationException` is thrown.

SampleAuthenticationBean.processLogoff()

This method carries out the following steps:

- Extract the user ID from the incoming `DataPacket`.
- Attempt to logoff the user by invoking `ThirdPartyAuthenticationMechanism.logoff()`.
- If the logoff is successful then produce a response to be sent back to the client by calling the `getLogoffResponse()` method.
- If the logoff fails then a `ThirdPartyException` is raised. This is caught and a `AuthenticationException` is thrown.

SampleAuthenticationBean.getLogonResponse()

This method simply produces a response `DataPacket` to be sent back to the client confirming that the logon was successful.

SampleAuthenticationBean.getLogoffResponse()

This method simply produces a response `DataPacket` to be sent back to the client confirming that the logoff was successful.

Defining the Remote Interface

The Remote Interface for the `SampleAuthenticationBean` is defined as follows:

```
import com.bankframe.services.authentication.Authentication;

public interface SampleAuthentication extends Authentication {

    public final String USER_ID="userId";

    public final String PASSWORD="password";

}
```

This interface defines two constants; `USER_ID` and `PASSWORD`, that define the names of the fields in logon request `DataPackets` that the user ID and password, required by the third party authentication mechanism, are stored in.

Defining the Home Interface

The Home Interface for the `SampleAuthenticationBean` is defined as follows:

```
import java.rmi.RemoteException;

import javax.ejb.EJBHome;

import javax.ejb.CreateException;

public interface SampleAuthenticationHome extends EJBHome {

    public SampleAuthentication create() throws CreateException,RemoteException;

}
```

This interface simply defines the `create()` method used to create instances of the `SampleAuthenticationBean`.

Defining the Deployment Descriptor

The deployment descriptor format differs from one application server to another. Consult your application server documentation for details on how to create a deployment descriptor.

Building and Deploying the Bean

Build `SampleAuthenticationBean` the same as any other session bean using the tools appropriate for the application server you are targeting. Deploy the bean to the application server as normal. Finally register the bean with MCA as detailed below.

Registering Authentication Mechanisms with MCA Services

See the Administrating MCA Services documentation.

Implementing a Client Authentication Application

In order for client applications to be able to access MCA Services the client must be able to authenticate itself with MCA. The example below illustrates a simple Java application that authenticates itself with MCA using the `SampleAuthenticationBean` example above. The `SampleAuthenticationBean` is deployed to Route 30003.

The `SampleAuthenticationBean`

```
import java.util.Vector;

import com.bankframe.bo.DataPacket;

import com.bankframe.ei.comms.EHTTCommsManager;

import com.bankframe.services.sessionmgmt.SessionManagementUtils;

import com.bankframe.services.authentication.AuthenticationUtils;

public class SampleClient {

    public final static String AUTH_REQUEST_ID="30003";

    public static void main(String[] args) {

        try {

            String appserver = args[0];

            String userId = args[1];

            String password = args[2];

            DataPacket dp = new DataPacket("SAMPLE LOGON REQUEST");

            AuthenticationUtils.makeLogonPacket(dp);

            dp.put(SampleAuthentication.USER_ID,userId);
```

```

dp.put(SampleAuthentication.PASSWORD,password);

dp.put(DataPacket.REQUEST_ID,AUTH_REQUEST_ID);

EHTTPCommsManager commsMgr = new EHTTPCommsManager("sample",appserver);

Vector response = commsMgr.sendDataPacket(dp);

String sessionId =
SessionManagementUtils.getSessionId((DataPacket)response.elementAt(0));

if ( sessionId != null ) {

    System.out.println("user: " + userId + " was successfully authenticated");

    dp = new DataPacket("SAMPLE LOGOFF REQUEST");

    AuthenticationUtils.makeLogoffRequest(dp,sessionId);

    dp.put(DataPacket.REQUEST_ID,AUTH_REQUEST_ID);

    response = commsMgr.sendDataPacket(dp);

    userId = AuthenticationUtils.getUserId((DataPacket)response.elementAt(0));

    if ( userId != null ) {

        System.out.println("logged off successfully");

    } else {

        System.out.println("failed to logoff successfully");

    }

} else {

    System.out.println("user: " + userID + " was not successfully authenticated");

}

} catch ( Exception ex ) {

    System.out.println(ex.toString());

}

}

```

SampleAuthenticationBean Code explanation

The sample client carries out the following steps

Create the logon request

- The client creates a `DataPacket`, the name is unimportant, (in this case it is: 'SAMPLE LOGON REQUEST')
- Uses the `AuthenticationUtils.makeLogonRequest()` to turn the `DataPacket` into a logon request
- Adds the `userId` and password information required by `SampleAuthenticationBean` to the `DataPacket`
- Sets the `DataPacket REQUEST_ID` to 30003, so that the request is routed to the `SampleAuthenticationBean`

Send the DataPacket to MCA

- The client creates an `EHTTPCommsManager` instance, passing it the URL of the application server where MCA is running.
- The client calls the comms manager's `sendDataPacket()` method to send the logon request to MCA.
- MCA receives the request and routes it to `SampleAuthenticationBean`, which in turn authenticates the request.
- MCA passes back the response from `SampleAuthenticationBean` to the client. This is the return value from the `sendDataPacket()` method call.

Check if the logon was successful

- The client uses the `SessionManagementUtils.getSessionId()` method to see if the returned response contains a session ID.
- If it does then the logon attempt was successful, because MCA will only generate a `sessionId` when the client has been successfully authenticated.
- If it does not then the user authentication must have failed.

Logoff

- If the logon attempt was successful, then the client attempts to logoff
- The client creates another `DataPacket`.
- It calls `AuthenticationUtils.makeLogoffPacket()` to convert the `DataPacket` to a logoff request.
- It sets the `REQUEST_ID` of the `DataPacket` to 30003 so the logoff request is routed to the `SampleAuthenticationBean`.
- If the logoff attempt is successful then, the returned `DataPacket` will contain an `AuthenticationUtils.USER_ID` field.
- If the attempt is not successful then, the `DataPacket` will not contain an `AuthenticationsUtils.USER_ID` field.

LDAP Authentication

LDAP based Authentication is implemented in the `com.bankframe.services.authentication.ldap` package. It can authenticate any user defined in an LDAP repository.

Configuring LDAP Authentication

- Deploy the `ldapauthentication.jar` EJB on the application server.
- Register the ldap authentication bean with MCA. The JNDI for the ldap authentication bean is: `eontec.bankframe.LDAPAuthentication`.

LDAP Authentication uses the ldap context named: `bankframeusers` to connect to the LDAP server. The configuration settings for the `bankframeusers` ldap context must be specified in `BankframeResource.properties` as follows:

The following settings are required, if they are not defined then LDAP Authentication will not be able to function:

`bankframeusers.ldap.baseDn` – Specifies the location in the LDAP server hierarchy within which to search for users, for example, `ou=Users,o=SomeOrganization`.
`bankframeusers.ldap.defaultSearchFilter`. Specifies the search filter to use to find a specific user, for example, `cn={0}`

All other `LDAPServerContext` settings can optionally be specified for the `bankframeusers` context. If they are not specified then default values will be inherited from the `ldap.default.*` settings defined elsewhere in `BankframeResource.properties`.

RDBMS Authentication

User authentication within a typical RDBMS is implemented in the `com.bankframe.services.authentication.ejb.user` package. It uses one session bean, `EJBUserAuthenticationBean`, and one entity bean, `EJBUserBean`.

Component Overview

EJBUserBean

`EJBUserBean` is a container-managed entity bean that houses information about `Users`. It maps to the `EJBUSERS` table in the database. This table has the following attributes:

Table 63. EJBUSERS Attributes

Attribute	Data Type	Description
USERID	VARCHAR2(80)	NOT NULL
PASSWORD	VARCHAR2(80)	
USERNAME	VARCHAR2(80)	

The Primary Key Field here is the `USERID`.

The `EJBUserBean` provides the following functionality.

- `getUserId()`
- `getName()`
- `validatePassword()`
- `toDataPacket()`

EJBUserAuthenticationBean

The `EJBUserAuthenticationBean` is a session bean used to validate users against passwords and to process user logon and logoff requests. This session bean is a subclass and implementation of the abstract `com.bankframe.services.authentication.AuthenticationBean`. It provides the following functionality:

Table 64. AuthenticationBean Methods

Method	Description
<code>processLogon()</code>	This takes a <code>DataPacket</code> with <code>userId</code> and <code>password</code> and returns a <code>Vector</code> of logon responses.
<code>processLogoff()</code>	Takes a <code>DataPacket</code> containing a <code>sessionId</code> and returns a <code>Vector</code> of logoff responses.

EJBUser table and Access Control to EJBs. The `EJBUSER` Table is used elsewhere within MCA to perform access control on specific EJBs. This is discussed further in the “MCA Access Control” document.

Configuring RDBMS User Authentication

- Deploy `userauthentication.jar` and `ejbuser.jar` EJBs on the server.
- Register the beans with MCA. The JNDI for the EJB authentication bean is `eontec.bankframe.EJBUserAuthentication`.

The JNDI name for the EJB user entity bean is `eontec.bankframe.EJBUser`.

Encrypting Sensitive Data

Message Digest Overview

A Message Digest is a digital fingerprint of a block of data. A number of algorithms have been designed to compute message digests – two of the most widely used are SHA, the secure hash algorithm and MD5.

MCA Message Digest service

MCA Services provides a Message Digest service enabling customers to ensure that sensitive information, for example, customer passwords, are stored/transmitted in a non-clear text format. The

hashing service is implemented in the `com.bankframe.services.security.MessageDigestUtils` class.

MCA Message Digest Configuration

The Message Digest service is configured in the `BankframeResource.properties` file. A name/value pair entry is configured to indicate which Message Digest algorithm is to be used. The entry in `BankframeResource.properties` is as follows:

```
# Defines the message digest algorithm to use

# Possible values are defined by the JCA

# Typical values are: MD5 | SHA-1

messageDigest.algorithm=SHA-1
```

Calling the `MessageDigest.digest(clearText)` service will return a String in non-clear text format. This non-clear text string will be based on the MessageDigest algorithm configured in `BankframeResource.properties`.

Session Management

Session Management is part of the MCA Security Provider Framework. The purpose of session management is to track which users are logged on. MCA provides both a framework for implementing session management and a standard implementation of session management. This allows custom solutions to be implemented which are integrated with MCA. MCA Session Management is independent of, and does not rely on, other session management systems such as HTTP sessions.

Components of MCA Services Session Management

Client	Can be a Java applet, application, servlet or JSP.
User	Authentication Mechanism.
<code>BankframeServlet</code> or <code>BankframePage</code>	Channel Managers
<code>RequestRouter</code>	Validates and routes requests to business processes.
<code>SessionManagment</code> implementation	Manages user sessions.
<code>BankFrameSessionServlet</code>	Provides administration facilities for session management.

Session Management Use Cases

There are four use cases for MCA session management:

Free Services	Services that can be used without a user being logged on.
Logging On	Authenticating the user.

Normal Use	Normal use of Financial Components
Logging Off	Ending an MCA Services session.

Free Services

Free Services are Financial Components which can be accessed without requiring a user to be logged on. Typically these services are required in the process of establishing the user session. For example the [GenerateRandomNumbers](#) service is normally a free service because it is required to generate the random selection of PIN digits that a user logging on should enter.

Logging On

Logging on is part of the user authentication process and is covered in more detail in the User Authentication document. MCA Services requires that all user authentication mechanisms provide a user ID that uniquely identifies the user. This user ID is used to generate the session ID that uniquely identifies each user session. When a user is successfully authenticated and a user ID is passed to the session management system, a new session is created for the user.

Normal Use

Once a session has been established the client can access the Siebel Retail Finance Financial Components. Each time the client sends a request to MCA Services it must include the session ID in the request. If the client does not include the session ID then MCA will refuse to process the request. When MCA receives the request it validates the session ID, for example, to make sure that the user session has not timed out through inactivity. If the session is determined to be valid the request is passed on to the access control mechanism (which will determine if the user has access rights to the requested business service). If the session is not valid then an exception will be returned to the client.

Logging Off

When a user wishes to log off they must inform MCA. When MCA receives a log off request, it informs the user authentication mechanism that the user is logging off, and deletes the user's session.

com.bankframe.services.sessionmgmt

MCA Session Management is implemented in the [com.bankframe.services.sessionmgmt](#) package. This package defines the functionality that all session management implementations must support. The package contains the following classes/interfaces

BankFrameSession	An interface that declares the methods all user sessions must expose.
SessionManagementBean	Abstract base class that all session management beans must extend.
SessionManagement	Remote Interface that declares the methods that session management beans must expose.

<code>SessionManagementUtils</code>	Utility class that facilitates the use of session management functionality.
<code>InvalidSessionException</code>	Exception thrown when an attempt is made to use an invalid session ID.
<code>Client</code>	Test application for testing user authentication and session management functionality

BankFrameSession

This interface defines the methods that all user sessions must have. It is up to the specific implementation to provide an implementation of this interface.

SessionManagementBean

This abstract base class defines the functionality that all session management implementations must provide. The class defines the following abstract methods:

<code>createSession()</code>	Create a new user session.
<code>deleteSession()</code>	Delete an existing user session.
<code>retrieveSession()</code>	Retrieve a user session instance, using the specified session ID.
<code>getNumValidSessions()</code>	Get the number of valid user sessions.
<code>getSessions()</code>	Retrieve a vector of all valid user sessions.
<code>removeInvalidSessions()</code>	Remove all invalid (expired) user sessions.
<code>removeAllSessions()</code>	Remove all users sessions, effectively logging off all users.

SessionManagement

This remote interface defines the functionality exposed by all session management implementations.

SessionManagementUtils

This class is a Utility class that facilitates the use of session management

InvalidSessionException

This exception is thrown whenever an attempt is made to use an invalid session ID A session ID is invalid if:

- The session it corresponds to has been deleted because the user has logged off
- The session it corresponds to has timed out through user inactivity

- MCA has not created a session for the specified ID.

com.bankframe.services.sessionmgmt.Client

This class is a test application used to test session management functionality

Implementing a session management aware client application

Before a client application can access MCA services it must establish a user session. This requires the client to authenticate itself with MCA. When the client application is finished it should inform MCA by logging off. See [Implementing a Client Authentication Application](#) for a detailed example of how to logon, access Siebel Retail Finance Services and logoff.

Implementing a custom session management implementation

In most cases one of the standard MCA implementations of session management should be sufficient, however in some cases it may be necessary to provide a custom implementation; for example if the session management system must integrate with some third party product.

All custom implementations must extend the [com.bankframe.services.sessionmgmt.SessionManagementBean](#) class. As described this class defines a number of abstract methods that must be implemented by the custom implementation.

The custom implementation must also provide an implementation of the [com.bankframe.services.sessionmgmt.BankFrameSession](#) interface.

Consult the JavaDocs reference for a full explanation of what behavior the above methods must implement.

Configuring and Administering Session Management

Deploying a Session Management Implementation

The session management implementation must be deployed on the application server, the same as any other service.

Secondly the session management implementation must be registered with MCA by assigning the implementation a Siebel Retail Finance Route. Assigning services to routes is covered in the MCA Deployment and Administration documentation.

Finally MCA must be told which EJB the session management implementation is deployed on. Setting the [security.sessionMgmtJndiName](#) property in [BankframeResource.properties](#) does this, for example, [security.sessionMgmtJndiName=eontec.bankframe.EJBSessionManagement](#)

Administering MCA Sessions

MCA sessions are administered using the [BankFrameSessionServlet](#). Check that this servlet has been deployed on your application server (The servlet is implemented in the

`com.bankframe.ei.servlet.BankFrameSessionServlet`). The `BankFrameSessionServlet` allows you to carry out the following operations:

- List all current sessions
- Remove expired sessions
- Remove all sessions
- Delete a specific session

List all current sessions

This option presents a list of all users currently logged on to MCA Services

Remove expired sessions

This option removes all sessions that have timed out due to user inactivity

Remove all sessions

This option logs off all users from MCA by deleting their sessions

Delete a specific session

This logs off a specific user by deleting their session

Standard Session Management Implementations

MCA Services provides two standard implementations of session management:

- A container managed Entity bean implementation that stores user sessions in an RDBMS
- A bean managed Entity bean implementation that stores user sessions in an LDAP repository

The first implementation generally gives better performance because user sessions need to have their time-stamp updated every time the user accesses an MCA service and LDAP servers are typically optimized for reads, not updates. This causes the LDAP implementation to perform slower than the RDBMS implementation.

The LDAP implementation may be useful for customers who want to keep all user related information in an LDAP repository.

RDBMS implementation

The RDBMS implementation is contained in the `ejbsessionmgmt.jar` JAR file. The RDBMS implementation has the following JNDI name: `eontec.bankframe.EJBSessionManagement`

The RDBMS implementation requires a database table called `SESSIONMGMT` to be created. The script to create this table is supplied with MCA Services.

LDAP Implementation

The ldap implementation is contained in the `ldapsessionmgmt.jar` JAR file. The ldap implementation has the following JNDI name: `eontec.bankframe.LDAPSessionManagement`

The LDAP implementation requires that a new object class is defined in the LDAP server's schema. The script to define this object class is supplied with MCA

Access Control

Access Control is part of the MCA Security Provider Framework. MCA Access Control provides secure access to MCA Financial Components. It controls which users can access which Financial Components.

Actors

The following actors exist in the MCA Access Control Model:

Users	Individual MCA Users.
User Groups	Arbitrary groupings of Users. A User Group contains one or more members. A User can belong to zero or more Groups.
Financial Component	A service available to Users.

MCA Access Control limits access to Financial Components to only those Users and/or Groups that have been granted access to the Financial Component.

Before a Siebel Retail Finance user can access Siebel Financial Components, they must authenticate themselves. This process is covered in the MCA User Authentication documentation.

When a user is successfully authenticated, a Siebel Retail Finance Session is created for that user. This session is uniquely identified by a session ID. Every time the user wishes to access a Siebel Retail Finance Financial Component they must provide a session ID. Before being granted access to the Financial Component the session ID is checked to ensure it is valid. After the session ID has been validated the access control rights for the corresponding user are checked to see if the user has access to the requested Financial Component. The user must have been granted access rights to the Financial Component, or alternatively be a member of a group with access to the Financial Component, before s/he can access the Financial Component. If the user does not have access an error will be reported.

Dependencies

MCA Access Control is dependent on the MCA User Authentication service to uniquely identify MCA Users.

MCA Access Control is dependent on the MCA Session Management service to ensure users are currently logged on.

Implementations

MCA provides two standard implementations of access control:

- An LDAP based Access Control Mechanism that leverages the access control mechanisms inherent in LDAP servers
- A CMP EJB based mechanism that uses several database tables to implement access control

Customisation

MCA provides an architecture for custom access control mechanisms to be implemented.

com.bankframe.services.accesscontrol

The MCA Access Control mechanism is implemented in the `com.bankframe.services.accesscontrol` package. This package provides a framework for implementing access control mechanisms. The package contains the following classes/interfaces:

<code>AccessControlBean</code>	Abstract base class that all access control mechanisms must extend.
<code>AccessControl</code>	Remote Interface that defines what functionality access control mechanisms expose.
<code>AccessControlException</code>	Exception thrown when an attempt is made to access a prohibited resource.

com.bankframe.services.accesscontrol.AccessControlBean

This base class defines the functionality that all access control mechanisms should implement. The class extends `com.bankframe.ejb.ESessionBean`. This means that access control mechanisms are standard Siebel Retail Finance Services. `AccessControlBean` provides a standard implementation of the required `processDataPacket()` method. `AccessControlBean` defines the following abstract method that must be defined by implementations:

```
public abstract boolean validateUserRequest(String userId,String requestId)
throws AccessControlException ;
```

This method takes a `userId` and a `requestId` as parameters and returns true if the user is allowed to access the Financial Component identified by `requestId`. If the user is not allowed access to the Financial Component then an `AccessControlException` will be thrown. An `AccessControlException` should also be thrown if the specified user or Financial Component cannot be located.

com.bankframe.services.accesscontrol.AccessControl

This remote interface defines the functionality exposed by access control mechanisms. The interface extends the `com.bankframe.ejb.ESessionRemote` interface. It defines the following method:

```
public boolean validateUserRequest(String userId,String requestId) throws
AccessControlException, RemoteException ;
```

This method can be invoked to check if a user has access to the Financial Component identified by `requested`.

com.bankframe.services.accesscontrol.AccessControlException

This exception is thrown when a user attempts to access a prohibited service.

Implementing a custom access control mechanism

Implementing a custom access control mechanism is very similar to implementing any other MCA Service; the only difference is that the `validateUserRequest()` method must be implemented. The following sample access control implementation integrates with a third party product that determines access rights. Assume the third party product has the following interface:

```
Public class ThirdPartyAccessControl {

    Public static Boolean checkAccess(String user,String resource) throws
    ThirdPartyException ;

}
```

Sample Bean Implementation

```
import com.bankframe.services.authentication.ldap.LDAPAuthentication;

import com.bankframe.services.accesscontrol.AccessControlBean;

import com.bankframe.services.accesscontrol.AccessControlException;

public class SampleAccessControlBean extends AccessControlBean {

    public boolean validateUserRequest(String userId,String requestId) throws
    AccessControlException {

        try {

            ThirdPartyAccessControl.checkAccess(userId,requestId);

            return true;

        } catch ( ThirdPartyException ex ) {

            String[] errparams = new String[2];

            errparams[0] = userId;

            errparams[1] = requestId;

            return new AccessControlException(10030,errparams);

        }

    }

}
```

```
}
```

The Bean Implementation Code Explained

The bean implementation needs to implement a single method: `validateUserRequest()`. In this example the implementation of `validateUserRequest()` delegates the task of verifying access rights to the `ThirdPartyAccessControl.checkAccess()` method. This method call is wrapped in a try-catch block which catches any `ThirdPartyExceptions`. If the user does have access to the resource (`requestId`) then the method will return true, otherwise a `ThirdPartyException` is thrown. This exception is caught and an `AccessControlException` is thrown instead.

Remote Interface

The remote interface for this bean just extends the `com.bankframe.services.accesscontrol.AccessControl` remote interface. It does not add an extra members or fields:

```
import com.bankframe.services.accesscontrol.AccessControl;

public interface SampleAccessControl extends AccessControl {

}
```

Home Interface

The home interface defines the `create()` method used to create bean instances:

```
import java.rmi.RemoteException;

import javax.ejb.EJBHome;

import javax.ejb.CreateException;

public interface SampleAccessControlHome extends EJBHome {

    SampleAccessControl create() throws CreateException, RemoteException;

}
```

LDAP Access Control Mechanism

The LDAP based Access Control Mechanism is implemented in the: `com.bankframe.services.accesscontrol.ldap package`. This implementation leverages the access control facilities inherent in LDAP servers such as IBM SecureWay Directory.

Configuring LDAP Access Control

- Deploy the `ldapaccesscontrol.jar` EJB on the application server

- Register the ldap access control bean with MCA Routing. The JNDI for the ldap access control bean is: `eontec.bankframe.LDAPAccessControl`.
- LDAP Authentication uses two ldap contexts (`bankframeusers` and `bankframeroutes`) to connect to the LDAP server. The `bankframeusers` context is used for validating users, and the `bankframeroutes` context is used for validating Financial Components.

The configuration settings for the `bankframeusers` ldap context must be specified in `BankframeResource.properties` as follows:

The following settings are required, if they are not defined then LDAP access control will not be able to function:

`bankframeusers.ldap.baseDn`. Specifies the location in the LDAP server hierarchy within which to search for users, for example, `ou=Users,o=SomeOrganization`
`bankframeusers.ldap.defaultSearchFilter`. Specifies the search filter to use to find a specific user, for example, `cn={0}`.

All other `LDAPServerContext` settings can optionally be specified for the `bankframeusers` context. If they are not specified then default values will be inherited from the `ldap.default.*` settings defined elsewhere in `BankframeResource.properties`.

The configuration settings for the `bankframeroutes` ldap context must be specified in `BankframeResource.properties` as follows:

The following settings are required, if they are not defined then LDAP access control will not be able to function:

`bankframeroutes.ldap.baseDn`. specifies the base distinguished name where MCA route information is stored.

`bankframeroutes.ldap.rdnAttribute`. specifies the name of the attribute used to form the relative distinguished name of each object.

All other `LDAPServerContext` settings can optionally be specified for the `bankframeroutes` context. If they are not specified then default values will be inherited from the `ldap.default.*` settings defined elsewhere in `BankframeResource.properties`.

Configuring Access Rights

Since the LDAP access control mechanism leverages the access control facilities in the LDAP server, the process for configuring Siebel Retail Finance Access Rights is identical to the process used to configure access rights to any other kind of resource in the LDAP server. You will need to consult your LDAP server documentation for details of how to configure access control, since each LDAP server product has differing implementations of access control.

Sample Implementation

This sample implementation illustrates how to configure access control rights using IBM SecureWay Directory. The following settings are required for the `bankframeusers` and `bankframeroutes` ldap contexts:

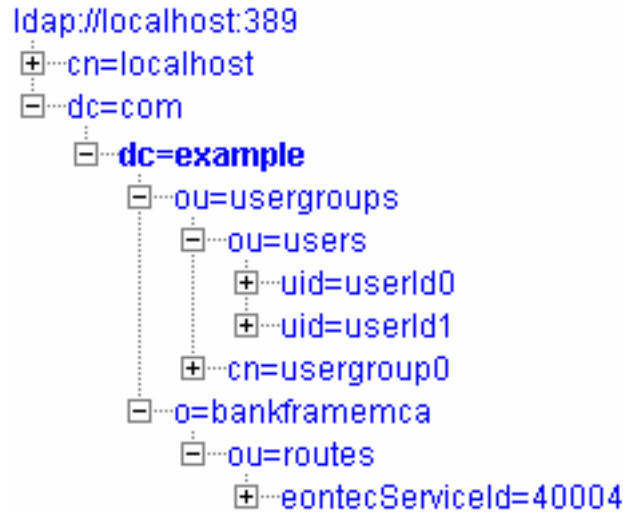
```
bankframeusers.ldap.baseDn=ou=users,ou=usergroups,dc=example,dc=com
```

```
bankframeusers.ldap.defaultSearchFilter=uid={0}
```

```
bankframeroutes.ldap.basedDn=ou=routes,o=bankframemca,dc=example,dc=com
```

```
bankframeroute.ldap.rdnAttribute=eontecServiceId
```

The example assumes the following tree structure in the LDAP server:



`Userld0` and `Userld1` are both members of the `usergroup0`

In this example access is granted to the Siebel Retail Finance Financial Component assigned to route 40004.

- 1 Launch the IBM Secureway Directory Management Tool
- 2 Log in using the administrator account
- 3 Select Browse Tree from the menu on the left
- 4 Expand the tree until you have selected the `eontecServiceId=40004,ou=routes,o=bankframemca,dc=example,dc=com` node.
- 5 Press the ACL button on the toolbar above the ldap tree window.
- 6 In the edit box indicated by the red arrow type:
`cn=usergroup0,ou=usergroups,dc=example,dc=com`
- 7 Select group from the drop down list and press the Add button
- 8 A new ACL entry will appear for `usergroup0`. Tick all the boxes under the Granted rights heading for this ACL entry
- 9 Press the change button.

The members of `usergroup0` have now been granted access to the Siebel Retail Finance Financial Component assigned to route 40004

EJB Access Control Implementation

MCA supports access control for EJBs within a conventional relational database system. A user can therefore be configured to only have access to certain Financial Components within this framework.

Configuring access rights

Model overview

There are conceptually two entities within this ejb access control system, users and groups. It behaves as follows:

- A group can be named and assigned various permissions.
- A user can be assigned to one or more groups. That user in turn inherits all the permissions assigned to his/her group(s).
- A user can be assigned specific permissions but does not have to be a member of a group.

This model has several advantages:

- Users can be grouped according to organizational status.
- Although a user is part of a group, a user can have permissions that extend beyond those of their predefined group.
- A user can use Financial Components independently of a group should the need arise.

Table overview

The system uses the following five database tables.

- EJBUSERS
- EJBUSER_PERMISSIONS
- EJBGROUPS
- EJBGROUP_MEMBERS
- EJBGROUP_PERMISSIONS

EJBUSERS. This table, discussed in the MCA Services User Authentication document, is a representation of all registered MCA Users. It has the following fields:

USERID	VARCHAR2 (80)	NOT NULL
PASSWORD	VARCHAR2 (80)	
USERNAME	VARCHAR2 (80)	

The Primary Key field here is the **USERID**. This field should be denoted preferably by a non-numeric code, which is similar to the real name of the user. For example, the **userId** of "Joe Bloggs" should resemble something like "jbloggs".

EJBUSER_PERMISSSIONS. This table will have one entry for each permission a user is assigned. This table will only have an entry if either of these conditions is satisfied:

- The user is not a member of a group and wants specific permissions.
- The user wants to be a member of a group but also wants extra permissions beyond the current scope of his/her assigned group.

It contains the following fields:

USERID	VARCHAR2(20)	NOT NULL
REQUESTID	VARCHAR2(15)	NOT NULL

The primary key field here is composed of both the `userId` and `requestId` to uniquely identify a `userId/requestId` pairing.

The `userId` in this table is a foreign key of `userId` in the `EJBUSERS` table. This means that for a user to have an entry in this table, they must have a corresponding entry in the `EJBUSERS` table. Similarly, a user cannot be removed from the `EJBUSERS` table if they are being referenced by an entry in this table.

EJBGROUPS. This table is a representation of the various user groups within MCA. It contains the following fields.

GROUPID	VARCHAR2(20)	NOT NULL
GROUPNAME	VARCHAR2(20)	

The primary key field here is the `groupId`.

EJBGROUP_MEMBERS. This table assigns users to groups. It contains the following fields. Note that a user can be a member of more than one group.

USERID	VARCHAR2(20)	NOT NULL
GROUPID	VARCHAR2(20)	NOT NULL

The primary key field here is a combination of the `userId` and `groupId`. This uniquely identifies a `userId/groupId` pairing.

`UserId` here is a foreign key of `userId` in the `EJBUSERS` table. A user therefore cannot be removed from the `EJBUSERS` table if a record in this table references them. Likewise, a user cannot be added to this table if they do not have a record to reference in the `EJBUSERS` table.

EJBGROUP_PERMISSSIONS. This table is a list of the permissions assigned to each group. This table will have one entry for each permission a group is assigned. It is conceptually equivalent to the `EJBUSER_PERMISSSIONS` table. It contains the following fields,

GROUPID	VARCHAR2(20)	NOT NULL
REQUESTID	VARCHAR2(20)	NOT NULL

The primary key field here is a combination of the `GROUPID` and `REQUESTID`. It uniquely identifies a `groupId/requestId` pairing.

`GroupId` here is a foreign key of `groupId` in the `EJBGROUPS` table. This constraint means that a record in the `EJBGroups` table cannot be deleted if referenced by an entry in this table. Also, a record cannot be entered in this table if there is not a corresponding entry for it to reference in the `EJBGROUPS` table.

EJB Overview

The access control system is implemented via one session bean, (`EJBAccessControlBean`) and five entity beans. They are:

<code>EJBUserBean</code>	An instance of this bean represents one record in the <code>EJBUsers</code> table.
<code>EJBGroupBean</code>	An instance of this bean represents one record in the <code>EJBGROUPS</code> table.
<code>EJBGroupMemberBean</code>	An instance of this bean represents one record in the <code>EJBGROUP_MEMBERS</code> table.
<code>EJBGroupPermissionBean</code>	An instance of this bean represents a record in the <code>EJBGROUP_PERMISSIONS</code> table.
<code>EJBUserPermissionBean</code>	An instance of this bean represents one row in the <code>EJBUSER_PERMISSIONS</code> table.

Session Bean Overview

The only session bean involved here is the `EJBAccessControlBean`. This session bean represents an implementation and subclass of the abstract `AccessControlBean`, a bean that declares common functionality to be implemented by all Access Control Mechanisms.

An instance of this bean exposes a single public method to a client.

<code>validateUserRequest()</code>	Validates a user against a <code>requestId</code> or permission. Returns true if user has access to the specified <code>REQUEST_ID</code> . Otherwise throws an <code>AccessControlException</code> .
------------------------------------	---

User and Group Administration Session Beans

UserAdministrationBean

This session bean represents an implementation and subclass of the abstract `ESessionBean`, it is the class responsible for the creation and removal of users and their permissions for MCA.

`com.bankframe.services.accesscontrol.administration.user`

The MCA User Administration mechanism is implemented in the `com.bankframe.services.accesscontrol.administration.user` package. This package provides a framework for implementing User Administration mechanisms. The package contains the following classes/interfaces:

<code>UserAdministrationBean</code>	The User Administration bean implementation.
<code>UserAdministration</code>	Remote Interface to the User Administration Bean
<code>UserAdministrationHome</code>	User Administration home interface.
<code>Client</code>	Application to test User Administration bean functionality

The JNDI name of the `UserAdministrationBean` is `eontec.bankframe.UserAdministration`

The `UserAdministrationBean`'s Methods

An instance of this bean exposes the following public methods to a client.

<code>getAllUsers()</code>	Returns an <code>Enumeration</code> of <code>User</code> objects for all users registered with MCA Services.
<code>getUser(String userId)</code>	Finds a user by <code>userId</code> and returns an instance of that user.
<code>getUserPermissions(String userId)</code>	Takes a <code>userId</code> and returns a <code>Vector</code> of that users permissions.
<code>deleteUser(String userId)</code>	Finds a user by <code>userId</code> and deletes that user. Returns <code>void</code> .
<code>createUser(String userId, String userName, String password)</code>	Creates a new user. Returns <code>void</code> .
<code>deleteUserPermission(String userId, String permission)</code>	Takes a <code>userId</code> and a <code>permission</code> and removes the permission from the user. Returns <code>void</code> .
<code>addUserPermission(String userId, String permission)</code>	Takes a <code>userId</code> and a <code>permission</code> and assigns the permission to the user. Returns <code>void</code> .
<code>addUserToGroup(String userId, String group)</code>	Takes a <code>userId</code> and a <code>group</code> and adds the user to the group. Returns <code>void</code> .
<code>deleteUserFromGroup(String userId, String group)</code>	Takes a <code>userId</code> and a <code>group</code> and removes the user from the group. Returns <code>void</code> .
<code>unassignedUserPermissions(String userId)</code>	Takes a <code>userId</code> and returns a <code>Vector</code> of the permissions the user doesn't have.

`processDataPacket()`

In order to invoke the methods of the `UserAdministrationBean` the client uses the `processDataPacket()` method.

getAllUsers(). To invoke this method using the `processDataPacket()` method a `DataPacket` with the following structure is sent by the client.

DATA PACKET NAME	GET_ALL_USERS
REQUEST_ID	MC054
OWNER	Usually Eontec LTD

The `processDataPacket()` method returns a `Vector` of one or more `DataPackets` containing a `DataPacket` for each user with the following structure.

DATA PACKET NAME	USERS_DETAILS
USER_ID	The <code>userId</code> of the user
USER_NAME	The full name of the user
REQUEST_ID	Default <code>REQUEST_ID</code> always 00000
OWNER	Usually Eontec LTD

getUser(String userId). To invoke this method using the `processDataPacket()` method a `DataPacket` with the following structure is sent by the client.

DATA PACKET NAME	GET_USER
REQUEST_ID	MC054
USER_ID	The <code>userId</code> of the user
OWNER	Usually Eontec LTD

The `processDataPacket()` method returns a `Vector` containing a `DataPacket` with the user details in it. This `DataPacket` has the same structure as one of the `DataPackets` returned by `getAllUsers()`.

getUserPermissions(String userId). To invoke this method using the `processDataPacket()` method a `DataPacket` with the following structure is sent by the client.

DATA PACKET NAME	GET_USER_PERMISSIONS
REQUEST_ID	MC054
USER_ID	The <code>userId</code> of the user
OWNER	Usually Eontec LTD

The `processDataPacket()` method returns a vector of one or more `DataPackets` containing a `DataPacket` for each permission with the following structure.

DATA PACKET NAME	ROUTE
REQUEST_ID	The <code>requestID</code> of the permission

JNDI_NAME	JNDI name of the permission
IS_SESSION_MANAGED	yes or no
DESCRIPTION	Description of the permission
OWNER	Usually Eontec LTD

deleteUser(String userId). To invoke this method using the `processDataPacket()` method a `DataPacket` with the following structure is sent by the client.

DATA_PACKET_NAME	DELETE_USER
REQUEST_ID	MC054
USER_ID	The <code>userId</code> of the user
OWNER	Usually Eontec LTD

The `processDataPacket()` method returns a `Vector` containing a `DataPacket` with a `DATA_PACKET_NAME` of `DELETE_USER` if successful or `USER_ADMINISTRATION_EXCEPTION` if unsuccessful.

createUser(String userId, String userName, String password). To invoke this method using the `processDataPacket()` method a `DataPacket` with the following structure is sent by the client.

DATA_PACKET_NAME	CREATE_USER
REQUEST_ID	MC054
USER_ID	The <code>userId</code> of the user
USER_NAME	The full name of the user
PASSWORD	A password for the user
OWNER	Usually Eontec LTD

The `processDataPacket()` method returns a `Vector` containing a `DataPacket` with a `DATA_PACKET_NAME` of `CREATE_USER` if successful or `USER_ADMINISTRATION_EXCEPTION` if unsuccessful.

deleteUserPermission(String userId, String permission). To invoke this method using the `processDataPacket()` method a `DataPacket` with the following structure is sent by the client:

DATA_PACKET_NAME	DELETE_USER_PERMISSION
REQUEST_ID	MC054
USER_ID	The <code>userId</code> of the user
PERMISSION	The <code>requestId</code> of the permission to be removed.
OWNER	Usually Eontec LTD

The `processDataPacket()` method returns a `Vector` containing a `DataPacket` with a `DATA_PACKET_NAME` of `DELETE_USER_PERMISSION` if successful or `USER_ADMINISTRATION_EXCEPTION` if unsuccessful.

addUserPermission(String userId, String permission). To invoke this method using the `processDataPacket()` method a `DataPacket` with the following structure is sent by the client:

DATA PACKET NAME	ADD_USER_PERMISSION
REQUEST_ID	MC054
USER_ID	The <code>userId</code> of the user
PERMISSION	The <code>requestId</code> of the permission to be assigned.
OWNER	Usually Eontec LTD

The `processDataPacket()` method returns a `Vector` containing a `DataPacket` with a `DATA PACKET NAME` of `ADD_USER_PERMISSION` if successful or `USER_ADMINISTRATION_EXCEPTION` if unsuccessful.

addUserToGroup(String userId, String group). To invoke this method using the `processDataPacket()` method a `DataPacket` with the following structure is sent by the client:

DATA PACKET NAME	ADD_USER_TO_GROUP
REQUEST_ID	MC054
USER_ID	The <code>userId</code> of the user
GROUP	The <code>groupId</code> of the group to add user to.
OWNER	Usually Eontec LTD

The `processDataPacket()` method returns a `Vector` containing a `DataPacket` with a `DATA PACKET NAME` of `DELETE_USER_PERMISSION` if successful or `USER_ADMINISTRATION_EXCEPTION` if unsuccessful.

deleteUserFromGroup(String userId, String permission). To invoke this method using the `processDataPacket()` method a `DataPacket` with the following structure is sent by the client.

DATA PACKET NAME	DELETE_USER_FROM_GROUP
REQUEST_ID	MC054
USER_ID	The <code>userId</code> of the user
GROUP	The <code>groupId</code> of the group to remove user from.
OWNER	Usually Eontec LTD

The `processDataPacket()` method returns a `Vector` containing a `DataPacket` with a `DATA PACKET NAME` of `ADD_USER_PERMISSION` if successful or `USER_ADMINISTRATION_EXCEPTION` if unsuccessful.

unassignedUserPermissions(String userId). To invoke this method using the `processDataPacket()` method a `DataPacket` with the following structure is sent by the client.

DATA PACKET NAME	UNASSIGNED_USER_PERMISSIONS
REQUEST_ID	MC054
USER_ID	The <code>userId</code> of the user

OWNER	Usually Eontec LTD
-------	------------------------------------

The `processDataPacket()` method returns a [Vector](#) of one or more [DataPackets](#) containing a [DataPacket](#) for each permission with the following structure:

DATA PACKET NAME	ROUTE
REQUEST_ID	The requestID of the permission
JNDI_NAME	JNDI name of the permission
IS_SESSION_MANAGED	yes or no
DESCRIPTION	Description of the permission
OWNER	Usually Eontec LTD

USER_ADMINISTRATION_EXCEPTION. When an exception is thrown by the above methods a [Vector](#) is returned containing a [DataPacket](#) with the following structure.

DATA PACKET NAME	USER_ADMINISTRATION_EXCEPTION
REQUEST_ID	Default REQUEST_ID always 00000
Message	A description of the problem which caused the exception to be thrown
OWNER	Usually Eontec LTD

GroupAdministrationBean

This session bean represents an implementation and subclass of the abstract [ESessionBean](#), it is the class responsible for the creation and removal of groups, their permissions and members.

com.bankframe.services.accesscontrol.adminstration.group

The MCA Group Administration mechanism is implemented in the [com.bankframe.services.accesscontrol.adminstration.group](#) package. This package provides a framework for implementing Group Administration mechanisms. The package contains the following classes/interfaces:

GroupAdministrationBean	The Group Administration bean implementation.
GroupAdministration	Remote Interface to the Group Administration Bean
GroupAdministrationHome	Group Administration home interface.
Client	Application to test Group Administration bean functionality

The JNDI name of the [GroupAdministrationBean](#) is [eontec.bankframe.GroupAdministration](#)

The GroupAdministrationBean's Methods

An instance of this bean exposes the following public methods to a client:

<code>getAllGroups()</code>	Returns an Enumeration of Group objects for all groups registered with MCA.
<code>getGroup(String groupId)</code>	Finds a group by groupId and returns an instance of that group.
<code>getGroupPermissions(String groupId)</code>	Takes a groupId and returns a Vector of that groups permissions.
<code>deleteGroup(String groupId)</code>	Finds a group by groupId and deletes that group. Returns void .
<code>createGroup(String groupId, String groupName)</code>	Creates a new group. Returns void .
<code>deleteGroupPermission(String groupId, String permission)</code>	Takes a groupId and a permission and removes the permission from the group. Returns void .
<code>addGroupPermission(String groupId, String permission)</code>	Takes a groupId and a permission and assigns the permission to the group. Returns void .
<code>getUnassignedGroups(String userId)</code>	Takes a userId and returns a Vector of the groups the user is not assigned to.
<code>getGroupMembers(String groupId)</code>	Takes a groupId and returns a Vector of the users assigned to it.
<code>getUserGroups(String userId)</code>	Takes a userId and returns a Vector of the groups the user is assigned to.
<code>unassignedGroupPermissions(String groupId)</code>	Takes a groupId and returns a Vector of the permissions the group doesn't have.

processDataPacket()

In order to invoke the methods of the [GroupAdministrationBean](#) the client uses the `processDataPacket()` method.

getAllGroups(). To invoke this method using the `processDataPacket()` method a [DataPacket](#) with the following structure is sent by the client:

DATA_PACKET_NAME	GET_ALL_GROUPS
REQUEST_ID	MC053
OWNER	Usually Eontec LTD

The `processDataPacket()` method returns a `Vector` of one or more `DataPackets` containing a `DataPacket` for each group with the following structure.

DATA PACKET NAME	GROUPS_DETAILS
GROUP_ID	The <code>groupId</code> of the group
GROUP_NAME	The name of the group
REQUEST_ID	Default <code>REQUEST_ID</code> always 00000
OWNER	Usually Eontec LTD

getGroup(String groupId). To invoke this method using the `processDataPacket()` method a `DataPacket` with the following structure is sent by the client:

DATA PACKET NAME	GET_GROUP
REQUEST_ID	MC053
GROUP_ID	The <code>groupId</code> of the group
OWNER	Usually Eontec LTD

The `processDataPacket()` method returns a `Vector` containing a `DataPacket` with the group details in it. This `DataPacket` has the same structure as one of the `DataPackets` returned by `getAllGroups()`.

getGroupPermissions(String groupId). To invoke this method using the `processDataPacket()` method a `DataPacket` with the following structure is sent by the client:

DATA PACKET NAME	GET_GROUP_PERMISSIONS
REQUEST_ID	MC053
GROUP_ID	The <code>groupId</code> of the group
OWNER	Usually Eontec LTD

The `processDataPacket()` method returns a `Vector` of one or more `DataPackets` containing a `DataPacket` for each permission with the following structure:

DATA PACKET NAME	ROUTE
REQUEST_ID	The <code>requestID</code> of the permission
JNDI_NAME	JNDI name of the permission
IS_SESSION_MANAGED	yes or no
DESCRIPTION	Description of the permission
OWNER	Usually Eontec LTD

deleteGroup(String groupId). To invoke this method using the `processDataPacket()` method a `DataPacket` with the following structure is sent by the client:

DATA PACKET NAME	DELETE_GROUP
REQUEST_ID	MC053
GROUP_ID	The <code>groupId</code> of the group
OWNER	Usually <code>Eontec LTD</code>

The `processDataPacket()` method returns a `Vector` containing a `DataPacket` with a `DATA PACKET NAME` of `DELETE_GROUP` if successful or `GROUP_ADMINISTRATION_EXCEPTION` if unsuccessful.

createGroup(String groupId, String groupName). To invoke this method using the `processDataPacket()` method a `DataPacket` with the following structure is sent by the client:

DATA PACKET NAME	CREATE_GROUP
REQUEST_ID	MC053
GROUP_ID	The <code>groupId</code> of the group
GROUP_NAME	The name of the group
PASSWORD	A password for the group
OWNER	Usually <code>Eontec LTD</code>

The `processDataPacket()` method returns a `Vector` containing a `DataPacket` with a `DATA PACKET NAME` of `CREATE_GROUP` if successful or `GROUP_ADMINISTRATION_EXCEPTION` if unsuccessful.

deleteGroupPermission(String groupId, String permission). To invoke this method using the `processDataPacket()` method a `DataPacket` with the following structure is sent by the client:

DATA PACKET NAME	DELETE_GROUP_PERMISSION
REQUEST_ID	MC053
GROUP_ID	The <code>groupId</code> of the group
PERMISSION	The <code>requestId</code> of the permission to be removed.
OWNER	Usually <code>Eontec LTD</code>

The `processDataPacket()` method returns a `Vector` containing a `DataPacket` with a `DATA PACKET NAME` of `DELETE_GROUP_PERMISSION` if successful or `GROUP_ADMINISTRATION_EXCEPTION` if unsuccessful.

addGroupPermission(String groupId, String permission). To invoke this method using the `processDataPacket()` method a `DataPacket` with the following structure is sent by the client:

DATA PACKET NAME	ADD_GROUP_PERMISSION
REQUEST_ID	MC053

GROUP_ID	The <code>groupId</code> of the group
PERMISSION	The <code>requestId</code> of the permission to be assigned.
OWNER	Usually <code>Eontec LTD</code>

The `processDataPacket()` method returns a `Vector` containing a `DataPacket` with a `DATA PACKET NAME` of `ADD_GROUP_PERMISSION` if successful or `GROUP_ADMINISTRATION_EXCEPTION` if unsuccessful.

getUnassignedGroups(String userId). To invoke this method using the `processDataPacket()` method a `DataPacket` with the following structure is sent by the client:

DATA PACKET NAME	GET_UNASSIGNED_GROUPS
REQUEST_ID	MC053
USER_ID	The <code>userId</code> of the user
OWNER	Usually <code>Eontec LTD</code>

The `processDataPacket()` method returns a `Vector` containing a `DataPacket` for each group with the following structure:

DATA PACKET NAME	GROUPS_DETAILS
GROUP_ID	The <code>groupId</code> of the group
GROUP_NAME	The name of the group
REQUEST_ID	Default <code>REQUEST_ID</code> always <code>00000</code>
OWNER	Usually <code>Eontec LTD</code>

getGroupMembers(String groupId). To invoke this method using the `processDataPacket()` method a `DataPacket` with the following structure is sent by the client:

DATA PACKET NAME	GET_GROUP_MEMBERS
REQUEST_ID	MC053
GROUP_ID	The <code>groupId</code> of the Group
OWNER	Usually <code>Eontec LTD</code>

The `processDataPacket()` method returns a `Vector` of one or more `DataPackets` containing a `DataPacket` for each user with the following structure:

DATA PACKET NAME	USERS_DETAILS
USER_ID	The <code>userId</code> of the user
USER_NAME	The full name of the user
REQUEST_ID	Default <code>REQUEST_ID</code> always <code>00000</code>

OWNER	Usually Eontec LTD
-------	------------------------------------

getUserGroups(String userId). To invoke this method using the [processDataPacket\(\)](#) method a [DataPacket](#) with the following structure is sent by the client:

DATA PACKET NAME	GET_USER_GROUPS
REQUEST_ID	MC053
USER_ID	The userId of the user
OWNER	Usually Eontec LTD

The [processDataPacket\(\)](#) method returns a [Vector](#) of one or more [DataPackets](#) containing a [DataPacket](#) for each group with the following structure:

DATA PACKET NAME	GROUPS_DETAILS
GROUP_ID	The groupId of the group
GROUP_NAME	The name of the group
REQUEST_ID	Default REQUEST_ID always 00000
OWNER	Usually Eontec LTD

unassignedGroupPermissions(String groupId). To invoke this method using the [processDataPacket\(\)](#) method a [DataPacket](#) with the following structure is sent by the client:

DATA PACKET NAME	UNASSIGNED_GROUP_PERMISSIONS
REQUEST_ID	MC053
GROUP_ID	The groupId of the group
OWNER	Usually Eontec LTD

The [processDataPacket\(\)](#) method returns a [Vector](#) of one or more [DataPackets](#) containing a [DataPacket](#) for each permission with the following structure:

DATA PACKET NAME	ROUTE
REQUEST_ID	The requestID of the permission
JNDI_NAME	JNDI name of the permission
IS_SESSION_MANAGED	Yes or no
DESCRIPTION	Description of the permission
OWNER	Usually Eontec LTD

GROUP_ADMINISTRATION_EXCEPTION. When an exception is thrown by the above methods a `Vector` is returned containing a `DataPacket` with the following structure.

DATA PACKET NAME	GROUP_ADMINISTRATION_EXCEPTION
REQUEST_ID	Default REQUEST_ID always 00000
Message	A description of the problem which caused the exception to be thrown
OWNER	Usually Eontec LTD

Routing

MCA Services Routing provides a flexible means for multiple clients communicating over multiple delivery channels to interact with Siebel Retail Finance Financial Components. The Routing Service takes care of delivering requests from clients to the correct Financial Components and returning the response data from those Financial Components to the client.

How MCA Services Routing Works

Rather than hard-code the name of the Financial Component into a client it is preferable to identify the Financial Component using a unique identifier called a `REQUEST_ID` and couple this to the Financial Component's name at runtime. This allows a Financial Component's implementation to be replaced with a different implementation without affecting the client. This coupling is what the Routing Service provides.

The Routing Service is implemented using an EJB session bean that contains the business logic for the Routing Service and an EJB entity bean that is used to store the routing data. The EJB Session bean is called the `RequestRouter` bean. The EJB Entity Bean is called the `Route` bean.

Each channel manager invokes the `RequestRouter` to route client requests to the correct Financial Component. The `RequestRouter` looks for the `REQUEST_ID` in each `DataPacket` sent from the client. It uses this five digit identifier to find a particular service. The `RequestRouter` maintains a mapping from `REQUEST_ID`s to Financial Components. Each time a request is received the `RequestRouter` looks up this mapping and translates the `REQUEST_ID` into the Financial Component name. Once the `RequestRouter` has discovered the Financial Components name, it creates an instance of the Financial Component and passes the client request on to the Financial Component. When the Financial Components has dealt with the request the `RequestRouter` returns the response data to the channel manager, which in turn passes the response back to the client.

This design is dependent on all the Financial Components conforming to the same interface, that is, implementing the method `processDataPacket()`. This method is defined as abstract in the class `com.bankframe.ejb.EsessionBean` class. All Siebel Retail Finance Financial Components extend this class and provide an implementation of this method.

Note that clients never interact directly with the `RequestRouter` service; they always interact with the service via the client connectivity framework.

In addition to performing routing of requests, the [RequestRouter](#) bean also uses the User Authentication, Session Management, and Access Control Services to ensure that clients only access the Financial Components they have been granted access to.

RequestRouter and Transactions

Accessing more than one database within the course of a single J2EE container managed transaction requires the application server and the JDBC driver to support the Java Transaction API (JTA). Many application servers and JDBC drivers do not provide full support for JTA specification.

The [RequestRouter](#) EJB accesses the [BANKFRM](#) database, via the [EJBRoute](#) EJB, to determine the appropriate Financial Component to invoke. In turn the Financial Component will usually access some other application specific database. If the [RequestRouter](#) EJB did use a transaction when either the application server or JDBC does not support JTA then the application server will produce a runtime exception when the Financial Component attempts to access the second database.

To work around this issue by default the [RequestRouter](#) EJB is configured not to use a transaction, thus only the Financial Component will access a database within the context of a transaction.

This workaround has one caveat which is that the Audit Provider and Security Provider which are invoked by the [RequestRouter](#) EJB cannot participate in the same transaction as the one used by the Financial Component, therefore it is impossible for the Audit Provider or the Security Provider to cause the rollback of the Financial Component transaction.

If the application server and JDBC driver being used do fully support the JTA specification then this issue can be remedied by updating the [RequestRouter](#) EJB deployment descriptor to use a container managed transaction, consult your application server vendor's documentation for information on how to do this.

If the application server and JDBC driver do not fully support JTA then the only workaround is to change all Financial Components to use the [BANKFRM](#) database, and to update the [RequestRouter](#) EJB deployment descriptor to use a container managed transaction.

The com.bankframe.services.requestrouter package

The business logic for the Routing Service is implemented in the [com.bankframe.services.requestrouter](#) package. This package consists of the following classes/interfaces:

RequestRouterBean	The session bean that implements MCA's routing logic.
RequestRouter	The remote Interface that declares the functionality RequestRouterBean exposes.
RequestRouterHome	The home interface used to create RequestRouterBean instances.
RequestRouterException	Exception thrown when an error occurs during the routing process.
RequestRouterUtils	Utility class to simplify channel manager's interactions with the RequestRouterBean .

RequestRouterBean

This class provides the implementation of MCA's Routing Service. Every time the [RequestRouterBean](#) receives a [DataPacket](#) it carries out the following operations:

- Check the [DataPacket](#) has a non-zero [REQUEST_ID](#).
- Look up the [Route](#) identified by the [REQUEST_ID](#).
- Check the [DataPacket](#) has a valid session ID.
- If the session ID is not present check to see if the [DataPacket](#) is a logon or logoff request; if so send the request to the User Authentication and Session Management Services.
- Otherwise use the Session Management service.
- Create an instance of the named Financial Component named in the [Route](#) and pass the [DataPacket](#) to the Financial Component, by invoking the EJB's [processDataPacket\(\)](#) method.
- Pass back the returned response data from the Financial Component.

RequestRouter

This remote interface defines the methods that [RequestRouterBean](#) exposes. [RequestRouterBean](#) is a standard MCA Enterprise Service and exposes only the standard [processDataPacket\(\)](#) method.

RequestRouterHome

This home interface has a single [create\(\)](#) method used to create instances of the [RequestRouterBean](#)

RequestRouterException

Exception is thrown when an error occurs during the routing process.

RequestRouterUtils

This utility class contains a single static method:

```
Vector processDataPacket(DataPacket data) throws RequestRouterException;
```

This method creates an instance of the [RequestRouterBean](#) and passes it the specified [DataPacket](#).

Channel Managers that need to pass [DataPackets](#) to the [RequestRouterBean](#) should use the above method to do so.

The com.bankframe.services.route package

This package contains the implementations of two entity beans that are used to persist the mapping of [REQUEST_ID](#)s to JNDI names. The two beans are [EJBRouteBean](#) and [LDAPRouteBean](#). [EJBRouteBean](#) persists data to an RDBMS, [LDAPRouteBean](#) persists data to an LDAP server. Apart from the datastore that the beans persist to, they are identical. This is reflected in the fact that both beans share the same home and remote interfaces and primary key class.

The `com.bankframe.services.route` package contains the following classes/interfaces:

<code>EJBRouteBean</code>	Container managed bean implementation.
<code>LDAPRouteBean</code>	Bean managed bean implementation that persists to LDAP server.
<code>Route</code>	Remote Interface that declares the methods of the <code>Route</code> Entity bean.
<code>RouteHome</code>	Home Interface used to create instances of the <code>Route</code> Entity bean.
<code>RoutePK</code>	Primary key class used to uniquely identify <code>Route</code> Entity bean instances.

EJBRouteBean

This is the standard container managed implementation of the `Route` bean

LDAPRouteBean

This is the ldap based implementation of the `Route` bean. It uses the `bankframeroutes` ldap context specified in the `BankframeResource.properties` configuration file

Route

This remote interface defines the attributes that the `Route` bean has. These are:

<code>REQUEST_ID</code>	The <code>REQUEST_ID</code> this Financial Component is mapped to.
<code>JNDI_NAME</code>	The JNDI name of the Financial Component.
<code>DESCRIPTION</code>	Brief description of the Financial Component.
<code>SESSION_MANAGED</code>	Boolean value that indicates if the Financial Component requires a user session to be established before it can be accessed.

RouteHome

This home interface declares the methods that can be used to create `Route` instances; these are:

<code>Create()</code>	Create a new <code>Route</code> instance.
<code>FindByPrimaryKey()</code>	Retrieve a specific instance.
<code>FindAll()</code>	Retrieve an enumeration of all instances.

RoutePK

This class uniquely identifies `Route` bean instances. The `Route` bean's primary key attribute is the `REQUEST_ID`.

Route Administration Session Bean

This session bean represents an implementation and subclass of the abstract [ESessionBean](#), it is the class responsible for the creation and removal of Routes.

com.bankframe.services.route.adminstration

The MCA Route Administration mechanism is implemented in the [com.bankframe.services.route.adminstration](#) package. This package provides a framework for implementing Route Administration mechanisms. The package contains the following classes/interfaces:

RouteAdministrationBean	The Route Administration implementation bean.
RouteAdministration	Route Administration remote interface.
RouteAdministrationHome	Route Administration home interface.
Client	Application to test Route Administration bean functionality.

The JNDI name of the [RouteAdministrationBean](#) is [eontec.bankframe.RouteAdministration](#)

The RouteAdministrationBean's Methods

An instance of this bean exposes the following public methods to a client:

getAllRoutes()	Returns an Enumeration of Route objects for all MCA routes.
getRoute(String requestId)	Finds a route by requestId and returns an instance of that route.
deleteRoute (String requestId)	Finds a route by requestId and deletes that route. Returns void .
createRoute(String requestId, String ejbName, String description, boolean isSessionManaged)	Creates a new route. Returns void .

processDataPacket()

In order to invoke the methods of the [RouteAdministrationBean](#) the client uses the [processDataPacket\(\)](#) method.

getAllRoutes()

To invoke this method using the [processDataPacket\(\)](#) method a [DataPacket](#) with the following structure is sent by the client:

DATA PACKET NAME	GET_ALL_ROUTES
REQUEST_ID	MC002
OWNER	Usually Eontec LTD

The `processDataPacket()` method returns a vector of one or more `DataPackets` containing a `DataPacket` for each route with the following structure.

DATA PACKET NAME	ROUTE
REQUEST_ID	The <code>requestId</code> of the route
JNDI_NAME	JNDI name of the route
IS_SESSION_MANAGED	yes or no
DESCRIPTION	Description of the route
OWNER	Usually Eontec LTD

getRoute(String requestId)

To invoke this method using the `processDataPacket()` method a `DataPacket` with the following structure is sent by the client:

DATA PACKET NAME	GET_ROUTE
REQUEST_ID	MC002
ROUTE_REQUEST_ID	The <code>requestId</code> of the route to be found.
OWNER	Usually Eontec LTD

The `processDataPacket()` method returns a vector containing a `DataPacket` with the route details in it. This `DataPacket` has the same structure as one of the `DataPackets` returned by `getAllRoutes()`.

deleteRoute(String requestId)

To invoke this method using the `processDataPacket()` method a `DataPacket` with the following structure is sent by the client:

DATA PACKET NAME	DELETE_ROUTE
REQUEST_ID	MC002
ROUTE_REQUEST_ID	The <code>requestId</code> of the route
OWNER	Usually Eontec LTD

The `processDataPacket()` method returns a vector containing a `DataPacket` with a `DATA PACKET NAME` of `DELETE_ROUTE` if successful, or `ROUTE_ADMINISTRATION_EXCEPTION` if unsuccessful.

createRoute(String requestId, String ejbName, String description, boolean isSessionManaged)

To invoke this method using the `processDataPacket()` method a `DataPacket` with the following structure is sent by the client:

DATA PACKET NAME	CREATE_ROUTE
REQUEST_ID	MC002
ROUTE_REQUEST_ID	The <code>requestId</code> of the route
JNDI_NAME	JNDI name of the route
SESSION_MANAGED	yes or no
DESCRIPTION	Description of the route
OWNER	Usually <code>Eontec LTD</code>

The `processDataPacket()` method returns a vector containing a `DataPacket` with a `DATA PACKET NAME` of `CREATE_ROUTE` if successful, or `ROUTE_ADMINISTRATION_EXCEPTION` if unsuccessful.

ROUTE_ADMINISTRATION_EXCEPTION

When an exception is thrown by the above methods a `Vector` is returned containing a `DataPacket` with the following structure:

DATA PACKET NAME	ROUTE_ADMINISTRATION_EXCEPTION
REQUEST_ID	Default <code>REQUEST_ID</code> always <code>00000</code>
Message	A description of the problem which caused the exception to be thrown
OWNER	Usually <code>Eontec LTD</code>

Request Contexts

Request Contexts are objects associated with requests that store some state. This state can then be maintained across all method invocations within the request call stack. One application of storing this state is for tracking transactions from start to finish.

Request Contexts and Threads

Request Contexts are based on the fact that in an application server a request corresponds to a single thread of execution. Leveraging this fact it is possible to associate some information with each thread. At the start of the processing of a request the Request Context object is created and initialized in the `RequestRouterBean.processDataPackets()` method. This information then exists for the duration of the request and can be accessed at any time.

The com.bankframe.services.requestcontext package

The business logic for the Request Context Service is implemented in the `com.bankframe.services.requestcontext` package. This package consists of the following classes/interfaces:

<code>DataPacketsRequest</code>	A wrapper object that maps a <code>Vector</code> of <code>DataPackets</code> to a <code>Request</code> object.
<code>NullRequestContextFactory</code>	The default <code>RequestContextFactory</code> . It does not associate any context with a request.
<code>Request</code>	This is a tagging interface to identify the data that makes up a request.
<code>RequestContext</code>	This is a tagging interface used to identify objects that are associated with a request.
<code>RequestContextFactory</code>	This class creates and configures <code>RequestContext</code> instances.
<code>SampleRequestContextFactory</code>	A sample factory for creating <code>RequestContext</code> objects that store the request <code>DataPacket</code> 's <code>REQUEST_ID</code> and <code>DATA PACKET NAME</code> .

Configuring Request Contexts

To configure Request Contexts the `BankframeResource.properties` file must be modified as follows:

Specify a `RequestContextFactory` like below

```
requestContext.factory=com.bankframe.services.requestcontext.PreferredRequestContextFactory
```

where `PreferredRequestContextFactory` is used to create and associate state with the preferred `RequestContext`.

NOTE: If this setting is not modified the default `NullRequestContextFactory` is used which doesn't associate any context with a request.

Accessing the state of a RequestContext

If one needs to access the state associated with a `RequestContext` object, then the following code can be used to obtain the instance of the `RequestContext` and access the information it holds.

```
RequestContext rc = RequestContextFactory.getRequestContext()

PreferredRequestContext src = (PreferredRequestContext)rc;

Object state = src.get();
```

The `PreferredRequestContextFactory` will be the same Request Context Factory specified in `BankframeResource.properties`. In the above example the variable `state` will contain the information `PreferredRequestContext` associated with the thread of execution.

Writing Custom Request Context Factory Classes

When needing to employ the Request Context mechanism it will be necessary to write a customised `RequestContextFactory` and `RequestContext` to associate one's desired information with the thread of execution. This information to be stored needs to be available in the request sent to the `RequestRouter` that is, the Vector of `DataPackets`. The `RequestRouter` will then wrap the request in a `DataPacketRequest` object and send it to the `RequestContextFactory` class. At this point the customised `RequestContextFactory` and `RequestContext` will be called. Customising the `RequestContext` and `RequestContextFactory` are described below.

Customising the Request Context

Firstly write a `RequestContext` class, for example, `MyRequestContext` that will specify the data from the request to be associated with the thread of execution. The `MyRequestContext` class must implement the `RequestContext` interface. The `MyRequestContext` class should be a simple class with some setter and getter methods to enable access to the desired fields. However there are performance issues to consider when deciding what to associate with the thread.

Customising the Request Context Factory

Once the customised `RequestContext`, `MyRequestContext`, is written a `RequestContextFactory`, for example, `MyRequestContextFactory` must be written. To do this one should subclass the `RequestContextFactory` class and implement its abstract methods `newRequestContext()` and `configureRequestContext(RequestContext, Request)`.

- The `newRequestContext()` method should instantiate and return an instance of the new Request Context class `MyRequestContext`.
- The `configureRequestContext(RequestContext, Request)` method should take the `RequestContext` object passed as parameter and if it is an instance of the `MyRequestContext` class (which it should be), then cast it to the `MyRequestContext` class. Now extract the information one wants to associate with the thread of execution from the `Request` passed as a parameter and use the setter methods on `MyRequestContext` to associate this information with the thread.

Now the information is available at any point in the request through accessing the `MyRequestContext` object.

Request Contexts and Performance

When deciding what information one wants to associate with a thread, one must take some points into consideration.

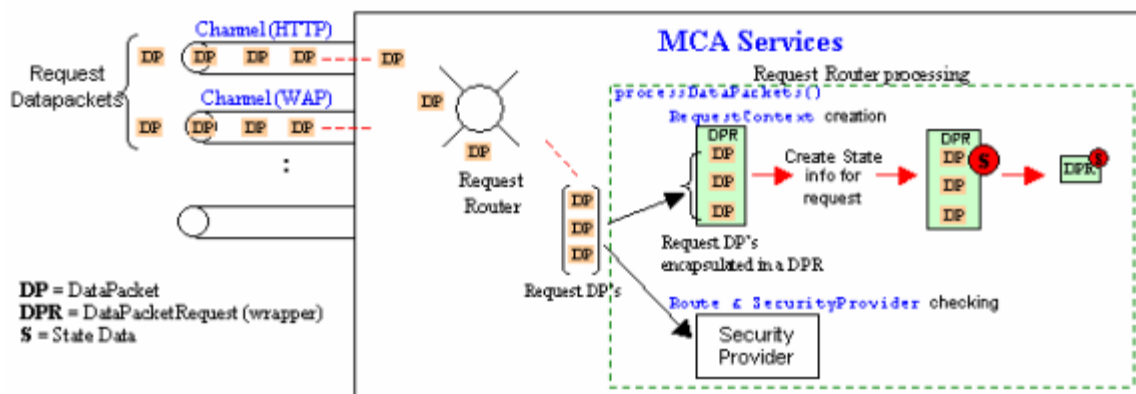
- The first point to understand is the lifecycle of the `RequestContext` object. One and only one `RequestContext` instance will be created for each thread in the application server. This instance will be re-initialized at the start of each request. This avoids unnecessary object creation overhead by re-using the `RequestContext` instance for multiple requests.
- The second point is that since there is one instance created per thread and the application may have hundreds or thousands of threads it is imperative that the `RequestContext` object does not require much memory. For example if each `RequestContext` object required 20Kb of storage and the application server is serving 5000 customers, with one thread per customer then you will need $20 \times 5000 = \sim 100\text{Mb}$ of storage. Obviously this amount of data will cause a lot of extra page faults and will significantly decrease performance and scalability.

- The third point is that since the `RequestContext` object may be used several times in the course of a request, the methods invoked on the `RequestContext` object should be of reasonable performance. For example a poor `RequestContext` implementation might use a `Map` or other `Collection` type internally to store some state. This is inadvisable since manipulating or interacting with `Collection` type objects is likely to lead to a lot of temporary objects being created. When this is being done thousands of times per second this is likely to significantly impact system performance.

Hence it is important to choose a reasonable amount of data to store and a suitable storage type for the customized `RequestContext` object.

Request Context Example

When a `DataPacket` is sent to the Request Router, this corresponds to a request on some channel. The Request Router then processes the `DataPackets` associated with this request. Within the processing the `DataPackets` for a request are wrapped inside a `DataPacketRequest` object, then the `RequestContextFactory` is called and this creates a `RequestContext` object which is used to store the state information for the request which then exists for the duration of the request.



The `RequestContextFactory` uses the `java.lang.ThreadLocal` to store the relevant `RequestContext` data for a request. Remember that it was previously stated that a request corresponds to a single thread of execution. `ThreadLocal` is used to store state for a Thread as long as it remains alive, and hence is used. The `RequestContext` can be customized in order to store specific state information for a request. In the following example the RequestId and Data Packet Name are the only state information that is stored for each request.

```
//Customized Request Context
public static class MyRequestContext implements RequestContext {
    //declare the state information required
    private String requestId;
    private String dataPacketName;

    protected MyRequestContext() {
        super();
    }
    //get and set methods for request Id
    public String getRequestId() {
        return requestId;
    }
}
```

```

    }
    public void setRequestId(String string) {
        requestId = string;
    }
    //get and set methods for the Data Packet Name
    public String getDataPacketName() {
        return dataPacketName;
    }
    public void setDataPacketName(String string) {
        dataPacketName = string;
    }
}

```

Next the customized Request Context Factory is defined, which allows the creation of new instances of the customized Request Context (MyRequestContext), and also the setting of the state information.

```

//Customized Request Context Factory
public class MyRequestContextFactory extends RequestContextFactory {
    public MyRequestContextFactory() {
        super();
    }

    protected RequestContext newRequestContext() {
        return new MyRequestContext();
    }

    protected void configureRequestContext(RequestContext requestContext, Request request)
    {
        if (request instanceof DataPacketsRequest) {
            MyRequestContext sample = (MyRequestContext) requestContext;

            DataPacketsRequest dps =(DataPacketsRequest) request;
            //set the state information
            sample.setRequestId(dps.getRequestId());
            sample.setDataPacketName(((DataPacket)dps.getDataPackets().elementAt(0)).getName());
        }
    }
}

```

Remote Notification

The Remote Notification Service provides a means for client applications to transmit notification messages to any remote machine that is registered with the notification server.

How Siebel Retail Finance Notification Works

Peer to Peer

The mid tier acts as a repository in which targets register when they log on. The server maintains a list of registered addresses, which correspond to users who are logged on. Initially when a user registers as a registered address any previous entries for that user are removed to ensure that only the latest IP address is maintained for that user.

When a user logs off the corresponding registered address are removed from the repository. The only details that must be maintained are a user ID, the IP address from where the user logged on and the

Port number that the target server is listening on. This implementation allows all types of users who are registered with the Notification Server and who have a local server running on their specific machines awaiting incoming connections, to communicate with each other.

High Level Overview

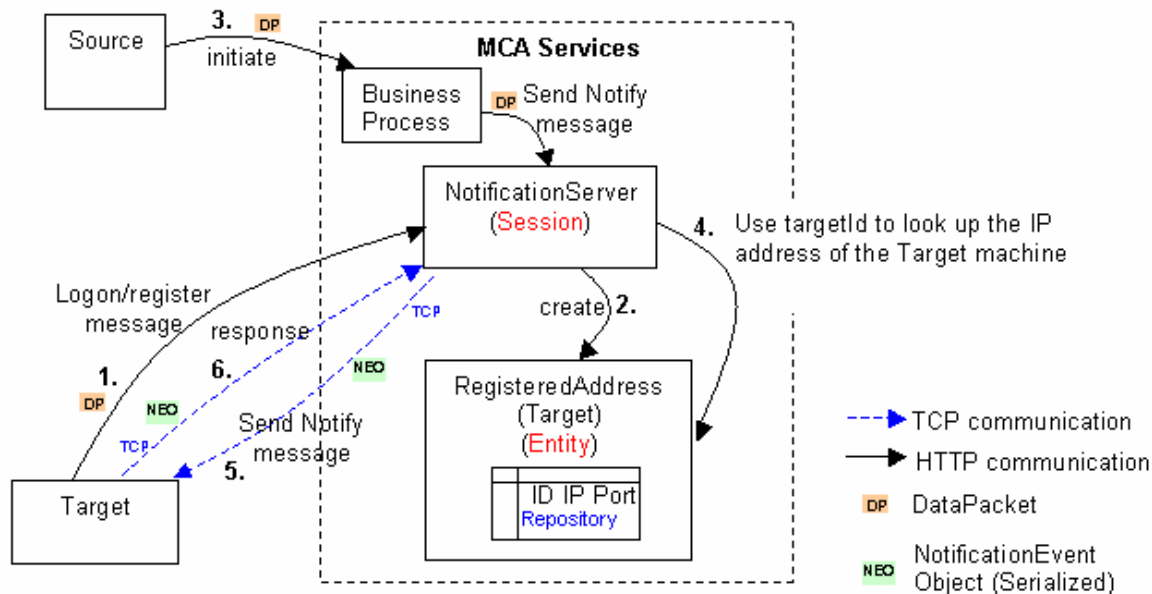
At logon the target user sends the registration request via the HttpClient to the EJB server. This then creates a record of the registration along with the IP address of the target, in the mid tier database by means of a RegisteredAddress container managed bean. The source front end communicates with the mid-tier in the usual manner. Once the target's IP address is retrieved from the mid-tier the communication from source to target is carried out by the mid tier forwarding the request to the target on behalf of the client source.

The target front-end starts a server listening on the agreed port. This port number is configured through the `BankframeResource.properties` file, and is passed to the NotificationServer when the Target registers and is stored in the database. The NotificationServer communicates with the Target machine via this port. The server is started upon target logon. This server receives incoming requests from clients and passes them to a Java thread whose job it is to deal with the message.

Remote Notification Architecture

If an event occurs on the Source workstation requiring notification then the `notifyUser(sourceId,targetId,action,date,payload)` method on the NotificationServer is called. A targetId representing the target user may or may not be passed into this method, if it is then the targetIp representing the target user's IP address is obtained using the targetId which is the primary key. If no targetId is passed in (the source doesn't know the target's ID) then the notification server selects a recipient based on a target selection algorithm specified in `com.bankframe.services.notification.targetselection`.

The steps involved in Remote Notification are outlined in the following diagram and explained below:



The following steps are involved in creating a target RegisteredAddress:

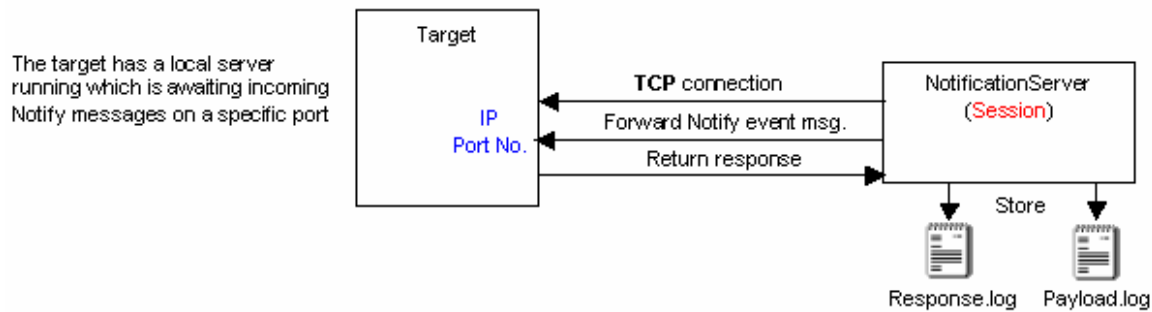
- The target registers with the notification server.
- The NotificationServer creates the new RegisteredAddress entity using the targets's ID, target's IP address and port number passed in.

The following steps are involved when a Notification Event occurs:

- Notification event occurs on the source workstation which initiates a business process on the server side. The business process then calls `notifyUser` passing source id, destination id, action, date and payload to the NotificationServer.
- If a `targetId` was passed in with the notify message then this is used to determine the appropriate IP address for the target, if no `targetId` is passed in then the method `getTargetIPForSource` is called. The default implementation of this method is to retrieve all registered addresses (targets) and select the first one. This method can be over-ridden to reflect the actual algorithm for selecting the appropriate target IP address.
- Using the selected target IP address a TCP connection is made to the target machine using the IP address and a known Port number. The notification event object is constructed and sent to the target via this connection.
- The target responds with an appropriate message. The response will be Fail or Success.

NotificationServer and Target Communication Procedure

The locations of the response and payload log files, as shown in the diagram below, are configured through the `BankframeResource.properties` file. The notification event message is in standard `DataPacket` format, within the NotificationServer this `DataPacket` is transformed into a `NotificationEvent` object. This notification event object message is a serialisable object.



The NotificationEvent object consists of the following:

<code>sourceIp</code>	The IP address of the Source workstation that the message originated on.
<code>targetIp</code>	The IP address of the Target workstation that the NotificationServer connected to
<code>Date</code>	The date the message was sent
<code>Action</code>	The action to perform on the client
<code>payload</code>	The notification event message details. This is a serializable object

Timeout and Retry Mechanism

A timeout and retry mechanism is included, which:

- prevents a socket blocking indefinitely while waiting for a response from a machine which may not be alive
- ensures that a target actually receives the Notification Event message and if not reports back a failure message

A certain number of retries is allowed until eventually a response is received or the notification fails. An appropriate message is forwarded back to the Source. Two types of messages are reported back to the Source; either Success or Failure. The timeout value and number of retries are configured in the `BankframeResource.properties` file.

Receiving Notification Event Messages

When a Target machine registers to receive notifications:

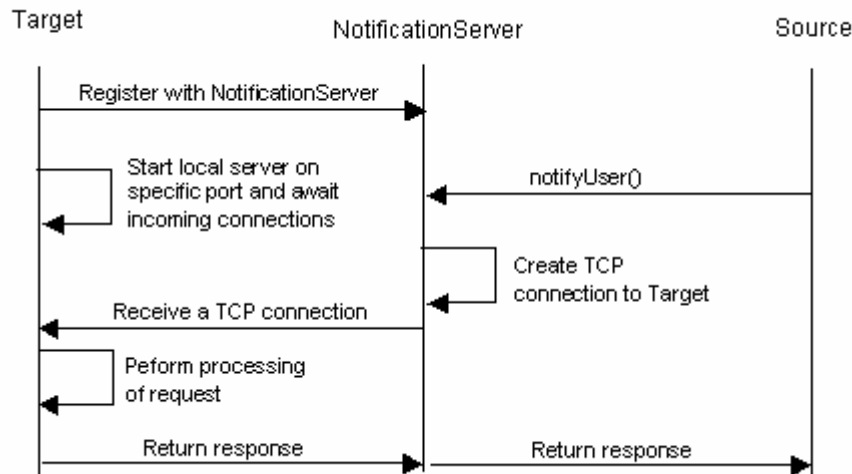
- The NotificationServer first checks to see if the TargetId is already in the RegisteredAddress table
- If it is then the TargetId is deleted and the TargetId along with the new TargetIp is updated to the RegisteredAddress table

Carrying out the registration in this way ensures that:

- If a Target workstation crashes and the TargetId remains in the RegisteredAddress table then this old value is over written

- If a Target user logs off without sending the unregister message to the NotificationServer and re-logs in on another machine the new IP address associated with this TargetId is updated to the RegisteredAddress table.

In order for a client to receive NotificationEvent messages they must have a local server running on their machines which is listening on a specified port for incoming connections. The registering process can be seen in the diagram below.



Remote Notification API

The `com.bankframe.services.notification` Package

The `com.bankframe.services.notification` package contains the following:

<code>NotificationEvent</code>	This is the NotificationEvent class which encapsulates the message to be sent to the Target
<code>SourceFrame</code>	This is an example Source front-end GUI which imitates the functionality of the Source machine
<code>TargetFrame</code>	This is an example Target front-end GUI which imitates the functionality of the Target machine
<code>TargetServer</code>	This is an example of how the target server listens for notification event messages from the notification server via a TCP connection
<code>WorkerThread</code>	This is an example worker class which shows how the notification message can be processed by the Target

The NotificationEvent Methods

<code>toString()</code>	Returns a String representation of the NotificationEvent details
<code>getSourceId()</code>	This method returns the Source ID
<code>setSourceId(String sourceId)</code>	This method sets the Source ID
<code>getTargetIp()</code>	This method gets the Target IP address
<code>setTargetIp(String targetIp)</code>	This method sets the Target IP address
<code>getPayload()</code>	This method gets the payload object
<code>setPayload(Object payload)</code>	This method sets the payload
<code>getDate()</code>	This method gets the date the message was sent
<code>setDate(String date)</code>	This method sets the date
<code>getAction()</code>	This method gets the action
<code>setAction(String action)</code>	This method sets the action

The com.bankframe.services.notification.notificationserver Package

<code>NotificationServer</code>	The remote Interface that declares the functionality <code>NotificationServerBean</code> exposes.
<code>NotificationServerBean</code>	The session bean that implements MCA Service's notification logic.

The NotificationServer Methods

<code>registerUser(String targetId, String targetIp, String targetPort)</code>	This method allows a target machine to register with the NotificationServer
<code>unregisterUser(String targetId, String targetIp, String targetPort)</code>	This method allows a target machine to un-register from the NotificationServer
<code>notifyUser(String sourceId, String targetId, String action, String date, Object payload)</code>	This method is called by the source in order to notify a target about a particular notification event

The com.bankframe.services.notification.registeredaddress Package

This package contains the implementations of the entity bean `RegisteredAddress`

<code>RegisteredAddress</code>	Remote Interface that declares the methods of the <code>RegisteredAddress</code> Entity bean.
<code>RegisteredAddressBean</code>	Container managed bean implementation.
<code>RegisteredAddressFinders</code>	The <code>RegisteredAddress</code> finders interface

The RegisteredAddressBean Methods

<code>create(String targetId,String targetIp,String targetPort)</code>	This method creates a new <code>RegisteredAddress</code>
<code>getTargetId()</code>	This method gets the Target ID
<code>setTargetId(String targeted)</code>	This method sets the Target ID
<code>getTargetIp()</code>	This method gets the Target IP address
<code>setTargetIp(String targetIp)</code>	This method sets the Target IP address
<code>getTargetPort()</code>	This method gets the Target Port
<code>SetTargetPort(String targetPort)</code>	This method sets the Target Port

The com.bankframe.services.notification.targetselection Package

This package contains a `TargetSelectionFactory` which, creates new instances of the `TargetSelectionFactoryImpl` class that is used to select a target specified by the algorithm in the `getTargetForSource(String sourceId)` method.

<code>TargetSelectionFactory</code>	This class is used to create a new instance of the <code>TargetSelectionFactory</code> implementation specified by the <code>targetSelectionFactory</code> setting in <code>BankframeResource.properties</code>
<code>DefaultTargetSelectionFactoryImpl</code>	This is the default <code>TargetSelectionFactory</code> implementation, this default algorithm returns the first Target IP address found in the <code>RegisteredAddress</code> table. The method <code>getTargetForSource(String sourceId)</code> must be implemented by any new factory implementations.

The TargetSelectionFactory Methods

<code>getInstance()</code>	This method returns a <code>TargetSelectionFactory</code> instance
<code>getTargetIPForSource(String sourceId)</code>	This method returns a target IP address based on the <code>sourceId</code> passed in.

Internationalization

This topic describes the internationalization facilities provided by MCA Services. For information on date and time localization see Data Validation.

All SRF internationalization is done on the client-side. This involves making sure that all data that needs to be localised is passed to the client in addition to any additional information that is required by the client to localise the data.

It may be appropriate to provide further localization to message arguments. For example the message `MSG001=Can not withdraw funds because account status is {0}` can be localized, however the status value in the message at `{0}` would not be. The MCA internalization framework allows for further localization by adding an additional attribute to the message key. The message can be stored in `BankframeMessages.properties` as `MSG001=Can not withdraw funds because account status is {0, 18n}`. The `18n` attribute tells MCA Services to replace the message argument value with a corresponding value from `BankframeMessages.properties`. The message can have another key `CLOSED=closed`. Therefore, a message `MSG001` with argument `CLOSED` would cause the MCA to look up the `CLOSED` key in `BankframeMessages.properties` to support status values. The resulting text message will read `Can not withdraw funds because account status is closed`.

Resource Bundles

Localised resources (that is, localised messages) are organised into resource bundles. A resource bundle is a set of property files, which contain locale specific text. For each locale a property file containing the localised text is required. The property files must follow the following naming convention: `BundleName_language_country`. The country is optional, it is only used if a language has a sub dialect specific to a country. This naming convention is required as the Java resource manager uses the class name to locate the most appropriate resource bundle for a locale. For example if a resource bundle for the Swiss-German locale was requested the resource manager would search for an appropriate resource bundle class using the following pattern:

`BundleName_de_CH`. Swiss-German locale resource bundle.

`BundleName_de`. General German language resource bundle.

`BundleName`. Root resource bundle.

So first of all the resource manager searches for the Swiss German resource bundle, if it cannot find Swiss German resources it will search for the German resource bundle, and if it cannot find German resources it will use the default resource bundle.

Localized Messages in BankframeMessages.properties

All messages are stored in a file called `BankframeMessages.properties`. Each locale will have a separate file containing the localised text for that locale. The file is named using the convention described above:

<code>BankframeMessages.properties</code>	The default messages file
<code>BankframeMessages_en_US.properties</code>	The US English message file
<code>BankframeMessages_de.properties</code>	The generic German message file
<code>BankframeMessages_de_CH.properties</code>	The Swiss German message file

MCA Internationalization Framework

The MCA Internationalization framework is implemented in the `com.bankframe.localization` package. This package contains the following classes:

BankFrameMessage

This class represents a message that can be localized. This class has the following methods:

BankFrameMessage(String messageKey)

This constructor creates a `BankFrameMessage` instance that uses the specified `messageKey` to obtain the localized message from the `BankframeMessages.properties` file

BankFrameMessage(String messageKey, String[] arguments)

This constructor creates a `BankFrameMessage` instance that uses the specified `messageKey` to obtain the localized message from the `BankframeMessages.properties` file and substitutes the specified arguments into the localized message.

BankFrameMessage(DataPacket bankframeMessageDataPacket)

This constructor creates a `BankFrameMessage` instance that uses the localization information in the specified `DataPacket`.

setMessageKey()

This method is used to set the message key. This method has the following signature:

```
public void setMessageKey(String messageKey);
```

- The `messageKey` parameter identifies the key of the localised message stored in the `BankframeMessages.properties` file

setMessageArguments()

This method is used to set the arguments for a message. This method has the following signature:

```
public void setMessageArguments(String[] arguments);
```

- The `arguments` parameter contains the arguments for the message

toString()

This method converts the `BankFrameMessage` to a localised `String`. This method has two forms:

```
public String toString();
```

- This method converts the `BankFrameMessage` using the default system locale. Use of this method is not recommended because the system locale may not match the user's locale.

```
public String toString(Locale locale);
```

- This method converts the `BankFrameMessage` using the specified locale.

toDataPacket()

This method converts the `BankFrameMessage` to a `DataPacket`. This method has the following signature:

```
public DataPacket toDataPacket();
```

- This method returns a `DataPacket` containing the information necessary for localising the message

fromDataPacket()

This method sets the `messageKey` and arguments for this `BankFrameMessage` from the information contained in the specified `DataPacket`. This method has the following signature:

```
public void fromDataPacket(DataPacket data);
```

- The data parameter specifies a `DataPacket` containing the information for the `BankFrameMessage`

BankFrameException

This class is the base class for all exceptions. This class works hand in hand with the `BankFrameMessage` class. Whereas most Java exceptions are created using a `String` error message, `BankFrameExceptions` are created using a `BankFrameMessage` error message.

This class contains the following methods:

BankFrameException()

This constructor creates an instance of `BankFrameException` using the specified `BankFrameMessage` for the error message. This constructor has the following signature:

```
public BankFrameException(BankFrameMessage message);
```

getBankFrameMessage()

This method returns the [BankFrameMessage](#) associated with this exception. This method has the following signature:

```
public BankFrameMessage getBankFrameMessage();
```

getMessage()

This method gets the error message for this [BankFrameException](#). This method has two forms:

```
public String getMessage();
```

- Using this method is not recommended since it uses the default system locale to localise the error message, which may not match the user's locale

```
public String getMessage(Locale locale);
```

- This method gets the error message for this exception, localising the message using the specified locale

toDataPacket()

This method converts the exception to a [DataPacket](#). This method has the following signature:

```
public DataPacket toDataPacket();
```

BankFrameMessageUtils

This class contains utility methods for manipulating [BankFrameMessages](#). This class contains the following methods:

parseDataPacket()

This method converts a [DataPacket](#) to a [BankFrameMessage](#). This method has the following signature:

```
public static BankFrameMessage parseDataPacket(DataPacket data);
```

- The `data` parameter is a [DataPacket](#) containing the information necessary to construct a [BankFrameMessage](#).
- A [BankFrameMessage](#) instance is returned, or `null` if the [DataPacket](#) does not contain any [BankFrameMessage](#) data.

toString()

This method converts a [DataPacket](#) containing [BankFrameMessage](#) data to a [String](#). This method has the following signature:

```
public static String toString(DataPacket bankframeMessageDataPacket, Locale locale);
```

- The `bankframeMessageDataPacket` parameter is a [DataPacket](#) containing the information necessary to construct a [BankFrameMessage](#).

- The locale parameter specifies the `Locale` to use for localizing the message
- The localized message is returned or `null` if the `DataPacket` does not contain any `BankFrameMessage` data.

containsBankFrameMessage()

This method determines if the specified `DataPacket` contains `BankFrameMessage` data. This method has the following signature:

```
public static boolean containsBankFrameMessage(DataPacket data);
```

- The `data` parameter is a `DataPacket` containing the information necessary to construct a `BankFrameMessage`.
- This method returns `true` if the `DataPacket` contains `BankFrameMessage` data, `false` otherwise.

BankFrameExceptionUtils

This class contains utility methods for manipulating `BankFrameExceptions`. This class contains the following methods:

containsBankFrameException()

This method determines if the specified `Vector` of `DataPackets` contains `BankFrameException` data. This method has the following signature:

```
public static boolean containsBankFrameException(Vector dataPackets);
```

- The `dataPackets` parameter is a `Vector` of one or more `DataPackets`.
- This method returns `true` if the first `DataPacket` in the `Vector` contains `BankFrameException` data, or `false` otherwise.

getMessage()

This method gets the error message for the `BankFrameException` data contained in the specified `Vector` of `DataPackets`. This method has the following signature:

```
public static String getMessage(Vector dataPackets, Locale locale);
```

- The `dataPackets` parameter is a `Vector` of one or more `DataPackets`.
- The locale parameter specifies the `Locale` to use for localizing the message
- The localized message is returned or `null` if the `Vector` of `DataPackets` does not contain any `BankFrameException` data.

toBankFrameException()

This method converts a `Vector` of `DataPackets` to a `BankFrameException`. This method has the following signature:

```
public static BankFrameException toBankFrameException(Vector dataPackets);
```

- The `dataPackets` parameter is a `Vector` of one or more `DataPackets`
- The `BankFrameException` is returned or `null` if the `Vector` of `DataPackets` does not contain any `BankFrameException` data.

toVectorResponse()

This method converts a `BankFrameException` to a `Vector` of `DataPackets`. This method has the following signature:

```
public static Vector toVectorResponse(BankFrameException ex);
```

- The `ex` parameter is the `BankFrameException` to be converted.
- A `Vector` containing a single `DataPacket` with the `BankFrameException` data is returned.

Sample Localization Implementations

Using BankFrameMessage

Below is some sample code that uses the `com.bankframe.localization.BankFrameMessage` class:

```
import com.bankframe.localization.BankFrameMessage;

public class Sample {

    public static final String HELLO_MSG_KEY="HELLO";

    public static void main(String[] args) {

        BankFrameMessage msg = new BankFrameMessage(HELLO_MSG_KEY,new
String[]{getUserName()});

        System.out(msg);

    }

}
```

Assuming `BankframeMessages.properties` contains the line `HELLO=Hello {0}`, the host system locale is English (`en`) and the `getUserName()` method returns a string containing `John Doe`, the above application will produce the following output:

```
Hello John Doe
```

Using BankFrameException

Below is some sample code that uses the `com.bankframe.localization.BankFrameException` class:

```
import com.bankframe.localization.BankFrameMessage;

import com.bankframe.localization.BankFrameException;
```

```

public class Sample {

    public static final String ERROR_MSG_KEY="ERROR";

    public static void main() {

        try {

            BankFrameMessage msg = new BankFrameMessage(ERROR_MSG_KEY);

            throw new BankFrameException(msg);

        } catch ( BankFrameException ex ) {

            System.out.println(ex.getMessage());

        }

    }

}

```

Assuming `BankframeMessages.properties` contains the line `ERROR=An error occurred`

And the host system locale is English (`en`) the above application will produce the output: `An error occurred`

Logging

Financial Components need messages to be logged at different times while performing processing; to meet this requirement MCA Services provides an extensible logging service. The MCA logging service is a thin bridge between different logging libraries. Logging libraries supported include:

- BEA WebLogic Logging Framework
- Apache Foundation LOG4J framework
- Generic Console output

The WebLogic Logging framework is a proprietary API available in WebLogic 6.1 and later. It enables logging messages to be logged directly into WebLogic's own log file. The benefits of this are:

- MCA logging messages are logged in sequence in the same file as WebLogic logging messages. This aids problem determination since it is possible to see the exact order in which events occurred
- Administration and configuration of the logging system can be done via the WebLogic Administration Console
- Logging Messages can be viewed in the WebLogic Administration Console

The LOG4J framework is a widely used logging framework developed under the auspices of the Apache Foundation. It provides an extremely rich library that be configured to format logging messages into any required format and to be logged to a number of different destinations including:

- Console output

- File output
- Rolling file output
- UDP datagrams
- Unix Syslog
- NT Event Log

The Generic console support enables log messages to be printed directly to the console. This option is provided for when neither of the two options above is available.

Logging Package and Classes

The logging service is implemented by the `com.bankframe.services.logger` package and its sub-packages.

The `com.bankframe.services.logger` package

The `ELogger` interface defines the methods that the logging service provides:

<code>boolean isDebugEnabled()</code>	Indicates whether <code>DEBUG</code> level messages should be logged. This method should be called before logging large <code>DEBUG</code> messages, in order to improve overall performance
<code>void debug(String msg)</code>	Logs the specified message at <code>DEBUG</code> level
<code>void debug(String s, Throwable throwable)</code>	Logs the specified exception at <code>DEBUG</code> level
<code>void info(String msg)</code>	Logs the specified message at <code>INFO</code> level
<code>void info(String s, Throwable throwable)</code>	Logs the specified exception at <code>INFO</code> level
<code>void warn(String msg)</code>	Logs the specified message at <code>WARN</code> level
<code>void warn(String s, Throwable throwable)</code>	Logs the specified exception at <code>WARN</code> level
<code>void error(String msg)</code>	Logs the specified message at <code>ERROR</code> level
<code>void error(String s, Throwable throwable)</code>	Logs the specified exception at <code>ERROR</code> level
<code>void fatal(String msg)</code>	Logs the specified message at <code>FATAL</code> level
<code>void fatal(String s, Throwable throwable)</code>	Logs the specified exception at <code>FATAL</code> level

The `ELoggerFactory` class is used to create `ELogger` instances. This class provides the following method:

<pre>public static ELogger getLogger(Class subsystem)</pre>	<p>This method returns the logger for the specified subsystem. This method should be called by Siebel Retail Finance Financial Components to create <code>ELogger</code> instances.</p>
---	---

The `com.bankframe.services.logger.wl61` package

This package contains classes that provide an `ELogger` instance that communicates with the WebLogic 6.1 Logging Framework. The classes in this package must not be called directly by Siebel Retail Finance Financial Components

The `com.bankframe.services.logger.log4j` package

This package contains classes that provide an `ELogger` instance that communicate with the LOG4J logging framework. The classes in this package must not be called directly by Siebel Retail Finance Financial components

The `com.bankframe.services.logger.console` package

This package contains classes that provide an `ELogger` instance that prints logging messages directly to the `System.out` stream. The classes in this package must not be called directly by Siebel Retail Finance Financial components

Using the Logging Service

Logging Levels

There are five levels of logging which can be used:

<code>FATAL</code>	Use only in cases where it is impossible for the Siebel Retail Finance application to recover or continue.
<code>ERROR</code>	Use when the request cannot be processed but the overall system is still functioning.
<code>WARN</code>	Use the <code>WARN</code> level for recording exceptions that indicate that something may be wrong but do not prevent the request being processed.
<code>INFO</code>	Use the <code>INFO</code> level for providing information about the running system, for example timing information.
<code>DEBUG</code>	Use the <code>DEBUG</code> level for recording information about how the system works, to aid in determining the cause of runtime problems.

These log levels are used to determine if a log message is of interest for a particular runtime configuration. For example, in a production system MCA Services could be configured to only log messages which are `FATAL` and the actual SRF Modules could log messages of `WARN` or higher.

Logging Subsystems

In a production system it is useful to be able to filter log messages by the functional area that they belong to, for example to be able to only view log messages relating to funds transfer. To enable this to be done the logging messages produced by the Siebel Retail Finance Solution must be categorised. The simplest way to do this is to categorise messages by the name of the class from which the message was produced. Since the names of all Siebel Retail Finance classes indicate which functional area they belong to, this becomes a powerful means for filtering messages by functional area.

Logging Best Practices

When writing a log statement in your code you have to determine what the message will be, what log level it requires and what subsystem it should be sent to. Follow the guidelines below to ensure you log messages appropriately

Define a Private Static Log Variable

Each Financial component should define a private static final `log` variable coded as follows:

```
import com.bankframe.services.logger.ELogger;

import com.bankframe.services.logger.ELoggerFactory;

...

public class Foo {

    private static final ELogger log = ELoggerFactory.getLogger(Foo.class);

}
```

Defining a `log` instance for each class enables logging to be switched on and off by functional area. This is important when trying to detect the cause of problems in a production system. In a production system it will not be feasible to turn on logging in all classes because this would produce such a large volume of logging information that it would degrade the performance of the system. Instead it must be possible to configure only a subset of logging messages to be turned on. The full name of each class is used to uniquely identify each `ELogger` instance. The `ELoggerFactory` class caches `ELogger` instances so that only one instance will be created per `ELogger` subsystem.

The log variable must be `static` so that it can be shared between all instances of that class. It must be `private` so that it is not visible by sub-classes. Making the variable `final` guarantees that it cannot be reassigned, thus assuring that there will only ever be one logger instance per class, in effect the logger instance becomes a singleton.

Always invoke the logger via the `log` variable as defined above, for example:

```
public class SomeClass {

    ...

    public void someMethod() {

        log.debug("This is a debug message");

    }

}
```

```
...
}
```

This ensures that the correct logger for the current class is always invoked.

Logging Exceptions

Always use the overridden logging method provided for logging exceptions, for example:

```
...
try {
    <some code which throws an exception>
} catch (Exception ex) {
    log.warn( "An error occurred", ex);
}
...
```

This will ensure that the full stack trace for the exception is logged. Having a full stack trace for an exception makes it much easier to determine the root cause of a problem.

Use the `isDebugEnabled()` Method

Even though logging output may be turned off in a production system the method calls to the logging framework are still invoked. If the arguments to the logging method involve time consuming evaluations then the overall performance of the system will be degraded, sometimes by a large amount. This is particularly true of `DEBUG` level log messages, which often print out large amounts of information such as the contents of a `DataPacket`. Therefore it is extremely important to ensure that these expensive `DEBUG` level log messages are not invoked when the system is running in production mode. This can be accomplished using code similar to the following:

```
...
if ( log.isDebugEnabled() ) {
    log.debug("These are the contents of the datapacket : " + someDataPacket);
}
...
```

Correct Log Level

When a system is running in production mode it should produce very little log output, therefore it is important to ensure that logging messages are logged at the correct level. For example it might be tempting to log all exceptions at `ERROR` level, however this would not be correct. Only exceptions that actually represent a true error condition, such as a `RemoteException` should be logged at this level.

The Logging context

When examining a large log file that contains many different log messages from many different threads it can be difficult to determine which log messages are related. Therefore it can be helpful to prepend information to each log message to better identify the source of the message. The `ELogger.Context` interface provides the means to do this. This interface has the following methods:

<code>void push(String context)</code>	This method pushes the specified String onto the context stack.
<code>void pop()</code>	This method removes the topmost element on the context stack.

Each thread will get its own logging context. This means by pushing a descriptive string onto the logging context it becomes possible to identify which thread produced a particular log message.

The `ELogger.Context` interface is accessed via the `getContext()` method of the `ELogger` interface.

The following sample implementation identifies all logging messages from within a financial component, or any other financial components it invokes:

```
public class SomeFinancialComponentBean {

    ...

    public Vector processDataPacket(DataPacket dataPacket) {

        try {

            log.getContext().push("SomeFinancialComponent");

            log.debug("This is a debug message");

            ...

        } finally {

            log.getContext().pop();

        }

    }

}
```

Now all logging calls from within `SomeFinancialComponentBean` will be prefaced with the string: `'SomeFinancialComponentBean'` making it easier to identify those logging messages.

Techniques for problem resolution using the logging framework

Examine logged stack traces

When an exception is logged, the full stack trace for that exception is logged. This stack trace should show the class and line number where the exception was raised. Often this information is sufficient to identify the cause of a problem

Filter by functional area

If you are attempting to identify the cause of a problem in a production system you can opt to turn on logging for only a subset of code, for example at component level. The following example uses the Transfers component of the Teller Module, and LOG4J is used for the logging.

The Transfers component is implemented in two packages:

```
com.bankframe.bp.retail.solutionset.transfers
com.bankframe.bp.retail.solutionset.impl.transfers
```

Since the loggers are created by passing a `Class` object to the `ELogger.getLogger()` method, each logger instance is categorised by the name of the class that created it. Thus LOG4J can be configured to only log messages produced by a specific class or package. To configure LOG4J to only display messages produced by the two packages above, the LOG4J configuration file;

`log4j.properties`, needs to be configured as follows:

```
# Default to only logging ERRORS

log4j.rootLogger=ERROR, CONSOLE

log4j.appender.CONSOLE=org.apache.log4j.ConsoleAppender

# Turn on logging of DEBUG and above messages for the Transfers functional area

log4j.logger.com.bankframe.bp.retail.solutionset.transfers=DEBUG

log4j.logger.com.bankframe.bp.retail.solutionset.impl.transfers=DEBUG
```

Filter by Logging Context

When there is a large volume of logging information being produced by logs it can become difficult to determine the order in which events occurred, for example, if there is a possibility that there is a problem somewhere within the Transfers component, the logging context can easily be used to identify all method calls that are invoked from within the transfers component. This is accomplished by adding the code below to the `processDataPackets()` method of the `TransfersSessionBean` class:

```
public Vector processDataPackets(Vector allData) throws ProcessingErrorException
{
    try {
        log.push("Transfers");
```

```

    Vector response = super.processDataPackets(allData);

    if (!DataPacketUtils.isValidResponse(response, false, null, false)) {

        this.getSessionContext().setRollbackOnly();

    }

    return response;

} finally {

    log.getContext().pop();

}

}

```

The `log.getContext().push("Transfers")` method call will cause the text 'Transfers' to be prepended to all log messages generated within the Transfers component, or any other components that the Transfers component calls. Then when examining the log files you can search for the 'Transfers' string to quickly identify those methods invoked from within the Transfers component.

Configuring the Logging Service

The logging service is configured by entries placed in the Java System Properties, or `eloggerfactory.properties` in the application classpath. These entries are defined at application server startup time, and cannot be changed once the application server has started.

Configuring the Logging Implementation factory Class

The first parameter to set is the one that determines which logging implementation to use. The parameter is set by specifying the following argument in the application server startup script:

```
java -Dcom.eontec.mca.elogger.factory=<logging implementation factory class>
```

Where `<logging implementation factory class>` is the full name of the factory class for the logging framework that you wish to use

The valid values for this setting are as follows:

<code>com.bankframe.services.logger.wl61.WL61LoggerFactory</code>	WebLogic 6.1 logging
<code>com.bankframe.services.logger.log4j.LOG4JLoggerFactory</code>	LOG4J logging
<code>com.bankframe.services.logger.console.ConsoleLoggerFactory</code>	Console logging

If this setting is not defined as a Java System property, the logging service will look for the property in a `eloggerfactory.properties` file. If the file does not exist, or the object defined cannot be instantiated, then the logging service will default to using an instance of `com.bankframe.services.logger.console.ConsoleLoggerFactory`

By checking Java System property first, and then `eloggerfactory.properties`, the logging service allows for enterprise applications deployed in the same server to have separate logging factories.

Enabling or Disabling All Logging

The entire logging framework can be enabled or disabled by specifying the following argument in the application server startup script to `true` or `false`:

```
java -Dcom.eontec.mca.elogger.enabled=<true|false>
```

The value of this setting is case sensitive.

Configuring WebLogic Settings

The following settings can be configured in the `BankframeResource.properties` file when using the WebLogic logging framework:

```
wl61.debugLoggingEnabled=<true|false>
```

This setting determines whether DEBUG level log messages should be forwarded to the WebLogic logging framework. This setting is case sensitive

```
wl61.redirectDebugToInfo=<true|false>
```

This setting determines whether DEBUG level log messages should be forwarded as INFO level messages to the WebLogic logging framework. This setting is case sensitive

Configuring LOG4J Settings

The following settings can be configured in the `BankframeResource.properties` file when using the LOG4J logging framework:

```
log4j.config.path=</path/to/some/log4j.properties>
```

This setting determines which LOG4J configuration file to use for configuring LOG4J. This setting must specify the absolute path to the properties file

```
log4j.config.refresh=<some time value in seconds>
```

This sets how often LOG4J checks its configuration file to see if any configuration changes have occurred. This value is specified in seconds

Please consult the LOG4J website for more detailed information on configuring LOG4J

Configuring Generic Console Logger Settings

The following setting can be configured in the `BankframeResource.properties` file when using the Generic Console logging framework:

```
console.logger=<logging level to use>.
```

Only `DEBUG` level messages are currently available. To switch debug level messages off in the console logger leave the `console.logger` value blank. This reduces console output and improves performance.

Configuring the value as `console.logger=DEBUG` enables debug level messages.

Integrating with other Logging Frameworks

The MCA Logging Service is designed to be extensible so that it can be adapted to direct logging messages to any logging service. This topic describes the steps required to do this using the `com.bankframe.services.logger.console` package as an example

Create a class that implements the ELogger interface

This class must do the actual logging of the logging messages. In most implementations this class will really be an adaptor class that redirects the logging message to third party logging framework. In the case of `ConsoleLogger` this class prints the message to the console using calls to `System.out.println()`.

Create a class that implements the ELogger.Context interface

This class must maintain a stack of per thread context information. Most implementations can just delegate this task to the `com.bankframe.services.logger.ELoggerContext` class:

```
protected static class ConsoleContext implements ELogger.Context {

    public void push(String context) {

        ELoggerContext.push(context);

    }

    public String pop() {

        return ELoggerContext.pop();

    }

    public ConsoleContext() {

    }

}
```

Create a factory class that extends ELoggerFactory

This class is responsible for creating `ELogger` instances. This class must extend `ELoggerFactory` and provide an implementation for the abstract `createLogger()` method. This method must create an `ELogger` instance for the specified subsystem. It should not cache instances as `ELoggerFactory` does this itself. Below is the source code for `ConsoleLoggerFactory`:

```
public class ConsoleLoggerFactory extends ELoggerFactory {

    public ConsoleLoggerFactory() {

        super();

    }

    protected ELogger createLogger(String subsystem) {
```

```

        return instance;
    }

    protected final static ELogger instance = new ConsoleLogger();
}

```

Since the console based logger only ever has one instance it creates a single static instance and always returns that through the `createLogger()` method.

Update application server startup script

To use your custom logger you must update the `com.eontec.mca.elogger.factory` setting in your application server startup script as follows:

```
java -Dcom.eontec.mca.elogger.factory=<logging implementation factory class>
```

Where `<logging implementation factory class>` is the full name of the factory class for the logging framework that you wish to use.

Logging Deprecations

BankFrameLog

The `com.bankframe.services.log.BankFrameLog` class has been deprecated and the `BankFrameLog` class has been updated to redirect all logging messages to the Logging Service described in this chapter

ESystem.out

The `com.bankframe.ESystem` object has been deprecated. The `ESystem` class has been updated to redirect all logging messages to the Logging Service described in this chapter. As there is no argument for subsystems all messages logged using the `ESystem` object will be sent to the `com.bankframe` subsystem.

Audit

The MCA Audit Service provides the means to record an audit of transactions carried out by SRF Modules.

Audit Classes and Package Structure

The Audit Service is located in the `com.bankframe.services.audit` package and its implementation is in the `com.bankframe.services.impl.audit` package.

Configuring the Audit Service

The Audit Service uses an Audit Provider framework (similar in operation to the Security Provider) to dispatch Audit requests to an Audit Implementation. The interface of the Audit Provider is `com.bankframe.services.audit.BankFrameAuditProvider` and all custom Audit Providers must implement this interface. MCA is supplied with two Audit Provider implementations:

```
com.bankframe.services.audit.NullBankFrameAuditProvider
```

```
com.bankframe.services.audit.DefaultBankFrameAuditProvider
```

The Audit Provider is configured in the `BankframeResource.properties` file using the `audit.provider` key and its value is set to the Audit Provider class, which is required for use in the runtime system.

For example, if a test MCA installation does not require any audit functionality then the Null Audit Provider would be configured as follows:

```
audit.provider=com.bankframe.services.audit.NullBankFrameAuditProvider
```

`com.bankframe.services.audit.NullBankFrameAuditProvider`

The `com.bankframe.services.audit.NullBankFrameAuditProvider` provides a dummy implementation which does not send any dispatched requests to an Audit Service. This Audit Provider can be used to switch off all Auditing of an MCA system and is often used in test installations which don't require an audit function.

`com.bankframe.services.audit.DefaultBankFrameAuditProvider`

The `com.bankframe.services.audit.DefaultBankFrameAuditProvider` dispatches to the default MCA Audit Service. This default service is implemented by three EJBs:

EJB Bean Name	EJB Implementation Package	EJB Type
<code>AuditBean</code>	<code>com.bankframe.services.impl.audit</code>	Session
<code>AuditRoute</code>	<code>com.bankframe.services.impl.audit.auditroute</code>	CMP Entity
<code>AuditRecord</code>	<code>com.bankframe.services.impl.audit.auditrecord</code>	CMP Entity

The `AuditBean` session EJB contains the logic of the audit service. The `AuditRoute` is an entity EJB that maps to a lookup table on the database which maps a Financial Component's `REQUEST_ID` to the Audit Service. This allows a BankFrame system to be configured to only a specified set of routes. Finally, the `AuditRecord` entity EJB maps to the `AUDIT_TRAIL` table on the database and contain the details of an audit.

An `AuditRecord` stores the following attributes for each Audit event:

```
AUDIT_DATE
```

```
AUDIT_TIME
```

```
REQUEST_ID
```

`REQUEST``RESPONSE`

The `REQUEST` and `RESPONSE` attributes are large strings (stored as `VARCHAR(7000)` in the underlying database) which contain an XML representation of the client request and the servers response respectively.

When MCA is configured to use this Audit Provider then the `RequestRouter` behaves as follows:

- Just before the `RequestRouter` returns a response to a client it invokes the `com.bankframe.services.audit.DefaultBankFrameAuditProvider`
- This provider performs a lookup on the Audit session EJB.
- The provider then calls the `audit()` method, passing in the current `REQUEST_ID`, the request and the response which is about to be returned.
- The Audit Session EJB then looks up the `AuditRoutes` entity EJB to enquire if the current `REQUEST_ID` represents a Fianancial Component which needs to be audited.
- If the route is auditable, then the Audit session EJB creates an `AuditRecord` entity EJB instance to contain the current date, time, `REQUEST_ID`, request and response and then stores them to the database.

Configuring Routes to the Audit Service

If the `audit.provider` is set to `DefaultBankFrameAuditProvider`, than the `RouteServlet` will show an extra option, as follows:

- Configure Default Audit Service

Selecting this displays the options available within the `AuditServlet`, which are as follows:

- Add a route to the Audit Service
- Delete a route from the Audit Service
- List all routes mapped to the Audit Service

Using these features any Fianancial Component may be added or deleted from the Audit Service, or a list of all the Fianancial Components currently mapped to the Audit Service is available. It is worth noting that deleting a Fianancial Component (using its `REQUEST_ID`) from the Audit Service does not delete it from the Routing Service.

Calling the Audit Service from within custom code

If an Audit event is required in custom code then the `com.bankframe.services.audit.AuditUtils` class can be used. This class contains the following methods,

```
audit(String requestId, Vector
request, Vector response)
```

This method is the same as used by the `RequestRouter`. It takes a `REQUEST_ID` and a request/response set of `DataPackets`.

<code>audit(Vector datapackets)</code>	This method is used when the concept of a <code>REQUEST_ID</code> and a request/response set of <code>DataPackets</code> make no sense within the context of the audit call. In this case then the database will have the text 'AUDIT' in the place of <code>REQUEST_ID</code> and both the request and response will contain the same XML representation of the <code>DataPackets</code> .
--	---

Exceptions in the Audit Service

Because the Audit Service partakes in the overall transaction (often initiated by the `RequestRouter`) and is a critical component, if an exception occurs within the Audit Service then the entire transaction is rolled back.

If you want this behavior in custom code which calls the Audit Service then calls to the `AuditUtils` class should be nested within a try/catch block which catches exceptions of type `ProcessingErrorException` and rollback the current transaction (using the `setRollbackOnly()` method on the `EJBContext` object) if the exception is caught.

It should be noted that the `EJBContext` object is usually only available within the context of an EJB.

For example,

```
try {
    AuditUtils.audit(requestId, request, responses);
} catch (ProcessingErrorException ex) {
    this.getSessionContext().setRollbackOnly(); //rollback tx
}
```

Timing Points

The Timing Point Service provides a facility for determining the length of time required for Siebel Retail Finance components to carry out their actions. The service is very useful in aiding the identification of performance bottlenecks. The service is highly flexible; allowing configuration of output into different formats while writing to either file or console, providing a framework for writing custom factory classes to create specialized Timing Points, and allowing for plug-in analyzer classes to carry out heuristics and analysis.

The `com.bankframe.services.trace` package

The business logic for the Timing Point Service is implemented in the `com.bankframe.services.trace` package. This package consists of the following classes/interfaces:

<code>BankFrameTrace</code>	Deprecated. To be replaced by Timing Point created through <code>TimingPointFactory</code> class.
	Provides a facility for determining the length of

	time required for Siebel Retail Finance components to carry out their actions.
<code>DefaultTimingPointAnalyser</code>	This class logs a Timing Point. It provides no analyzing of the Timing Point.
<code>DefaultTimingPointFactory</code>	This class is the default class used for the creation of Timing Points.
<code>EndToEndTrace</code>	Deprecated. To be replaced by Timing Point created through TimingPointFactory class. This class enables the sampling of elapsed time between timing points, aiding the identification of performance bottlenecks.
<code>NullTimingPointAnalyser</code>	This class logs a Timing Point. It provides no analyzing of the Timing Point.
<code>TimingPoint</code>	This class is a Timing Point. It is used in order to time events or actions within Siebel Retail Finance code.
<code>TimingPointAnalyser</code>	Implementers of this interface analyze a timing point. The <code>TimingPointUtil</code> class will call the <code>analyse()</code> method of an implementing class to allow some additional custom analysis to be done.
<code>TimingPointConstants</code>	Constants used for Timing Points.
<code>TimingPointFactory</code>	This class creates and configures Timing Point instances.
<code>TimingPointProperties</code>	This class is used for the storing of optional key value pairs for inclusion in the logging of Timing Points.
<code>TimingPointUtil</code>	This class provides utilities to work with Timing Points.

BankFrameTrace

This class provides a mechanism for creating a `Trace` object, calling `Trace.start()` to start recording the elapsed time, and finally calling `Trace.stop()` to finish recording. When `Trace.stopAndReport()` is called an informational message is displayed in the log indicating the elapsed time, for example,

```
Trace trace = new Trace();

trace.start("A sample description here");

...some code here ...

trace.stopAndReport();
```

NOTE: The use of this mechanism for time measurement has been deprecated and has been replaced by the use of a [TimingPointFactory](#) for creation of Timing Points.

EndToEndTrace

The [EndToEndTrace](#) class is similar to the [BankFrameTrace](#) class, however it provides the added extra of being able to specify the logging of timing points at specific intervals through the following setting in the [BankframeResource.properties](#) file:

```
trace.sampleSize
```

Configuring this setting to, for example, `trace.sampleSize =20` means that after every 20 requests, the tracing times for the previous 20 requests will be written to the console. The default setting is 1000.

It is also possible to disable the [EndToEndTrace](#) utility through the [BankframeResource.properties](#) file. This was not possible in the [BankFrameTrace](#) class. Do so by modifying the `trace.enabled` setting in the [BankframeResource.properties](#) file as follows:

```
trace.enabled=false
```

this will disable the utility while setting it to `true` will enable it.

NOTE: This class has now been deprecated. This is because recorded timing points are stored by associating them with the `java.lang.ThreadLocal` variable via a [HashMap](#). This has a performance overhead, especially if, for example, one is storing 1000 timing points within the [ThreadLocal](#) variable. The [EndToEndTrace](#) class is to be replaced by creating a [TimingPoint](#) through a [TimingPointFactory](#) class.

TimingPoint

The [TimingPoint](#) class is used to time events or actions within Siebel Retail Finance code. A Timing Point records the start time, object and also the subsystem in which the timing point occurs. Subsystems are a mechanism by which it is possible to group Timing Points together that is, creating a Timing Point as part of a subsystem and enabling that subsystem ensures the Timing Point, and all other Timing Points in that subsystem, are logged to file or disk as appropriate. The Timing Point is recorded by calling the `exit()` method which will pass the Timing Point onto a utility class that will then process it.

TimingPointProperties

This class is used for the storing of mandatory and optional key/value pairs for inclusion in the logging of timing points. Its constructor takes as parameter an array of [Objects](#). These objects form the properties to be included in the timing point logging. This array of [Objects](#) must be instantiated in the form:

```
Object[] objects = new Object[]{key0, value0, key1, value1, key2, value2};
```

where keyx is the key that indexes valuex.

This array of [Objects](#) is then used to instantiate a [TimingPointProperties](#) object as follows:

```
TimingPointProperties properties = new TimingPointProperties(objects);
```

The variable `properties` then forms the parameter for the construction of a `TimingPoint`.

TimingPointFactory

This abstract class is used for the creation of Timing Points. The `createTimingPointFactory()` method creates an instance of the concrete singleton `TimingPointFactory` class as specified by the following setting in the `BankframeResource.properties` file:

```
timingPoint.factory=com.bankframe.services.trace.DefaultTimingPointFactory
```

where `DefaultTimingPointFactory` is the default `TimingPointFactory` class.

Writing Timing Points into Code

To place a Timing Point in a suitable location in the code, the following must be done:

- Create an `Object` Array containing all the properties one wishes to associate with the Timing Point.
- Create a `TimingPointProperties` object using this Object Array.
- Use the `TimingPointFactory.getTimingPoint()` method to create a Timing Point.

Use the following code as an example:

```
//create the Object Array

Object[] objects = new Object[]{TimingPointConstants.TIMING_POINT_SUBSYSTEM,
BankFrameLogConstants.MCA_SUBSYSTEM, TimingPointConstants.TIMING_POINT_TYPE,
"Request Router", TimingPointConstants.TIMING_POINT_MAJOR_TYPE,
TimingPointUtil.MAJORTYPE_SERVLET_STRING,
TimingPointConstants.TIMING_POINT_REQUEST, this};

//create the TimingPointProperties object and create the Timing Point

TimingPoint tp = TimingPointFactory.getTimingPoint(new
TimingPointProperties(objects));
```

And to stop or exit this Timing Point:

```
tp.exit(this);
```

Customized TimingPointFactory classes

It is possible to write a customized `TimingPointFactory` class and specify its use instead of the `DefaultTimingPointFactory`. A customized class is useful when adding some extra properties to a Timing Point which may not be available in the client of the `TimingPointFactory.getTimingPoint()` method. It can also serve as a place for doing operations on the contents of the `TimingPointProperties` object used to create a Timing Point.

Guidelines for writing a Customized TimingPointFactory class. This customized class must, at the least, extend the `com.bankframe.services.trace.TimingPointFactory` class and provide an

implementation of the `configureTimingPoint()` method. The `configureTimingPoint()` method must do the following:

- create a new instance of a `TimingPoint`.
- set the `startTime` on the newly created `TimingPoint`.
- set the `user` on the newly created `TimingPoint`.

The following step should be done for any of the following values which appear as a key in the `TimingPointProperties` object passed as parameter to the `configureTimingPoint()` method

```
TimingPointConstants.TIMING_POINT_START_TIME
TimingPointConstants.TIMING_POINT_END_TIME
TimingPointConstants.TIMING_POINT_ELAPSED_TIME
TimingPointConstants.TIMING_POINT_SUBSYSTEM
TimingPointConstants.TIMING_POINT_USER
TimingPointConstants.TIMING_POINT_REQUEST
TimingPointConstants.TIMING_POINT_RESPONSE
TimingPointConstants.TIMING_POINT_TIMING_POINT_ID
TimingPointConstants.TIMING_POINT_THREAD_ID
TimingPointConstants.TIMING_POINT_TYPE
TimingPointConstants.TIMING_POINT_MAJOR_TYPE
TimingPointConstants.TIMING_POINT_HOST_RECORDING
TimingPointConstants.TIMING_POINT_SERVLET_RECORDING
TimingPointConstants.TIMING_POINT_CUSTOM_RECORDING
TimingPointConstants.TIMING_POINT_TXN_HANDLER_RECORDING
```

So for example, if the `TimingPointProperties` object had a key of `TimingPointConstants.TIMING_POINT_MAJOR_TYPE`, one should do the following:

set the `majorType` on the newly created `TimingPoint`, if there was a value returned for `majorType` in the following code:

```
String
majorType=(String)properties.getProperty(TimingPointConstants.TIMING_POINT_MAJOR_
TYPE);
```

At this point, if a value was found, the property should be removed from the `TimingPointProperties` object named `properties` using the following code:

```
properties.removeProperty(TimingPointConstants.TIMING_POINT_MAJOR_TYPE);
```

Once all these keys have been addressed and removed from `properties`, any additional processing or setting values in the `properties` object should be done now.

Finally the remaining `properties` from the passed `TimingPointProperties` object should be set on the `TimingPoint` as follows:

```
timingPoint.setProperties(properties)
```

where `timingPoint` is the `TimingPoint` created as first step of this `configureTimingPoint()` method.

DefaultTimingPointFactory

This class is used for the creation of Timing Points. The class extends the abstract class `TimingPointFactory` and provides an implementation for the `configureTimingPoint()` method. The `configureTimingPoint()` method uses the `TimingPointProperties` object passed as parameter to create and set values on a Timing Point.

TimingPointAnalyser

Classes that implement this interface are used to analyze a Timing Point. Implementing classes will write an `analyse()` method, taking a `TimingPoint` object as parameter. The `TimingPointUtil` class will call the `analyse()` method of an implementing class to allow some additional custom analysis to be done. The default `TimingPointAnalyser` is the

`com.bankframe.services.trace.DefaultTimingPointAnalyzer` class that only prints the `TimingPoint` object passed as parameter to the console/file log. It is possible to write Custom `TimingPointAnalyser` classes and have their `analyse()` method called during execution. Simply implement the `TimingPointAnalyser` interface, replace the default setting in `BankframeResource.properties` file with the new custom `TimingPointAnalyser` class as follows:

```
timingPoint.analyzerClassName=com.bankframe.services.trace.myCustomTimingPointAnalyzer
```

where `com.bankframe.services.trace.myCustomTimingPointAnalyzer` is the fully qualified name of this new custom class.

Configuring Timing Points

The settings in the `BankframeResource.properties` file that control the configuration of Timing Point Services are as follows:

EndToEndTrace

EndToEndTrace is set as follows:

```
trace.sampleSize=1000

trace.enabled=true
```

timingPoint

Timing Points are set as follows:

```
timingPoint.enabled=true
```

where `timingPoint.enabled` can have value of `true` or `false`

timingPoint.writePointsToDisk

`timingPoint.writePointsToDisk` is set as follows:

```
timingPoint.writePointsToDisk=true
```

where `timingPoint.writePointsToDisk` can have a value of `true` or `false` and `timingPoint.writePointsToDisk=true` means data will be written to console and not to file.

timingPoint.subsystem.BANKFRAME.MCA

`timingPoint.subsystem.BANKFRAME.MCA` is set as follows:

```
timingPoint.subsystem.BANKFRAME.MCA=BANKFRAME.MCA
```

where Timing Points can be grouped in a subsystem named `BANKFRAME.MCA`. It is possible to have many settings with the prefix `timingPoint.subsystem` and this means all subsystems listed here will have their data flushed to file or console.

timingPoint.doSummary

`timingPoint.doSummary` is set as follows:

```
timingPoint.doSummary=false
```

this will flush a summary all timing points to file or console.

timingPoint.fileName

`timingPoint.fileName` is set as follows:

```
timingPoint.fileName=/export/home/server/bea/user_projects/eontec/timingpoints.log
```

this will flush the timing points to the file

```
/export/home/server/bea/user_projects/eontec/timingpoints.log
```

timingPoint.bufferSize

`timingPoint.bufferSize` is set as follows:

```
timingPoint.bufferSize=1000
```

the maximum size of buffer to hold timing points. Once this is exceeded all timing points will be flushed to file or console.

timingPoint.analyzerClassName

`timingPoint.analyzerClassName` is set as follows:

```
timingPoint.analyzerClassName=com.bankframe.services.trace.DefaultTimingPointAnalyzer
```

where `com.bankframe.services.trace.DefaultTimingPointAnalyzer` is the name of the analyzer class to process timing points.

timingPoint.transactionHandler.recording

timingPoint.transactionHandler.recording is set as follows:

```
timingPoint.transactionHandler.recording=true
```

where `timingPoint.transactionHandler.recording` is an alternative to subsystems and would be placed within Financial Process Integrator code. It can have the value `true` or `false`, specifying whether the timing point is to be recorded or not.

timingPoint.custom.recording

timingPoint.custom.recording is set as follows:

```
timingPoint.custom.recording=true
```

where `timingPoint.custom.recording` is an alternative to subsystems and could be placed anywhere in code. It can have the value `true` or `false`, specifying whether the timing point is to be recorded or not.

timingPoint.host.recording

timingPoint.host.recording is set as follows:

```
timingPoint.host.recording=true
```

where `timingPoint.host.recording` is an alternative to subsystems and would be placed within host transaction code. It can have the value `true` or `false`, specifying whether the timing point is to be recorded or not.

timingPoint.servlet.recording

timingPoint.servlet.recording is set as follows:

```
timingPoint.servlet.recording=true
```

where `timingPoint.servlet.recording` is an alternative to subsystems and would be placed within servlet code. It can have the value `true` or `false`, specifying whether the timing point be recorded or not.

timingPoint.format

timingPoint.format is set as follows:

```
timingPoint.format=TIMING_POINT_ID;THREAD_ID;MAJOR_TYPE;SUBSYSTEM;
TYPE;USER;START_TIME;END_TIME;ELAPSED_TIME;REQUEST;RESPONSE
```

above is the format string representing how a timing point will be logged to console or file. Above are all the possible base values that can be arranged in any order as long as they are delimited by a semi-colon.

If upon instantiation of a `TimingPoint` object in the code an additional parameter has been added to be output with the Timing Point, for example, if one has a Timing Point constructed as follows with an additional string named `'TRACE_ID'`:

```
Object[] objects = new Object[]{TimingPointConstants.TIMING_POINT_SUBSYSTEM,
BankFrameLogConstants.MCA_SUBSYSTEM, TimingPointConstants.TIMING_POINT_TYPE,
"Request Router", "TRACE_ID", "1234"};
```

Then the `timingPoint.format` setting should include the `'TRACE_ID'` as follows:

```
timingPoint.format=TRACE_ID;TIMING_POINT_ID;THREAD_ID;MAJOR_TYPE;
SUBSYSTEM;TYPE;USER;START_TIME;END_TIME;ELAPSED_TIME;REQUEST;RESPONSE
```

NOTE: `TRACE_ID` can appear anywhere in the format string.

Mail Service

The Mail service allows an MCA Services based system to send e-mail messages to a specified administrator or end user over the Internet or Intranet. It uses Sun's `javax.mail` API to create and send e-mail messages and is implemented using a stateless session EJB. Note that the MCA mail service only sends e-mail.

Classes and Package Structure

The mail service is contained in the following package.

```
com.bankframe.services.mail
```

It consists of the following files:

<code>SendMailBean</code>	The Actual Mail Bean
<code>SendMail</code>	Remote Interface to the Mail Bean
<code>SendMailHome</code>	<code>SendMailBean</code> home interface
<code>Client</code>	Application to test mail bean functionality

Here is a rundown of the methods in the `SendMailBean` that can be invoked by a `MailBean` client.

<code>processDataPacket()</code>	Pulls out all the details from the <code>DataPacket</code> and passes them to <code>sendMail()</code>
<code>sendMail(String mailFrom, String[] addresses, String subject, String message)</code>	Sends the mail via the <code>javax.mail</code> API. This method takes a String message
<code>sendMail(String mailFrom, String[] addresses, String subject, StringBuffer message)</code>	Sends the mail via the <code>javax.mail</code> API. This method takes a StringBuffer message

<code>sendMail(String mailFrom, String[] addresses, String[] ccAddresses, String subject, String text, String content, String connTimeout, String debug)</code>	Sends the mail via the javax.mail API. This method takes a String message. This method has optional parameters for CC'd addresses, content type, connection timeout and enabling the javax.mail API debug mode
<code>sendMail(String mailFrom, String[] addresses, String[] ccAddresses, String subject, StringBuffer message, String content, String connTimeout, String debug)</code>	Sends the mail via the javax.mail API. This method takes a StringBuffer message. This method has optional parameters for CC'd addresses, content type, connection timeout and enabling the javax.mail API in debug mode

DataPacket Structure

In order for the `processDataPacket()` method in the mail service to work properly, the [DataPacket](#) passed as an argument must conform to the following structure:

NAME	Name of the DataPacket
REQUEST_ID	Request ID of the mail bean
FROM	String containing the sender of the mail
SUBJECT	String containing mail subject
MESSAGE	String containing mail message
ADDRESS_1-n	1-n number of addresses to send the mail to
NUMBER_OF_RECEIVERS	Number of receivers for the mail
CONNECTION_TIMEOUT	This is an optional parameter specifying the connection timeout period, in seconds, for sending the email.
CONTENT	This is an optional parameter specifying the content type of the e-mail message, for example, text/html
CC_ADDRESS_1-n	1 - n number of addresses to CC the mail to. This is an optional parameter
NUMBER_OF_CC_RECEIVERS	Number of CC receivers for the mail. This is an optional parameter
DEBUG	This is an optional parameter specifying that the javax.mail API operates in debug mode

The response [DataPacket](#) passed back to the client will be of the following form:

NAME	SENT MAIL
TO	String concatenated with all the addresses the mail was intended for.

SUBJECT	String containing mail subject
----------------	--------------------------------

Also note that in addition to deploying the mail bean on the server you must provide a name for an SMTP mail server using the property `mail.smtpServer` in the `BankframeResource.properties` file.

Using the Mail Service

In order to use the mail service, the client must communicate with the `EHHTPCommsManager` on the server and pass to it a `DataPacket` matching the structure discussed previously.

The following client example shows how to do this:

```
import java.util.Vector;

import com.bankframe.bo.DataPacket;

import com.bankframe.ei.channel.client.HttpClient

public class MailClient {

    public static void main(String [] args) {

        DataPacket dp = new DataPacket("SEND MAIL");
        dp.put("REQUEST_ID", "MC201");
        dp.put("FROM", "Administrator@eontec.com");
        dp.put("SUBJECT", "Test");
        dp.put("MESSAGE", "Testing Mail Bean");
        dp.put("ADDRESS_1", "User1@eontec.com");
        dp.put("ADDRESS_2", "User2@eontec.com");
        dp.put("ADDRESS_3", "User3@eontec.com");
        dp.put("NUMBER_OF_RECEIVERS", "3");
        dp.put("CC_ADDRESS_1", "User4@eontec.com");
        dp.put("NUMBER_OF_CC_RECEIVERS", "1");
        dp.put("CONNECTION_TIMEOUT", "10000");//10 seconds timeout

        HttpClient client = new HttpClient();
```

```

        Vector responses = client.send(dp);
    }
}

```

This client will return a Vector of response DataPackets, each one matching the structure discussed above.

Ping Utility

The Ping utility is used to confirm that an MCA Services installation is working and that the servlets on the web server are communicating with the MCA installation correctly. This utility should be used when setting up the environment. It is part of MCA and can be executed from a browser or from the command line.

When a request is sent to the Ping EJB, it will respond with a [DataPacket](#) that gives the time of the request and a message indicating that the deployment environment is live.

Classes and Package Structure

The ping service is contained in the following package.

```
com.bankframe.services.ping
```

It consists of the following files:

PingBean	The Actual Ping Bean.
Ping	Remote Interface to the Ping Bean.
PingHome	PingBean home interface.
Client	Application to test ping bean functionality.

DataPacket Structure

The [DataPacket](#) passed to the server must be supplied the [REQUEST_ID](#) of the Ping bean so the server can find it and route the [DataPacket](#) to it.

NAME	Name of the DataPacket
REQUEST_ID	Request ID of the Ping bean

The returned [DataPacket](#) will have the following fields.

NAME	PING RESULT
RESULT	String representing the result of the Ping

Using the Ping Service

Calling the Ping Service Using a Client

The following piece of client code will generate a `DataPacket` by supplying the `REQUEST_ID` and sending it to the server via the `EHTTCommsManager` for processing.

This client uses the URL `http://host name:portnumber` as an example http server.

NOTE: The `REQUEST_ID` of the Ping bean is usually "MC999", but verify this.

```
import java.util.Vector;

import com.bankframe.bo.DataPacket;

import com.bankframe.ei.comms.EHTTCommsManager;

public class PingClient {

    public static void main(String [] args) {

        DataPacket dp = new DataPacket("TEST PING");

        dp.put("REQUEST_ID", "MC999");

        EHTTCommsManager commsManager = new EHTTCommsManager("",
            "http://hostname:portnumber/BankframeServlet");

        Vector response = commsManager.sendDataPacket(dp);

        DataPacket data = (DataPacket) response.elementAt(0);

        System.out.println(data.getString("Result"));

    }

}
```

This client will result in the following being printed to the console:

```
Tue Nov 28 16:57:47 GMT 2000 EJB Server is t3://hostname:portnumber is alive
```

Calling the Ping Service Using a Browser

The Ping Service can also be called from a browser using the `MonitorServlet` to do this type in the following url:

```
http://hostname:portnumber/MonitorServlet
```

On the `MonitorServlet` screen input the `REQUEST_ID` of the Ping Service, usually 'MC999'. The Ping Result `DataPacket` fields will display, including the owner and Request_ID fields.

LDAP Connectivity

Lightweight Directory Access Protocol (LDAP) defines a standard protocol for accessing information stored in directory services. Typically, directory services are used for storing information such as User information, names and addresses, phone numbers, e-mail addresses and user ID's. Information in LDAP repositories is stored in a hierarchical structure. Each LDAP repository has a schema, which defines the types of objects that can be stored in the repository.

MCA Services and LDAP

In order to ease integration with customers' existing IT infrastructure MCA needs to be able to access information stored in LDAP repositories. MCA provides this connectivity through the [com.bankframe.ei.ldap](#) package. This package provides facilities for accessing LDAP repositories directly and for creating Bean Managed Entity beans that persist data to/from LDAP repositories.

com.bankframe.ei.ldap

The [com.bankframe.ei.ldap](#) package provides MCA's LDAP connectivity. The package contains the following classes/interfaces:

LDAPServerContext	Represents a connection to an LDAP server.
LDAPServerContextFactory	Creates and manages connections to the LDAP server.
LDAPEntityBean	Abstract class used for implementing BMP Entity beans that map attributes to data stored in the LDAP server.
LDAPPrimaryKey	Interface used to encapsulate the data that comprises the primary key of an LDAPEntityBean instance.
LDAPEntityBeanPK	Standard implementation of LDAPPrimaryKey

com.bankframe.ei.ldap.LDAPServerContext

This class provides the connectivity to an LDAP server. Connecting to the server requires several configuration parameters; these are defined as the following constant fields in this class:

PROVIDER_URL	URL of the ldap server
INITIAL_CONTEXT_FACTORY	The JNDI factory class to use to make the connection.
SECURITY_AUTHENTICATION	The authentication method.
SECURITY_PRINCIPAL	The user to authenticate.
SECURITY_CREDENTIALS	The password to use for authentication.
SECURITY_PROTOCOL	Specifies whether to connect using Secure Sockets Layer.

<code>BASE_DN</code>	Specifies the base distinguished name of this context.
<code>RDN_ATTRIBUTE</code>	Specifies the name of the attribute that is used to form the dn.
<code>DEFAULT_SEARCH_FILTER</code>	Specifies a default search filter to use for searches.
<code>CONTEXT_ALIAS</code>	Specifies the name of the alias that defines the above settings.

These parameters are passed to the constructor as key-value pairs in a `Hashtable`. All the parameters may not be required, for example the LDAP server may not require authentication, so the security parameters will not need to be specified.

NOTE: When an `LDAPServerContext` instance is created the physical connection to the server is not immediately established. It will only be created when it is required, that is, when one of its methods is invoked. The physical connection will be closed when the context is destroyed, it can also be closed explicitly by calling the `close()` method. The `open()` method can be used to explicitly establish the physical connection. See the JavaDocs for this class for more details of the methods it implements.

`com.bankframe.ei.ldap.LDAPServerContextFactory`

This class simplifies the task of creating correctly configured `LDAPServerContext` instances. It maps an alias to sets of `LDAPServerContext` configuration properties, which are stored in the `BankFrameResource.properties` configuration file. Instead of explicitly specifying all the configuration properties in order to create an `LDAPServerContext` instance, you can call the `LDAPServerContextFactory.getServerContext(String aliasName)` method, which will retrieve the settings from `BankFrameResource.properties` and create an `LDAPServerContext` with those settings. Here's an example set of configuration settings:

```
samplecontext.java.naming.factory.initial=com.sun.jndi.ldap.LdapCtxFactory
samplecontext.java.naming.security.authentication=simple
samplecontext.java.naming.security.principal=someUserId
samplecontext.java.naming.security.credentials=somePassword
samplecontext.java.naming.security.protocol=SSL
samplecontext.ldap.baseDn=ou=someOrganizationalUnit,o=someOrganization
samplecontext.ldap.rdnAttribute=cn
samplecontext.ldap.defaultSearchFilter=cn={0}
```

To retrieve these values and instantiate an `LDAPServerContext` with the above values you would do the following:

```
LDAPServerContext ctx =
LDAPServerContextFactory.getServerContext("samplecontext");
```

When you are finished using the `LDAPServerContext` instance you should release it as follows:

```
LDAPServerContextFactory.releaseServerContext(ctx);
```

`LDAPServerContextFactory` caches `LDAPServerContexts`. The first time a request is made for a specific `LDAPServerContext`, the context is instantiated, and a reference to the instance is cached. If a second request is made for the same context, then the reference to the existing context is passed back, rather than creating another instance of the same context.

Configuring LDAP Caching

To specify whether the server context is cached or not, configure the following setting:

```
ldap.context.cache=
```

Set the value to `true` to enable caching.

Set the value to `false` to disable caching.

`com.bankframe.ei.ldap.LDAPEntityBean`

While it is possible to access data in LDAP repositories directly using the only the methods in `LDAPServerContext`, it is recommended that a Bean Managed Entity Bean is developed to wrap any data that needs to be accessed in the LDAP repository. This has a number of benefits:

- Scalability, since the Data Access is being managed via an EJB, the application server can manage and share bean instances
- Reusability, The bean can be changed to Container managed or to some other Bean Managed implementation, without affecting the business logic that uses the bean.
- Consistency, The bean will be consistent with the MCA Architecture where data is represented as Entity Beans.

The `LDAPEntityBean` class simplifies the process of creating an LDAP based BMP Entity Bean. It takes care of writing and reading data to/from the LDAP repository. It provides standard implementations of all the methods required by the EJB specification including standard `ejbFindByPrimaryKey()` and `ejbFindAll()` implementations. `LDAPEntityBean` extends the `com.bankframe.ejb.EntityBean` class, therefore `LDAPEntityBean` subclasses can be treated the same as any other MCA Entity Bean.

`com.bankframe.ei.ldap.LDAPPrimaryKey`

The `LDAPPrimaryKey` interface defines the methods that `LDAPEntityBean` expects Primary key classes to implement. There follows a sample implementation as an `LDAPEntityBean` based EJB

```
// Get the value of the relative distinguished name attribute

public String getRdnAttributeValue() ;

// set the value of the relative distinguished name attribute

public void setRdnAttributeValue(String value) ;

// required by the EJB 1.1 specification

public boolean equals(java.lang.Object o) ;

// required by the EJB 1.1 specification

public int hashCode() ;
```

In LDAP terminology the relative distinguished name is the name that uniquely identifies an object. It is always of the form: `attribute-name=attribute-value`, where `attribute-name` is the name of one of the attributes in the object. The `rdn` is equivalent to a primary key.

com.bankframe.ei.ldap.LDAPEntityBeanPK

`LDAPEntityBeanPK` is a standard implementation of the `LDAPPrimaryKey` class. It can be used as the primary key class for most LDAP based entity beans.

Sample Bean Managed LDAP based Entity Bean

If you use the `LDAPEntityBean` class to write an LDAP based Entity Bean it takes care of all the EJB implementation code. It provides fully functional implementations of the `ejbLoad()`, `ejbStore()`, `ejbActivate()`, `ejbPassivate()`, `ejbRemove()`, `ejbFindByPrimaryKey()`, `ejbFindAll()`, and `toDataPacket()` methods. Classes that extend `LDAPEntityBean` only need to provide `ejbCreate()`, `ejbPostCreate()`, and attribute access methods and `finder()` methods.

This sample implementation defines an Entity Bean that wraps the standard LDAP Person objectclass: The Person objectclass has the following attributes:

Attributes	<code>cn</code> <code>sn</code>
Optional Attributes	<code>userPassword</code> <code>telephoneNumber</code> <code>seeAlso</code> <code>description</code>

Bean Implementation

Here's a bean implementation to wrap the above attributes:

```
import java.rmi.RemoteException;

import javax.ejb.CreateException;

import com.bankframe.ei.ldap.LDAPEntityBean;

public class LDAPPersonBean extends LDAPEntityBean {

    private final String ATTRIBUTE_COMMON_NAME="cn";

    private final String ATTRIBUTE_SURNAME="sn";

    private final String ATTRIBUTE_PASSWORD="userPassword";

    private final String ATTRIBUTE_PHONE_NUMBER="telephoneNumber";
```



```

private final String ATTRIBUTE_SEE_ALSO="seeAlso";

private final String ATTRIBUTE_DESCRIPTION="description";

private final String OBJECT_CLASS="person";


// ejb creation method

public LDAPEntityBeanPK ejbCreate(String commonName,String surName,String
password,String phoneNumber,String seeAlso,String description) throws
CreateException {

    this.putObjectClass(this.OBJECT_CLASS); // set the object class of this object

    // set the attributes of this object

    this.put(this.ATTRIBUTE_COMMON_NAME,commonName);

    this.put(this.ATTRIBUTE_SURNAME,surName);

    this.put(this.ATTRIBUTE_PASSWORD,password);

    this.put(this.ATTRIBUTE_PHONE_NUMBER,phoneNumber);

    this.put(this.ATTRIBUTE_SEE_ALSO,seeAlso);

    this.put(this.ATTRIBUTE_DESCRIPTION,description);

    // initialize

    return super.ejbCreate();

}

// required method

public void ejbPostCreate(String commonName,String surName,String
password,String phoneNumber,String seeAlso,String description) {

}

// return the name of the attribute that is used as to form the rdn

public String getRdnAttributeName() {

    return this.ATTRIBUTE_COMMON_NAME;

}

// EntityBean attribute getter

public String getCommonName() {

    return this.get(this.ATTRIBUTE_COMMON_NAME);

```

```

    }

    public String getSurName() {

        return this.get(this.ATTRIBUTE_SURNAME);

    }

    public String getPassword() {

        return this.get(this.ATTRIBUTE_PASSWORD);

    }

    public String getPhoneNumber() {

        return this.get(this.ATTRIBUTE_PHONE_NUMBER);

    }

    public String getSeeAlso() {

        return this.get(this.ATTRIBUTE_SEE_ALSO);

    }

    public String getDescription() {

        return this.get(this.ATTRIBUTE_DESCRIPTION);

    }

    public void setDescription(String description) {

        this.put(this.ATTRIBUTE_DESCRIPTION,description);

    }

}

```

Bean Implementation Explained

As can be seen from the example above the bean implementation only needs to do a few things to be able to access the data stored in the LDAP repository:

Specify the ldap objectclass

This is done in the `ejbCreate()` method using the following method call:

```
this.putObjectClass(this.OBJECT_CLASS);
```

This tells `LDAPEntityBean` what the LDAP objectclass is, so that `LDAPEntityBean` can create the correct type of object in the LDAP repository.

Specify the ldap attributes

This is also done in the `ejbCreate()` method by calling the `LDAPEntityBean.put()` method. The `put` method takes two parameters, the name of the attribute and the value of the attribute. The value can be any simple Java type such as `String`, `Long` or `Double`. For example the 'common name' attribute is set using the following method call:

```
this.put(this.ATTRIBUTE_COMMON_NAME,commonName);
```

Create the Primary Key instance

The EJB Specification requires that all Bean Managed Entity Beans' `ejbCreate()` methods return an instance of the Primary Key class. `LDAPEntityBean` provides a standard `ldapCreate()` method that creates an initialized instance of `LDAPEntityBeanPK`.

Specify the Rdn Attribute Name

In order for `LDAPEntityBean` to be able to manage primary keys, it must know which attribute in the object is used to form the relative distinguished name. In the case of the `Person` object, this is the 'cn' attribute. This is done using the following code:

```
public String getRdnAttributeName() {

    return this.ATTRIBUTE_COMMON_NAME;

}
```

Implementing getter methods

To provide read access to the Entity Bean's attributes, 'getter' methods must be implemented, for example:

```
public String getDescription() {

    return this.get(this.ATTRIBUTE_DESCRIPTION);

}
```

This method uses the `LDAPEntityBean.get()` method to retrieve the current value of the description attribute.

Implementing setter methods

To enable the value of entity bean attributes to be changed, 'setter' methods must be provided, for example:

```
public void setDescription(String description) {

    this.put(this.ATTRIBUTE_DESCRIPTION,description);

}
```

Implementing `ejbFindByPrimaryKey()` and `ejbFindAll()`

All entity beans must provide an `ejbFindByPrimaryKey()` method. The `ejbFindByPrimaryKey()` method in the above example wraps the `LDAPEntityBean.ldapFindByPrimaryKey()` method casting the returned primary key instance to the correct type. Entity beans can optionally provide custom finder methods, one such common method is an `ejbFindAll()` method. `LDAPEntityBean` provides a method: `ldapFindAll()` that retrieves all entries in the current ldap context. The `ejbFindAll()` method in the above example uses `ldapFindAll()` passing it the primary key class to use to uniquely identify each entry.

The Remote Interface

The Remote Interface for an LDAP based Entity bean is defined in exactly the same manner as any other Siebel Retail Finance Entity Bean. The interface should extend the `com.bankframe.EEntityRemote` interface and define the methods used to access the entity bean's attributes. The Remote Interface for the example above would be:

```
import java.rmi.RemoteException;

import com.bankframe.ejb.EEntityRemote;

public interface LDAPPerson extends EEntityRemote {

    public String getCommonName() throws RemoteException;

    public String getSurName() throws RemoteException;

    public String getPassword() throws RemoteException;

    public String getPhoneNumber() throws RemoteException;

    public String getSeeAlso() throws RemoteException;

    public String getDescription() throws RemoteException;

}
```

The Home Interface

The home interface is also defined in the same manner as other Siebel Retail Finance Entity Beans:

```
import java.rmi.RemoteException;

import javax.ejb.CreateException;

import javax.ejb.FinderException;

import javax.ejb.EJBHome;

import com.bankframe.ei.ldap.LDAPEntityBeanPK;
```

```

public interface LDAPPersonHome extends EJBHome {

    public LDAPEntityBeanPK create(String commonName,String surName,String
password,String phoneNumber,String seeAlso,String description) throws
CreateException, RemoteException ;

    public LDAPEntityBeanPK findByPrimaryKey() throws
FinderException,RemoteException;

    public Enumeration findAll() throws FinderException,RemoteException;

}

```

LDAP Connectivity Advanced Topics

Using Custom Primary Keys

In some cases it may not be possible or desirable to use the `com.bankframe.ei.ldap.LDAPEntityBeanPK` class as the primary key class for an LDAP Entity Bean. In these cases a custom Primary Key class needs to be developed that implements the `LDAPPrimaryKey` interface. The following sample implementation uses a custom primary key class called `CustomPK`.

Class Definition:

```

import com.bankframe.ei.ldap.LDAPPrimaryKey;

public class CustomPK implements LDAPPrimaryKey {

    public String commonName;

    public CustomPK() {}

    public CustomPK(String commonName) { this.commonName = commonName ;}

    public String getRdnAttributeValue() { return this.commonName; }

    public void setRdnAttributeValue(String value) { this.commonName = value;}

    public boolean equals(java.lang.Object o) {

        if (o instanceof CustomPK) {

            CustomPK otherKey = (CustomPK) o;

            return ((this.commonName.equalsIgnoreCase(otherKey.commonName)));

        } else {

            return false;

        }

    }

}

```

```

    }

    public int hashCode() { return commonName.hashCode(); }

}

```

Modifying the LDAPPerson Example to Use CustomPK

Change the ejbCreate() method

The `ejbCreate()` method must return an instance of the primary key class, that is, `CustomPK`. The `LDAPPerson.ejbCreate()` methods need to be changed as follows:

```

    public CustomPK ejbCreate(...parameters as before...) {

        ... configure objectclass and attributes as before...

        super.ejbCreate();

        return new CustomPK((String)this.get(this.ATTRIBUTE_COMMON_NAME));

    }

```

To change the ejbCreate method

- 1 Change the return type of the `ejbCreate()` method to `CustomPK`.
- 2 Call `super.ejbCreate()` (to initialize the bean) but do not return the primary key it creates.
- 3 Create a `CustomPK()` instance, initializing it with the current value of the `commonName` attribute.

Define type correct ejbFindByPrimaryKey() method

An `ejbFindByPrimaryKey()` method is required that has a return type of `CustomPK`. `LDAPEntityBean` and has a protected method `LDAPPrimaryKey (LDAPPrimarykey primaryKey)`. This method can be overridden to implement a type correct `ejbFindByPrimaryKey()` as follows:

```

    CustomPK ejbFindByPrimaryKey(CustomPK primaryKey) throws FinderException {

        return (CustomPK)super.ldapFindByPrimaryKey(primaryKey);

    }

```

Define type correct ejbFindAll() method

An `ejbFindAll()` method is required that creates instances of the `CustomPK`. `LDAPEntityBean` and has a protected method: `Enumeration ejbFindAll(Class primaryKeyClass)`. This method can be used to create an enumeration of instances of the specified primary key class as follows:

```

    Enumeration ejbFindAll() throws FinderException {

        return super.ejbFindAll(CustomPK.class);

    }

```

Modify the `LDAPPersonHome.findByPrimaryKey()` method

The primary key type for the `LDAPPersonHome.findByPrimaryKey()` method needs to be changed to `CustomPK`:

```
CustomPK findByPrimaryKey(CustomPK primaryKey) throws
FinderException, RemoteException;
```

Modify the Deployment Descriptor

The `primaryKeyClassName` field in the deployment descriptor should be changed to: `CustomPK`

Handling Multiple Values

Some LDAP attributes can have multiple values. `LDAPEntityBean` provides two methods for accessing these kinds of attributes: `LDAPEntityBean.getMultiple()` and `LDAPEntityBean.putMultiple()`. `getMultiple()` retrieves the values of the specified attribute and returns them as an `Enumeration`. `putMultiple()` takes the name of the attribute, and an `Enumeration` of values to store.

Implementing Custom Finder Methods

In some cases the `findByPrimaryKey()` and `findAll()` methods will not be sufficient. Custom EJB finder methods can be implemented as follows:

Define the method in the implementation bean

This builds on the example above and defines a custom finder called `ejbFindBySurName()`:

```
Enumeration ejbFindBySurname(String surname) throws FinderException {
    try {
        LDAPServerContext ctx = this.getServerContext();

        String[] filterArgs = new String[1];

        FilterArgs[0] = surname;

        NamingEnumeration enum = ctx.search("sn={0}", filterArgs);

        Vector v = new Vector();

        While ( enum.hasMore() ) {
            SearchResult res = (SearchResult)enum.next();

            String surname =
                (String)res.getAttributes().get(this.ATTRIBUTE_SURNAME).get();

            v.addElement( new CustomPK(surname));
        }

        this.releaseServerContext(ctx);
    }
}
```

```

        return v.elements();

    } catch ( Exception ex ) {

        throw new FinderException(ex.toString());

    }

}

```

`LDAPEntityBean` contains a protected method `getServerContext()`, which returns a reference to the current LDAP connection. The `LDAPServerContext.search()` method is then used to find all entries with the specified surname. The `search()` method returns an `Enumeration`, which is iterated through, creating Primary key instances for each result. Finally an `Enumeration` of these primary key instances is returned.

Add the corresponding method to the `LDAPPersonHome` interface

The second and final step is to add the corresponding finder method in the home interface:

```
Enumeration findBySurname(String surname) throws FinderException,RemoteException;
```

Data Validation

During the execution of Financial Components, certain data types need to be formatted, validated, or converted to another data type. The functionality to do this is provided within a number of classes in MCA Services.

Classes and Package Structure

The validation and data conversion classes are implemented in the package `com.bankframe.validation`. This package contains the following classes:

```

com.bankframe.validation.ValidationException

com.bankframe.validation.DataTypeValidator

com.bankframe.validation.DataTypeConvertor

com.bankframe.validation.DateValidator

com.bankframe.validation.DateConvertor

```

`com.bankframe.validation.ValidationException`

This exception is thrown whenever a validation error occurs. This class replaces the `com.bankframe.ejb.ValidationException` class. This class extends the `com.bankframe.EonException` class. This class has the following `public` methods:

<code>ValidationException(int</code>	Create a validation exception identified by the
--------------------------------------	---

<code>errorNumber)</code>	specified <code>errorNumber</code>
<code>ValidationException(int errorNumber,String[] params)</code>	Create a validation exception identified by the specified <code>errorNumber</code> and with the arguments specified by the <code>params[]</code> array
<code>ValidationException(int errorNumber,Locale locale)</code>	Create a validation exception identified by the specified <code>errorNumber</code> , using the specified <code>Locale</code> to localise the error message
<code>ValidationException(int errorNumber,String[] params,Locale locale)</code>	Create a validation exception identified by the specified <code>errorNumber</code> and with the arguments specified by the <code>params[]</code> array, using the specified <code>Locale</code> to localize the error message
<code>DataPacket toDataPacket()</code>	Convert the exception to a <code>DataPacket</code>

com.bankframe.DataTypeValidator

This class contains useful methods that can be used to validate various data-types. This class contains the following `public static` methods:

<code>boolean isDigitsOnly(String value)</code>	This method returns <code>true</code> if the specified <code>String</code> contains only digits.
<code>isExactLength(String value,int length)</code>	This method returns <code>true</code> if the specified <code>String</code> is exactly the specified length.
<code>isLengthLessThanOrEqualTo(String value, int maxLength)</code>	This method returns <code>true</code> if the specified <code>String</code> is less than or equal to the specified length.
<code>boolean isLetterOrDigitsOnly(String value)</code>	This method returns <code>true</code> if the specified <code>String</code> contains only letters or digits
<code>boolean isLettersOnly(String value)</code>	This method returns <code>true</code> if the specified <code>String</code> contains only letters.
<code>boolean isNullOrEmpty(Object value)</code>	This method returns true if the specified value is null, or empty, or contains the value: <code>'null'</code>

com.bankframe.DataTypeConvertor

This class contains useful methods that can be used for converting data from one type to another. This class contains the following `public static` methods:

<code>Boolean getBoolean(String value) throws ValidationException</code>	This method converts a <code>String</code> value to a <code>Boolean</code> value. It throws a <code>ValidationException</code> if the <code>String</code> cannot be converted to a <code>Boolean</code> value
--	---

<code>Double getDouble(String value)</code> throws <code>ValidationException</code>	This method converts a <code>String</code> value to a <code>Double</code> value. It throws a <code>ValidationException</code> if the <code>String</code> cannot be converted to a <code>Double</code> value
<code>Float getFloat(String value)</code> throws <code>ValidationException</code>	This method converts a <code>String</code> value to a <code>Float</code> value. It throws a <code>ValidationException</code> if the <code>String</code> cannot be converted to a <code>Float</code> value
<code>Integer getInteger(String value)</code> throws <code>ValidationException</code>	This method converts a <code>String</code> value to an <code>Integer</code> value. It throws a <code>ValidationException</code> if the <code>String</code> cannot be converted to an <code>Integer</code> value
<code>String getString(Object value)</code> throws <code>ValidationException</code>	This method converts an <code>Object</code> value to a <code>String</code> value. It throws a <code>ValidationException</code> if the <code>Object</code> cannot be converted to a <code>String</code> value
<code>String padString(String value, char padChar, int length, boolean padRight)</code> throws <code>ValidationException</code>	This method returns a <code>String</code> padded with the specified amount of padding characters. The <code>String</code> can be padded to the left or to the right. This method throws a <code>ValidationException</code> if the value is null or too long to be padded
<code>Double round(Double value, int decimalPlaces)</code>	This method rounds up the specified value to the specified number of decimal places
<code>Double round(Double value, int decimalPlaces, int roundMethod)</code>	This method rounds the specified value to the specified number of decimal place using the specified rounding method. See the Java API documentation of <code>java.lang.BigDecimal</code> for information on rounding methods.

com.bankframe.validation.DateValidator

This class contains useful methods for validating dates, times and timestamps. This class contains the following `public static` methods:

<code>int compare(Date dateOrTime1, Date dateOrTime2)</code> throws <code>ValidationException</code>	This method compares two <code>Date</code> objects it returns an <code>int</code> value, <code>DateValidator.EQUALS</code> if the argument is a <code>Date</code> equal to this <code>Date</code> ; <code>DateValidator.AFTER</code> if the argument is a <code>Date</code> after this <code>Date</code> ; <code>DateValidator.BEFORE</code> if the argument is a <code>Date</code> before this <code>Date</code> . It throws a <code>ValidationException</code> if the date/time inputs are null or empty.
<code>int compare(Date date1, Date date2)</code> throws	This method compares two <code>Date</code> objects ignoring the hours minutes and seconds portion of the <code>Date</code> object. It returns an <code>int</code>

<code>ValidationException</code>	value, <code>DateValidator.EQUALS</code> if the argument is a <code>Date</code> equal to this <code>Date</code> ; <code>DateValidator.AFTER</code> if the argument is a <code>Date</code> after this <code>Date</code> ; <code>DateValidator.BEFORE</code> if the argument is a <code>Date</code> before this <code>Date</code> . It throws a <code>ValidationException</code> if the date/time inputs are null or empty.
<code>int compare(Time time1, Time time2) throws ValidationException</code>	This method compares two <code>Time</code> objects ignoring the day month and year portion of the <code>Time</code> object. It returns an <code>int</code> value, <code>DateValidator.EQUALS</code> if the argument is a <code>Time</code> equal to this <code>Time</code> ; <code>DateValidator.AFTER</code> if the argument is a <code>Time</code> after this <code>Time</code> ; <code>DateValidator.BEFORE</code> if the argument is a <code>Time</code> before this <code>Time</code> . It throws a <code>ValidationException</code> if the date/time inputs are null or empty.
<code>boolean isValid(String pattern, String dateOrTime) throws ValidationException</code>	This method compares a date/time string with a <code>SimpleDateFormat</code> pattern to ensure that it is valid. It throws a <code>ValidationException</code> if the pattern or the date/time inputs are null or empty.

com.bankframe.validation.DateConvertor

This class contains methods that can be used to convert Strings to `Date`, `Time` or `Timestamp` objects and vice versa. This class contains the following `public static` methods:

<code>Date getDate(String pattern, String date) throws ValidationException</code>	This method uses the <code>SimpleDateFormat</code> class to convert a <code>String</code> to a <code>Date</code> Object. It throws a <code>ValidationException</code> if the pattern or the date inputs are null or empty and if the date is invalid.
<code>Time getTime(String pattern, String time) throws ValidationException</code>	This method uses the <code>SimpleDateFormat</code> class to convert a <code>String</code> to a <code>Date</code> Object and then gets a <code>Time</code> object from the <code>Date</code> . It throws a <code>ValidationException</code> if the pattern or the time inputs are null/empty or if the time is invalid.
<code>Timestamp getTimestamp (String pattern, String timestamp) throws ValidationException</code>	This method uses the <code>SimpleDateFormat</code> class to convert a <code>String</code> to a <code>Date</code> Object and then gets a <code>Timestamp</code> object from the <code>Date</code> . It throws a <code>ValidationException</code> if the pattern or the timestamp inputs are null/empty or if the time is invalid.
<code>String getString(String pattern, Date dateOrTime) throws</code>	This method uses a <code>SimpleDateFormat</code> pattern to convert a <code>Date</code> object into a <code>String</code> . It throws a <code>ValidationException</code> if

<code>ValidationException</code>	the pattern or the date/time inputs are null or empty.
----------------------------------	--

Data Validator Sample Implementations

DataTypeValidator Example

Below is some sample code that illustrates how to use the `DataTypeValidator` class:

```
public class TestDataTypeValidator {

    public static void main(String[] args) {

        String value1 = "345123";

        String value2 = "Hello World";

        boolean result = DataTypeValidator.isDigitsOnly(value1);

        // result will be true

        result = DataTypeValidator.isDigitsOnly(value2);

        // result will be false

        result = DataTypeValidator.isExactLength(value1,6);

        // result will be true

        result = DataTypeValidator.isExactLength(value1,7);

        // result will be false

        result = DataTypeValidator.isLengthLessThanOrEqualTo(value1,7);

        // result will be true

        result = DataTypeValidator.isLengthLessThanOrEqualTo(value1,5);

        // result will be false

        String nullReference = null;

        String emptyString = "";

        String nullString = "null";

        result = DataTypeValidator.isNullOrEmpty(value1);

        // result will be false

        result = DataTypeValidator.isNullOrEmpty(nullReference);

        // result will be true
    }
}
```

```

    result = DataTypeValidator.IsNullOrEmpty(emptyString);

    // result will be true

    result = DataTypeValidator.IsNullOrEmpty(nullString);

    // result will be true
}
}

```

DataTypeConvertor Example

Below is some sample code that illustrates how to use the [DataTypeConvertor](#) class:

```

public class TestDataTypeConvertor {

    public static void main(String[] args) {

        try {

            Boolean booleanValue = DataTypeConvertor.getBoolean("True");

            // booleanValue will be true

            booleanValue = DataTypeConvertor.getBoolean("FALSE");

            // booleanValue will be false (Note case of String is unimportant)

            booleanValue = DataTypeConvertor.getBoolean("yes");

            // booleanValue will be true, (getBoolean() treats 'yes' as true and 'no'
as false)

            booleanValue = DataTypeConvertor.getBoolean("No");

            // booleanValue will be false, (getBoolean() treats 'yes' as true and 'no'
as false)

            Double doubleValue = DataTypeConvertor.getDouble("2.3123");

            // double value will be 2.3123

            Integer integerValue = DataTypeConvertor.getInteger("1000");

            // integerValue will be 1000

            String stringValue = DataTypeConvertor.getString(integerValue);

            // stringValue will be '1000'

            String paddedString = DataTypeConvertor.padString(stringValue, '0', 8, false);

            // paddedString will be '00001000'

```

```

        Double roundedValue = DataTypeConvertor.round(new Double(2.0/3.3),3);

        // roundedValue will be 0.607

        roundedValue = DataTypeConvertor.round(new
Double(2.0/3.3),3,DataTypeConvertor.ROUND_DOWN);

        // roundedValue will be 0.606
    } catch ( ValidationException vex) {

        vex.printStackTrace();

    }

}

}

```

DateValidator Example

```

public class TestDateValidator {

    public static void main(String[] args) {

        try {

            Date date1 = DateConvertor.getDate("dd/MM/yyyy HH:mm:ss", "26/03/2001
14:00:51");

            //this creates the following date object: Mon Mar 26 14:00:51 GMT 2001

            Date date2 = DateConvertor.getDate("dd/MM/yyyy", "26/02/2001");

            //this creates the following date object: Mon Feb 26 00:00:00 GMT 2001;

            Date date3 = DateConvertor.getDate("dd/MM/yyyy hh:mm:ss", "26/03/2001
12:05:00");

            //this creates the following date object: Mon Mar 26 00:05:00 GMT 2001;

            Date date4 = DateConvertor.getDate("dd/MM/yyyy HH:mm:ss", "26/04/2001
16:30:05");

            //this creates the following date object: Thu Apr 26 16:30:05 GMT 2001;

            Time time1 = DateConvertor.getTime("HH:mm:ss", "13:56:01");

            //this creates the following date object: 13:56:01;

            Time time2 = DateConvertor.getTime("HH:mm:ss", "11:20:01");

            //this creates the following date object: 11:20:01;

            Time time3 = DateConvertor.getTime("HH:mm:ss", "13:56:01");

```

```

//this creates the following date object: 13:56:01;
Time time4 = DateConvertor.getTime("HH:mm:ss", "22:30:05");

//this creates the following date object: 22:30:05;
int result = DateValidator.compare(date1, date2);
// result will be DateValidator.AFTER
result = DateValidator.compare(date1, date3);
// result will be DateValidator.AFTER dates not equal because of time
result = DateValidator.compare(date1, date4);
// result will be DateValidator.BEFORE
result = DateValidator.compareDateOnly(date1, date2);
// result will be DateValidator.AFTER
result = DateValidator.compareDateOnly(date1, date3);
// result will be DateValidator.EQUALS as without time element dates are
equal
result = DateValidator.compareDateOnly(date1, date4);
// result will be DateValidator.BEFORE
result = DateValidator.compareTimeOnly(time1, time2);
// result will be DateValidator.AFTER
result = DateValidator.compareTimeOnly(time1, time3);
// result will be DateValidator.EQUALS
result = DateValidator.compareTimeOnly(time1, time4);
// result will be DateValidator.BEFORE
}

catch (ValidationException vex) {
    System.out.println(vex);
}

String date = "26/03/2001";
String time = "22:25:23";
String timestamp = "20/06/2001 22:25:23";

```

```

String pattern1 = "dd/MM/yyyy";

String pattern2 = "dd/MMM/yyyy";

String pattern3 = "hh/mm/ss";

String pattern4 = "HH/mm/ss";

String pattern5 = "dd/MM/yyyy hh/mm/ss ";

try {

    boolean reponse = DateValidator.isValid(pattern1, date);

    // result will be true

    reponse = DateValidator.isValid(pattern2, date);

    // result will be false

    reponse = DateValidator.isValid(pattern3, time);

    // result will be false

    reponse = DateValidator.isValid(pattern4, time);

    // result will be true

    reponse = DateValidator.isValid(pattern5, timestamp);

    // result will be true

    reponse = DateValidator.isValid(pattern1, timestamp);

    // result will be false

}

catch (ValidationException vex) {

    System.out.println(vex);

}

}

}

```

DateConvertor Example

```

public class TestDateConvertor {

    public static void main(String[] args) {

        String date1 = "23/03/2001";

        String time1 = "22:25:23";
    }
}

```



```

String timestamp1 = "20/06/2001 22:25:23";

String pattern1 = "dd/MM/yyyy";

String pattern2 = "HH:mm:ss";

String pattern3 = "dd/MM/yyyy HH:mm:ss";

try {

    Date result = DateConvertor.getDate(pattern1, date1);

    // result will be Fri Mar 23 00:00:00 GMT 2001

    result = DateConvertor.getTime(pattern2, time1);

    // result will be 22:25:23

    result = DateConvertor.getTimestamp(pattern3, timestamp1);

    // result will be 20-06-2001 22:25:23.0

}

catch (ValidationException vex) {

    System.out.println(vex);

}

String pattern4 = "dd/MM/yyyy";

String pattern5 = "hh:mm:ss";

String pattern6 = "dd/MM/yyyy hh:mm:ss ";

try {

    Date date2 = DateConvertor.getDate("dd/MM/yyyy", "23/03/2001");

    //creates the following date object Fri Mar 23 00:00:00 GMT 2001

    Time time2 = DateConvertor.getTime("HH:mm:ss", "22:25:23");

    //creates the following time object 22:25:23

    Timestamp timestamp2 = DateConvertor.getTimestamp("dd/MM/yyyy HH:mm:ss",
"26/04/2001 16:30:05");

    //creates the following timestamp object 20-06-2001 22:25:23.0";

    String response = DateConvertor.getString(pattern4, date2);

    // result will be 23/03/2001

    response = DateConvertor.getString(pattern5, time2);

```

```

        // result will be 10:25:23

        response = DateConvertor.getString (pattern6, timestamp2);

        // result will be 20/06/2001 22:25:23
    }

    catch (ValidationException vex2) {

        System.out.println(vex2);

    }

}
}

```

Peripherals Support

This topic describes the peripheral device support framework built into MCA Services. This support allows the user to use peripheral devices connected to the system. The architecture of the MCA device support allows the addition of support for new types of peripherals if required.

MCA currently has implementations for three types of peripheral devices:

- The MagTek MiniMicr cheque reader.
- The MagTek IntelliPIN swipe-card reader.
- The Epson TMU375 slip printer.

These implementations allow the user to control these devices at a basic level. They do not contain any business logic such as calculating cheque amount totals or swipe card amounts. The device implementations allow the user access to the raw information processed by the devices.

MagTek MiniMicr cheque reader

The MCA implementation for this peripheral allows the developer to:

- Setup the connection to the peripheral
- Prompt the user to swipe a cheque
- Read the raw cheque details. The details are read from the foot of the cheque by the MiniMicr peripheral and MCA returns the raw data to the user for further processing.

MagTek IntelliPIN swipe-card reader

The MCA implementation for this peripheral implements a basic subset of the MagTek IntelliPIN Pad functionality. This subset allows the developer to:

- Setup the device in interactive mode (PC controlled) with a Master encryption key (used to encrypt and decrypt pin data passed to the PC from the physical device.)

- Prompt for a card swipe and read the card track details from the physical device.
- Decrypt the pin data returned by the physical device when a user enters a pin number.
- Display string messages and modify the default messages displayed on the physical MagTek IntelliPIN device LCD display.

It is up to the developer to process the card track details and pin number information and validate the details.

Epson TM-U375 slip printer

The MCA implementation for this peripheral allows the user to

- Print text to the printer
- Perform basic printing operations such as line-feed and carriage-return. MCA does not have business logic for creating receipt information for printing. It is up to the developer to write the business logic to create specific types of receipts which are then passed to MCA for printing on the peripheral.

Adding new types of peripherals to MCA Services

New types of peripherals can be supported by MCA by extending the classes in MCA. This involves coding and subclassing of the appropriate classes and is not a plug-in mechanism.

Currently MCA has a general Serial-Port implementation which can be subclassed for any peripheral connected to the serial port (other types of connections will be supported in the future.)

The serial-port support in MCA encapsulates the Java Communications Extension API.

MCA Device Base Classes

MCA has a set of base classes for supporting peripherals. These classes can be subclassed to support new types of peripherals. Currently MCA has base classes for supporting peripherals connected to the serial-port. Support for any peripheral device connected to the serial-port can be added to MCA by subclassing these classes. General base classes for peripherals connected to the system by other means will be added in the future.

All the base classes for device support in MCA are contained in the package `com.bankframe.services.devices`. All implemented classes for specific device types are contained as sub-packages of this package.

MCADevice base interface

Every type of device object in the MCA framework must implement the `com.bankframe.services.devices.MCADevice` interface. This defines the basic set of commands that an MCA device must implement.

MCASerialPort base class

The basic class for serial port communication is the abstract class `com.bankframe.services.devices.MCASerialPort`. This class implements the

`com.bankframe.services.devices.MCADevice` interface and manages the connection to, communication with and initialization of a serial port. This class encapsulates the Java Communications Extension API.

Classes subclassing this base class can transmit and receive information on the serial port.

MCASerialPort.InputReaderThread class

The `MCASerialPort` class implements a thread

```
com.bankframe.services.devices.MCASerialPort.InputReaderThread
```

to asynchronously detect serial port events. `MCASerialPort.InputReaderThread` implements the interface `javax.comm.SerialPortEventListener` in the Java Communications Extension API.

Therefore when an event occurs on the serial port the method

`InputReaderThread.serialEvent(javax.comm.SerialPortEvent event)` is called by the Java Comm API.

MCASerialPort.handleEvent(...)

The `MCASerialPort.InputReaderThread` class always calls the method

`protected void handleEvent(java.util.EventObject theEvent)` which is defined in `com.bankframe.services.devices.MCASerialPort`. This method is over-ridden to customize the handling of serial-port device events. This method processes data from the physical device asynchronously because it is called by `MCASerialPort.InputReaderThread`. The method `handleEvent(java.util.EventObject theEvent)` calls the method `dataAvailable(Object data)` to store the received data for retrieval by the user. The method `waitForDataAvailable()` retrieves this data when called by the user.

If a class subclasses `MCASerialPort` then its implementation of

`handleEvent(java.util.EventObject theEvent)` parses and validates the data received and stores the result using `dataAvailable(Object data)`, thereby making more specific information available for the user.

Any exceptions that occur in `handleEvent(java.util.EventObject theEvent)` should be stored as `com.bankframe.services.devices.DeviceException` using `dataAvailable(Object data)`. This allows the user to retrieve any exceptions that might have occurred during parsing of the data from the physical device.

MCASerialPort.waitForDataAvailable()

After a user has instantiated an MCA serial device object the user can query the serial device object for available data. The method:

`public Object waitForDataAvailable(int timeOut)` waits the specified timeout period for data to be received by the serial device object from the peripheral device connected to the serial port.

MCASerialPort read() and write() and available()

`com.bankframe.services.devices.MCASerialPort` wraps the standard serial-port `read()`, `write()` and `available()` methods for interacting directly with the serial port.

■ `read(byte[] bytes, int off, int len)` and `read()` reads data from the serial port.

- `write(byte[] bytes, int off, int len)` and `write(int theByte)` write data from the serial port.
- `available()` determines the number of bytes that can be read from the serial port without blocking the program.

MCASerialPort.open() and MCASerialPort.setup()

After a user has instantiated an MCA serial device object the `open()` method is called to setup the device for use by the user. In the case of MCA serial port devices this method always calls the method:

```
abstract protected void setup()
```

If `MCASerialPort` is subclassed the method `setup()` is over-ridden to initialize the device as required by the peripheral.

MCADeviceProperties class

The class `com.bankframe.services.devices.MCADeviceProperties` is a class that wraps a static hashtable of all the properties for the MCA device classes being instantiated by the user. The properties for the MCA device classes are contained in a single properties file `BankframeDevices.properties` which must be on the classpath of the system. The `MCADeviceProperties` object when created allows a device to access its properties during initialization. The `MCADeviceProperties` class contains a hashtable (called `serialPortValueData`) of values which translate string values in the `BankframeDevices.properties` file into `javax.comm.SerialPort` defined values for initializing the serial port.

NOTE: If the properties file is modified while the program is running the changes will not be detected until the program is restarted.

Basic serial port entries in the file `BankframeDevices.properties` are of the form:

```
COM1.MiniMicr.serialport.portname=COM1
```

```
COM1.MiniMicr.serialport.baud=9600
```

```
COM1.MiniMicr.serialport.databits=DATABITS_8
```

```
COM1.MiniMicr.serialport.stopbits=STOPBITS_1
```

```
COM1.MiniMicr.serialport.parity=PARITY_NONE
```

```
COM1.MiniMicr.serialport.flowcontrol=FLOWCONTROL_RTSCCTS_OUT, FLOWCONTROL_RTSCCTS_IN
```

`COM1.MiniMicr` is the name of the device specified in the constructor when the MCA device object is created. The `serialport` entries are settings required for an `MCASerialPort` derived object. These entries are used to initialize the serial port. `serialport.portname` is the name of the port that the device is attached to on the PC/Unix machine.

On an Windows machine this is of the form:

```
serialport.portname=COM1
```

On a Unix machine this would be of the form:

```
serialport.portname=/dev/term/a
```

Each property value in the `BankframeDevice.properties` file is parsed as a comma-separated line. Using the logical bitwise operator `OR (|)` the comma separated values are combined to produce the serial port value required. The property `serialport.flowcontrol` shown above will result in the two serial port defined values `FLOWCONTROL_RTSCS_OUT` and `FLOWCONTROL_RTSCS_IN` being combined using the logical bitwise operator `OR` to produce the serial port flowcontrol type for the device `COM1.MiniMicr`.

The `BankframeDevice.properties` can contain specific settings for each instantiated device object, for example, a MagTek MiniMicr object has the following specific settings:

```
COM1.MiniMicr.commtype=BAUD_9600, DATAPARITY_8N, CTS_DSR_IGNORE, STOPBITS_1, INTERCHAR_DELAY_NO
```

DeviceException class

All exceptions thrown back to a calling class by an MCA device object are of the type `com.bankframe.services.devices.DeviceException`. This class inherits from the class `com.bankframe.EonException`. This class allows localizable messages to be defined in the `BankframeMessages.properties` file.

MCADeviceProtocol class

The class `com.bankframe.services.devices.MCADeviceProtocol` wraps the message protocol creation for the device. All subclassed device objects should subclass `com.bankframe.services.devices.MCADeviceProtocol` for generation of device-specific protocol messages. Messages or commands which are transmitted to the physical device are created in the subclassed `MCADeviceProtocol` class. `MCADeviceProtocol` contains a byte stream which can be passed to the peripheral.

MCA Device Implementations

MCA provides implementations for the following peripheral devices:

- The MagTek Mini Micr cheque reader.
- The MagTek IntelliPIN swipe-card reader.
- The Epson TMU375 slip printer.

MagTek MiniMicr cheque reader device

The classes for this device implementation are in the package:

```
com.bankframe.services.devices.MTMiniMicr.
```

`MTMiniMicr` is a subclass of the base class

```
com.bankframe.services.devices.MCASerialPort.
```

The physical MagTek Mini Micr device is connected to the serial port. The class `MTMiniMicr` over-rides the method `handleEvent(...)` and therefore asynchronously handles serial port events. All the MagTek

Mini Micr specific communication codes are defined in the `com.bankframe.services.devices.MTMiniMicr.MagTekMiniMicrDeviceCodes` interface.

The class `com.bankframe.services.devices.MTMiniMicr.MagTekMiniMicrDeviceProtocol` defines methods for creating MagTek Mini Micr specific serial commands to send to the physical device. The class is a subclass of `com.bankframe.services.devices.MCADeviceProtocol`.

MTMiniMicr(String deviceName) Constructor

The `MTMiniMicr(String deviceName)` is used to instantiate a Mini Micr device object. The String parameter is the unique device name for the device object. This string is used in the `BankframeDevices.properties` file to define the serial communications settings for the device object. The `setup()` method reads the settings for the device object from `BankframeDevices.properties` to setup the device correctly.

MTMiniMicr.setup()

This method is called by the base class method `MCASerialPort.open(...)`. This method does the following:

- Sets up the serial communications to the physical device.
- Configures the format of the cheque data that the physical Mini Micr device will send to the PC when a cheque is swiped.

MTMiniMicr.setCommand(...) and requestCommand(...)

The two forms of the method `MTMiniMicr.setCommand(...)` creates a MagTek MiniMicr command-message which is sent to the physical device by the `MTMiniMicr` object. This is used to set up various settings in the physical Mini Micr device. The method `MTMiniMicr.requestCommand(...)` formats a MagTek MiniMicr request-command which is sent to the physical device by the `MTMiniMicr` object. The method `requestCommand(...)` is used to request information from the physical MiniMicr device.

Using the MiniMicr Device in a Client Application

See the class `com.bankframe.services.devices.unittest.MiniMicrTest` for a basic example of a java client using the Mini Micr device classes. The MCA Example `com.bankframe.examples.devices.fe.ui` demonstrates a full Swing front-end example using the device classes.

The following code is a sample client class using a Mini Micr device object:

```
import com.bankframe.services.devices.*;

import com.bankframe.services.devices.MTMiniMicr.*;

public class myClientClass {

    ...

    MagTekMiniMicr miniMicr;
```

```

public void run() {
    try {
        //The name passed to the device corresponds to the entries
        //used in the BankframeDevices.properties file
        //These names have to match for the device to be setup
        //with the correct serial port setting and specific settings.
        miniMicr = new MagTekMiniMicr("COM1.MiniMicr");

        //Opens the port device specified in the
        //BankframeDevices.properties file:-
        miniMicr.open();

    } catch (DeviceException ex) {
        //exceptions thrown back from device are of type DeviceException
        ex.printStackTrace();
    }

    System.out.println("Swipe check now...");
    //Make the MiniMicr object wait for
    //data received from the connected device
    //(This call times-out after 10 seconds):
    String data = (String)miniMicr.waitForDataAvailable(10000);
    if(data!= null && data.length() != 0) {
        //
        //Check it is data of the correct format, that is, <ESC>CHEQUE_DATA<CR>
        //
        if(data.length() > 0 &&
            data.charAt(0) == MagTekMiniMicrDeviceCodes.ESC) {
            System.out.println("\nGot Check code:" + data.substring(1) + "\n");
        }
    }
}

```



```

    }

}

miniMicr.close( );

miniMicr = null;

} //end of run()

} //end of myClientClass

```

The sample Mini Micr implementation functions as follows:

- The `MagTekMiniMicr` class handles serial events itself including device replies containing the cheque data. The java sample code shown waits for available cheque data by calling the method `waitForDataAvailable(...)`.
- The method `open()` contains the standard setup procedure for the MiniMicr device. The setup can be modified by editing the `BankframeDevices.properties` file before running the test example.
- The method `waitForDataAvailable(int timeoutMilliseconds)` waits the specified time for a message from the Mini Micr physical device. If the data received from the physical Mini Micr is cheque data then the device classes will return this data when `waitForDataAvailable(int timeoutMilliseconds)` is called. The cheque data returned is the raw cheque data as displayed at the foot of the cheque, it is not parsed into separate fields. The device classes do not do any calculations on the cheque data, such as the cheque amount or account number details. It is left to the client class using the Mini Micr device classes to parse the cheque data and calculate the cheque amount or any other details.
- When the Mini Micr is no longer required the connection is shut-down using `close()`
- The classpath must include `mca.jar`. The Java Communications Extension API jar (`comm.jar`) must also be included in the classpath. `mca.jar` is located in `eontec/Common/lib/eontec/`.
- The standard serial port settings for the Mini Micr are: `Baud=9600, data Bits = 8, stop Bits=1, parity = none, flow control = none`.

Epson TM-U375 Slip Printer Device

The classes for this device implementation are in the package:

```
com.bankframe.services.devices.SlipPrinter.
```

The Epson slip-printer device is a subclass of the base class

```
com.bankframe.services.devices.MCASerialPort.
```

The physical slip-printer device is connected to the serial port.

The class `SlipPrinter` over-rides the method `handleEvent(...)` and therefore asynchronously handles serial port events.

All the Epson slip-printer specific communication codes are defined in the `com.bankframe.services.devices.SlipPrinter.SlipPrinterDeviceCodes` interface.

The class `com.bankframe.services.devices.SlipPrinter.SlipPrinterDeviceProtocol` defines methods for creating the slip-printer specific serial commands to send to the physical device. The class contains methods for implementing all the standard printing facilities on an Epson slip-printer.

SlipPrinter(String deviceName) Constructor

The `SlipPrinter(String deviceName)` is used to instantiate a Slip Printer device object. The String parameter is the unique device name for the device object. This string is used in the `BankframeDevices.properties` file to define the serial communications settings for the device object. The `setup()` method reads the settings for the device object from `BankframeDevices.properties` to setup the device correctly.

SlipPrinter.setup()

This method is called by the base class method `MCASerialPort.open(...)`. This method does the following sets up the serial communications to the physical device.

Using the SlipPrinter Device in a Client Application

See the class `com.bankframe.services.devices.unittest.SlipPrinterTest` for a basic example of a java client using the slip-printer device object. The MCA Example `com.bankframe.examples.devices.fe.ui` demonstrates a full Swing front-end example using the device classes.

The following code is a sample client using the slip-printer device object:

```
import com.bankframe.services.devices.*;

import com.bankframe.services.devices.SlipPrinter.*;

import javax.comm.SerialPortEvent;

public class myClient Class {

    String deviceName = "COM2.SlipPrinter";

    SlipPrinter slipPrinter;

    Public void run() {

        try {

            //The name passed to the device

            //corresponds to the entries used in the

            // BankframeDevices.properties file.

            //These names have to match for the device
```

```

//to be setup with the correct serial port setting and specific settings.

slipPrinter = new SlipPrinter( deviceName );

//Opens the port device specified

//in the BankframeDevices.properties file:-

slipPrinter.open();


System.out.println("Testing now...");

slipPrinter.print("hello Ruairi");//prints to printer

slipPrinter.lineFeed();

slipPrinter.clearPrinter();

slipPrinter.test();//prints a few things to printer.


} catch (DeviceException ex) {

    ex.printStackTrace();

}


slipPrinter.close( );

slipPrinter = null;

}

}

```

The slip-printer sample implementation functions as follows:

- The slip-printer device is created passing the device name to the constructor. The device name identifies the device's settings in the `bankframeDevices.properties` file.
- After the slip-printer device is opened methods on the slip-printer are called to do some sample printing. No replies are obtained from the printer for these method calls.
- When the slip-printer device is no longer required the connection is shut-down using `close()`
- The classpath must include `mca.jar`. The Java Communications Extension API jar (`comm.jar`) must also be included in the classpath. `mca.jar` is located in `eontec/Common/lib/eontec/`.
- The standard serial port settings for the Epson TM-U375 slip-printer are: `Baud=9600, data Bits = 8, stop Bits=1, parity = none, flow control = none`.

MagTek IntelliPin Plus Swipe-Card Device

The classes for this device implementation are in the package:
`com.bankframe.services.devices.MTPinPad`.

The `MTPinPad` class subclasses the base class

`com.bankframe.services.devices.MCASerialPort`.

The physical MagTek IntelliPIN device is connected to the serial port.

The class implements the method `handleEvent(..)` and therefore asynchronously handles serial port events

All the MagTek MiniMicr specific communication codes are defined in the
`com.bankframe.services.devices.MTPinPad.MagTekIntelliPINDeviceCodes` interface.

The class `com.bankframe.services.devices.MTPinPad.MagTekIntelliPINDeviceProtocol` defines methods for creating MagTek IntelliPIN specific serial commands to send to the physical device.

MagTekIntelliPIN(String deviceName) Constructor

The `MagTekIntelliPIN(String deviceName, boolean decryptPinData)` is used to instantiate an IntelliPIN device object. The `String` parameter `deviceName` is the unique device name for the device object. This string is specified in the `BankframeDevices.properties` file to define the serial communications settings for the device object. The `setup()` method reads the settings for the device object from `BankframeDevices.properties` to setup the device correctly. The `boolean` parameter `decryptPinData` specifies if pin number data received from the physical device is decrypted by the `MagTekIntelliPIN` device object or remains encrypted. It may be desirable not to decrypt the pin number in the client but to transmit the pin number while still encrypted to a Server where the pin number will be decrypted and validated.

MagTekIntelliPIN.open(...)

Once the `MagTekIntelliPIN` object has been instantiated one of the two `open()` methods is called to set up and connect to the physical IntelliPIN device. The `open()` method has the following two forms:

- `open(long masterKeyResponseTimeout)`. This form generates a unique master encryption key for encrypting pin data during communication with the physical IntelliPIN device. The Java Cryptography API generates the unique encryption key. This `open()` method is slower than the second form.
- `open(byte[] theMasterKey, long masterKeyResponseTimeout)`. This form allows the user to specify a master encryption key for encrypting pin data during communication with the physical IntelliPIN device. The `byte[]` array is an 8 byte DES encryption key.

In both cases the `MagTekIntelliPIN` object must download the master encryption key to the physical device. The `long` argument `masterKeyResponseTimeout` is the time-out period for a positive response from the physical device verifying that the encryption key was successfully downloaded. If the device does not respond in this time then a `DeviceException` is thrown.

The sequence of method calls within the `open()` method is as follows:

- 1 an encryption key is generated using the `com.bankframe.util.Cryptography` class. This class wraps the standard Java Cryptography API.
- 2 the base class `MCASerialPort.open()` method is called.

- 3 the base class `MCASerialPort.open()` method calls the over-ridden `MagTekIntelliPIN.setup()` method which sets up the serial communications parameters for the physical device. The master encryption key is downloaded to the physical IntelliPIN device. This throws a `DeviceException` if the physical device does not respond confirming the key within the period `masterKeyResponseTimeout`.

MagTekIntelliPIN.setup()

This method is called by the base class method `MCASerialPort.open(...)`. This method does the following:

- Sets up the serial communications to the physical device.
- The master encryption key is downloaded to the physical IntelliPIN device. This throws a `DeviceException` if the physical device does not respond confirming the key within the period `masterKeyResponseTimeout`.

MagTekIntelliPIN.replaceDefaultDisplay(...)

The method

```
MagTekIntelliPIN.replaceDefaultDisplay(String displayNumber, String lineOne, String lineTwo)
```

replaces one of the default displays on the physical MagTek IntelliPIN pad's LCD display. This command can be used to customize the IntelliPIN display for a particular language/bank. This customized display is stored in the physical device so it appears again when it is next turned on. See java docs for usage.

MagTekIntelliPIN.enableDefaultDisplay(...)

The method `MagTekIntelliPIN.enableDefaultDisplay()` disables any previously customized default displays. The factory installed displays are all enabled on the physical device.

MagTekIntelliPIN.cardDataEntryRequest (...)

The method

```
MagTekIntelliPIN.cardDataEntryRequest(String firstMessage, String secondMessage, long timeOut)
```

instructs the physical IntelliPIN device to prompt the user for a card swipe. The two messages are displayed on the IntelliPIN's LCD display. The user has the period `timeOut` to swipe a card before the method returns.

MagTekIntelliPIN.pinEntryRequest (...)

The method

```
MagTekIntelliPIN.pinEntryRequest(String accountNumber, char keyNumber, String transactionAmount, long timeout)
```

instructs the physical IntelliPIN device to prompt the user for a pin number entry. The `accountNumber` is used for encrypting the returned pin number. `KeyNumber` specifies whether to use the Master Key or a session key. Currently only the master key is supported by the MCA object. The `keyNumber` to

specify use of a Master Key is '4'. `TransactionAmount` is a decimal string to two decimal places which is displayed on the IntelliPIN LCD display. The user has the period `timeOut` to enter a pin number.

MagTekIntelliPIN.cancelSessionRequest (...)

The method

```
MagTekIntelliPIN.cancelSessionRequest()
```

instructs the physical IntelliPIN device to cancel the current request in progress.

MagTekIntelliPIN.displaySingleString (...)

The method

```
MagTekIntelliPIN.displaySingleString(String firstMessage, String secondMessage)
```

instructs the physical IntelliPIN device to display the two strings on its LCD display.

MagTekIntelliPIN.requestSoftSwitch (...)

The method

```
MagTekIntelliPIN.requestSoftSwitch(char switchNumber, long timeout)
```

requests the current configuration settings of the physical IntelliPIN device.

MagTekIntelliPIN.setSoftSwitch (...)

The method

```
MagTekIntelliPIN.setSoftSwitch(char switchNumber, byte theSettingData, long timeOut)
```

sets the specified configuration settings in the physical IntelliPIN device.

MagTekIntelliPIN.waitCondition...(...)

The methods

```
MagTekIntelliPIN.waitConditionCardData(long timeout)
```

```
MagTekIntelliPIN.waitConditionKeyLoaded(long timeout)
```

```
MagTekIntelliPIN.waitConditionPinData(long timeout)
```

```
MagTekIntelliPIN.waitConditionRequestSettings(long timeout)
```

start a wait cycle in the IntelliPIN device object until the specified condition has occurred.

- `waitConditionCardData(...)` waits until a card has been swiped and the device object has received card track details from the physical device.
- `waitConditionKeyLoaded(...)` waits until the physical device responds indicating that it has accepted the downloaded Master Encryption key.
- `waitConditionPinData(...)` waits until a pin number has been entered by the user and the device object has received the pin number.

- `waitConditionRequestSettings(...)` waits until the physical device has responded to a request to change its settings.

PinPadListener interface

The class `com.bankframe.services.devices.MTPinPad.PinPadListener` interface has one method `handlePinPad(PinPadEvent event)`. A client implements this listener interface and calls the method `MagTekIntelliPIN.addPinPadListener(PinPadListener ppl)` to register its listener. The method `handlePinPad(PinPadEvent event)` is called by the `MagTekIntelliPIN` device object when an event occurs.

MagTekIntelliPIN.addPinPadListener (...)

The method

```
MagTekIntelliPIN.addPinPadListener( PinPadListener ppl)
```

allows a client object to register a listener class to receive notification of asynchronous `MagTekIntelliPIN` events. The listener class will receive `PinPadEvent` objects when one of the following events occur:

- When a card is swiped
- A pin number is entered
- An Exception occurs parsing data received from the physical IntelliPIN device.

An alternative to this asynchronous method of receiving IntelliPIN events is to use the base class method `waitForDataAvailable(long timeout)` specifying the period to wait for the data to be available. This method will return any of the above events that occur to the java client object.

PinPadEvent class

The `com.bankframe.services.devices.MTPinPad.PinPadEvent` class stores event details to be passed to the client object. Card track details, Pin number and exception details can be obtained from this object when passed to the client object.

PinDataBlock class

The `com.bankframe.services.devices.MTPinPad.PinDataBlock` class has two purposes:

- It stores the encrypted pin number received from the physical device.
- It provides a method `decrypt(SecretKey masterKey, String algorithm, String provider)` for decrypting and un-mangling the received pin data from the physical IntelliPIN. This method uses the `com.bankframe.util.Cryptography` class. This class wraps the standard Java Cryptography API. The `masterKey` must be the same master encryption key originally downloaded to the physical device during the device setup. The algorithm and provider must be of the same form used to generate the original encryption key. If the encryption key was generated using `algorithm="DES"` and `provider="SunJCE"` then the `decrypt()` method has to be called using a form of the `DES` algorithm, for example, `"DES/ECB/NoPadding"`

CardData class

The `com.bankframe.services.devices.MTPinPad.CardData` class stores the card track details obtained from the physical IntelliPIN device. The object parses the raw card data into the three track details.

Using the IntelliPIN Pad Card-Swipe Device in a Client Application

See the class `com.bankframe.services.devices.unittest.PinPadTest` for a basic example of a java client using the slip-printer device. The MCA Example `com.bankframe.examples.devices.fe.ui` demonstrates a full Swing front-end example using the device classes.

The following are the basic steps that a client java class generally takes to use the IntelliPIN device classes:

- 1 Instantiate the IntelliPIN device object.
- 2 Open the IntelliPIN device passing it a Master encryption key. The pin data returned by the physical IntelliPIN device to the PC will be encrypted using this key.
- 3 Request the user to swipe their card through the physical IntelliPIN device.
- 4 Wait for a card to be swiped by the user.
- 5 Request the user to enter their pin number on the physical IntelliPIN device.
- 6 Wait for the pin to be entered by the user.
- 7 Close the IntelliPIN device.

For steps 3 and 5 the returned data can be parsed and validated by the client java class as required.

The following code is a sample client using the MagTekIntelliPIN device object:

```
import com.bankframe.services.devices.*;

import com.bankframe.services.devices.SlipPrinter.*;

public class PinPadTest extends Object implements PinPadListener{

    MagTekIntelliPIN pinPad;

    void run() {

        try {

            //The name passed to the device corresponds to the entries used in the

            //BankframeDevices.properties file

            //These names have to match for the device to be setup with the correct

            //serial port setting and specific settings.

            pinPad = new MagTekIntelliPIN("COM1.IntelliPinPad",true);
```



```

//PinPad object notifys this test object when a check is swiped,
//a pin is entered, or when an exception occurs.
pinPad.addPinPadListener(this);

//Open the port device specified in the BankframeDevices.properties file:-
//
//There are two forms of this method,
//First Form of open(...):-
//Takes a parameter which is the Master Encryption key
//sent to the IntelliPIN to encrypt all messages.
//Second parameter is the time-out to wait for response
//from device after loading master key.
//DeviceException is thrown if pinPad times-out.
//Encryption key specified in MagTek Programing reference
// manual:"23AB4589EF6701CD", passed in as a byte array:-

byte[] theMasterKey = {(byte)0x23,(byte)0xAB,(byte)0x45
                        ,(byte)0x89,(byte)0xEF,(byte)0x67
                        ,(byte)0x01,(byte)0xCD};

pinPad.open(theMasterKey, 30000);

//Second form of open(...):-
//Parameter is the time-out to wait for
//response from device after loading master key.
//This method generates an Encryption Master key
//and is therefore slower than the above method.
//DeviceException is thrown if pinPad times-out.

```

```

        //pinPad.open(30000);

        ...

        BankframeLog.log(Bankframe.DEBUG,BankframeLogConstants.
BANKFRAME_SUBSYSTEM,"Swipe card now...");

        if (pinPad.cardDataEntryRequest("swipe your", "card now",10000)) {

            //First param = an Account Number which is used in the
            //encryption of the returned PIN number, can be blank "":-
            //Second param = use the Master key:-
            //Third param = the transaction amount to two decimal places,
            //can be blank "":-

            if (!pinPad.pinEntryRequest("4761234567812348"

                                        ,MagTekIntelliPINDeviceCodes.UseMasterKey

                                        ,"12300", 20000);

                //wait 30 seconds for pin entry then cancel

                pinPad.cancelSessionRequest();

            }

        }

    } catch (DeviceException ex) {

        BankframeLog.log(Bankframe.ERROR,BankframeLogConstants.
BANKFRAME_SUBSYSTEM,ex);

    }

    BankframeLog.log(Bankframe.DEBUG,BankframeLogConstants.
BANKFRAME_SUBSYSTEM,"Closing down pinPad...");

    pinPad.close( );

    pinPad = null;

```

```

    }

    public void handlePinPad(PinPadEvent event) {

        if (event.getType() == event.EXCEPTION_OCCURRED) {

            BankframeLog.log(Bankframe.DEBUG,BankframeLogConstants.
                BANKFRAME_SUBSYSTEM,event.toString());

            BankframeLog.log(Bankframe.DEBUG,BankframeLogConstants.
                BANKFRAME_SUBSYSTEM,"Closing down pinPad...");

            pinPad.close();

            System.exit(1);

        }

        BankframeLog.log(Bankframe.DEBUG,BankframeLogConstants.
            BANKFRAME_SUBSYSTEM,event.toString());

        // TODO:

        // Do something with the result here, ie., display it or validate it.

        //

    }

}

```

The sample MagTekIntelliPIN implementation functions as follows:

- The IntelliPIN Pad device is created passing the device name to the constructor. The device name identifies the devices settings in the bankframeDevices.properties file.
- The test class implements the com.bankframe.services.devices.MTPinPad.PinPadListener interface and therefore is directly notified when a card has been swiped or a pin entered by a user. The test class registers itself as a Pin Pad listener by calling the method addPinPadListener(). When an IntelliPIN pad event occurs the method handlePINPad() is called on the test class. This event handler is also used to capture exceptions which may occur on the input thread of the IntelliPIN device allowing the test class to handle all exceptions/errors.

NOTE: IntelliPIN events/exceptions could also be detected directly by using the following code:

```

// To wait for pin entry do the following in the code:

Object data = null;

if (pinPad.waitConditionPinData(30000)) {

    data = pinPad.getReceivedData();
}

```

```
//do something with the pin data.

}
```

- There are two forms of the IntelliPIN Pad device `open()` method. The first form of the `open()` method takes an array of 8 bytes representing a standard `DES` encryption key. This key is sent to the physical IntelliPIN device as a Master Key for encrypting all pin data sent back to the PC. The second form of the `open()` method generates a `DES` encryption key itself, this method is slower but generates a unique Master Key. The second parameter is the time-out value, the physical IntelliPIN device must accept the Master Key within this time.
- The test class tells the IntelliPIN device to request a card swipe by calling the method `cardDataEntryRequest()`. The two strings are the text that are shown on the IntelliPIN device during the request. The third parameter is the time-out period in milliseconds, this specifies the length of time that the user has to swipe a card
- The test class tells the IntelliPIN device to request a pin entry from the user by calling the method `pinPad.pinEntryRequest("4761234567812348", MagTekIntelliPINDeviceCodes.UseMasterKey, "12300", 20000)`. The first parameter is an account number which is used in the encryption of the pin number entered by the user on the physical IntelliPIN device. The second parameter tells the device to use the Master key created previously. The third parameter is the amount of the transaction to two decimal places. The two strings can be empty. The fourth parameter is the time-out period in milliseconds, if the user does not enter a pin in this time then the session is cancelled by calling the method `pinPad.cancelSessionRequest()`.
- When the IntelliPIN device is no longer required the connection is shut-down using `pinPad.close()`
- The classpath must include `mca.jar`. The Java Communications Extension API jar (`comm.jar`) must also be included in the classpath. `mca.jar` is located in `eontec/Common/lib/eontec/`.
- The Java Cryptography API jar files must also be on the classpath or in the `jre\lib\ext` folder if the `jre` is being used to compile and run the test class.
- The standard serial port settings for the MagTek IntelliPIN card-swipe are: `Baud=9600, data Bits = 7, stop Bits=1, parity = even, flow control = none`.

Implementing a New MCA Peripheral Device Object

The basic requirements for implementing a new type of MCA device object are as follows:

- The device classes must be in a subpackage of the package `com.bankframe.services.devices`
- The device object must implement the interface `com.bankframe.services.devices.MCADevice`. This interface declares all the standard device methods.
- The device object must instantiate the `com.bankframe.services.devices.MCADeviceProperties` object to obtain the settings for the device.
- To create messages and commands to transmit to the physical device the class `com.bankframe.services.devices.MCADeviceProtocol` is subclassed. The subclass implements message-generating code specific to the physical device's communication protocol.

- The `BankframeDevices.properties` file must be present on the classpath and must contain the necessary entries for each device object being instantiated in the client application. If the file or necessary entries are missing then the device initialization will fail.
- All exceptions must be returned to the calling class/client as a `com.bankframe.services.devices.DeviceException`. This is generally achieved by converting an exception to a `com.bankframe.services.devices.DeviceException` as follows:

```
throw (DeviceException) new DeviceException(0).fromException(theGeneralException);
```

Implementing a Serial-Port device

To design and implement a new type of MCA serial-port device you must implement the following:

- A serial port device must subclass the basic `com.bankframe.services.devices.MCASerialPort` abstract class. `MCASerialPort` implements the `MCADevice` interface, and contains a `com.bankframe.services.devices.MCADeviceProperties` object.
- The derived device class must implement the `MCASerialPort` abstract method `protected void setup()`
- The `String deviceName` member of the `MCASerialPort` must be initialized during creation of the derived device class, for example, `"Com1.SlipPrinter"`. This is normally performed in the constructor. During initialization `deviceName` is passed to the `MCADeviceProperties` object to obtain the correct settings for the device object.
- The device class subclassing `MCASerialPort` implements `void handleEvent(java.util.EventObject event)` if it has to use data sent by the physical device to the PC. When a serial event occurs this method will always be called by `MCASerialPort`.
- Any data captured, parsed and validated by the method `handleEvent(...)` must be stored in the `MCASerialPort` variable `Object receivedData`. This is accomplished by calling the `MCASerialPort` method `dataAvailable(Object data)`. Therefore the client object can then query the device object for available data by calling the method `public Object waitForDataAvailable(int timeOut)`.

Client-side Applet

To use the MCA devices support in an Applet a policy file is required with entries of the following form:

```
grant codebase "http://theAppletSite" signedBy "THE_ALIAS_HERE" {
permission java.lang.RuntimePermission "loadLibrary.win32com";
permission java.io.FilePermission "${java.home}\\lib\\win32com.dll", "read";
permission java.io.FilePermission "${java.home}\\lib\\javax.comm.properties",
"read";
permission java.io.FilePermission "${java.home}\\lib\\javax.comm.properties",
"delete";
permission java.util.PropertyPermission "java.home", "read";
permission java.util.PropertyPermission "javax.comm.properties", "read";
permission java.io.FilePermission "BankframeFrontendApplication.properties",
"read";
```

```

permission java.util.PropertyPermission "BankframeFrontendApplication.properties",
"read";

permission java.io.FilePermission "BankframeDatePatterns.properties", "read";

permission java.util.PropertyPermission "BankframeDatePatterns.properties", "read";

permission java.io.FilePermission "BankframeDevices.properties", "read";

permission java.util.PropertyPermission "BankframeDevices.properties", "read";

permission java.io.FilePermission "BankframeFrontend.properties", "read";

permission java.util.PropertyPermission "BankframeFrontend.properties", "read";

permission java.io.FilePermission "BankframeMessages.properties", "read";

permission java.util.PropertyPermission "BankframeMessages.properties", "read";

permission java.io.FilePermission "BankframeResource.properties", "read";

permission java.util.PropertyPermission "BankframeResource.properties", "read";

};

```

The MCA example `com.bankframe.examples.devices.fe.ui` demonstrates a full Swing front-end example using the device classes. To use this example as an Applet the Sun Java Plug-in is required.

Unit Test classes

The MCA device support classes can be tested by using the unit-test classes in `mca.jar`. The device unit tests are in the package `com.bankframe.services.devices.unittest`. These can be used as standalone console applications or as an applet. They initialise and start an MCA device. To use a unit test class the following command is used:

```
java -classpath ./myClasses/mca.jar $JAVA_HOME/lib/ext/comm.jar
```

where `$JAVA_HOME` is the location of the `JDK/JRE` being used

The Java Communications API must be installed on the machine and the `BankframeDevices.properties` file must contain the correct settings to initialize the required device.

The unit-tests can be used:

- As a simple console application with no graphical user interface or
- As an applet in a html page

The MagTek MiniMicr is tested by the class `com.bankframe.services.devices.unittest.MiniMicrTest`. If the device is working then the user will be prompted to swipe a cheque or to exit.

The MagTek IntelliPIN is tested by the class `com.bankframe.services.devices.unittest.PinPadTest`. If the device is working then the user will be prompted to swipe a card or to exit. The Java Cryptography API must be installed on the machine.

The MagTek MiniMicr is tested by the class `com.bankframe.services.devices.unittest.SlipPrinterTest`. If the device is working then the slip-printer will print out test information.

Sample Implementation

Sample Serial Port Communication classes are contained in the javacomm extension pack.

See the package `com.bankframe.services.devices.unittest` for a basic examples of using the MCA implemented device types. The MCA example `com.bankframe.examples.devices.fe.ui` demonstrates a full Swing front-end example using the device classes.

Printing Framework

The MCA Services printing framework supports high quality form printing via the third party Accelio product. The framework includes a Session EJB which accepts a `Vector` of `DataPackets` and generates an XML file through XSLT. The XML file is then processed by Accelio to produce an output to the specified printing device.

The MCA Services printing framework is implemented as a standard two layer Session Bean EJB as follows:

- The solution set is provided in the `com.bankframe.services.print` package.
- The implementation is provided in the `com.bankframe.services.impl.print` package.

com.bankframe.services.print

This package defines the MCA printing framework to produce the XML file necessary for use with Accelio. It contains the following classes:

PrintBean

This class provides all the methods for accepting a vector of `DataPackets` and exporting an XML file for use with Accelio.

imPrint()

This method has the following signature:

```
public Vector imPrint(Vector data)

throws ProcessingErrorException, RemoteException;
```

This method accepts a `Vector` of `DataPackets` containing the data to be printed, and transforms the data in the vector into an XML file, for processing with Accelio.

XML files do not accept tags with white space, however the keys in the `DataPacket` contain white space. This method replaces space characters within the keys in the `DataPacket` with underscores, before parsing the `Vector` of data. The data is transformed from the `Vector` to Oracle's Siebel Retail Finance XML format, and then transformed to Accelio XML format.

The first `DataPacket` in the `Vector` must contain a Key named `JF_JOB_CARD`. This key is a requirement for the Printing process for Accelio. The key specifies values, which define printing information. The Accelio jobname must be specified.

Generating the Service

The `REQUEST_ID` of the first `DataPacket` in the `Vector` must have a `REQUEST_ID` of `MC065`. For example, from a JSP front end, the following would be specified within the JSP:

```
<FORM NAME="printPage" ACTION=" " METHOD="post">

<p>Title

    <input type="text" name="Title">

</p>

<p>First Name

    <input type="text" name="FirstName">

</p>

<p>Surname

    <input type="text" name="Surname">

</p>

<input type="hidden" name="REQUEST_ID" value="MC065">

<input type="hidden" name="JF_JOB_CARD" value="jobname printername">

<input type="submit" name="Submit" value="Submit">
```

Calling the Service from another Session

When calling the Print service from another session, a `Vector` of `DataPackets` must be passed to the `PrintBean` and the `imPrint` method, for example:

```
import com.bankframe.services.impl.PrintHome;

import com.bankframe.services.Print;

Class SampleBankingProcessBean {

    Public testPrint() {

        PrintHome home = (PrintHome)Server.lookup("eontec.bankframe.print");

        Print print = home.create();
```



```

        create DataPackets

        dp.put("JF_JOB_CARD", "jobname printername");

        print.imPrint(dataPackets);

    }

}

```

NOTE: The `JF_JOB_CARD` must be specified within the `Vector`, and should be placed as the first `DataPacket` in the `Vector`.

Accelio Architecture

`BankframeResource.properties` holds three properties relating to the XML file produced by the print service. These three properties are:

Collector Directory location: for example,

```
print.datFilePath=D:\\JetForm\\Central\\Server\\data\\
```

The stylesheet location used by the print service, for example,

```
print.styleSheetLocation=D:\\Mca\\Printing\\stylesheets\\
```

The stylesheet name used by the print service, for example, `print.styleSheetName=JetFormXSL.XSL`

Control Process

The MCA Services printing framework requires two instances of Accelio, this involves two installs.

- The XML file produced by the printing service is stored temporarily in the Accelio collector directory and has the extension `.DAT`
- This `.DAT` file is then picked up by the Accelio control, converted to a Field-Nominated Format file and dropped into another collector directory in the second instance of Accelio, for example, `D:\\JetForm\\Central\\Server2\\data` Field-Nominated Format is structured keys and values from the `Vector` of `DataPackets`, for example,

```

^job jobname printername

^field LoanType

car

^field Occupation

Architect

^field DOB

23/12/74

^field Basic_Income

40000

```

```
^field Home_Telephone
5556767
```

- Dropping the field-nominated format file in the second instance of Accelio is a result of the task table entry in the first instance of Accelio, for example,

```
!x JFNOJOB * xmlimport "-config @IniFilename. data.uri=@InFile.
output.uri=D:\jetform\central\server2\data\@InFileBase..dat" "Outputs DAT to
other Central instance"
```

- The task above is called `JFNOJOB` as the original XML file contains a `JF_JOB_CARD` entry and no specific header information.
- As there is a `JF_JOB_CARD` entry in the first file, this now gets converted to the `jobname` header information in the field-nominated format file and contains the `jobname` to be carried out on the second instance of Accelio.

Example conversion

- The field nominated format file will contain `jobname` information derived from the original `JF_JOB_CARD`: The first instance XML file contains:

```
<?xml version="1.0" encoding="UTF-8"?>

<v>

  <d>

    <JF_JOB_CARD>jobname printername</JF_JOB_CARD>

    <OWNER>eontec Ltd</OWNER>

    <DATA_PACKET_NAME>Test1 DP</DATA_PACKET_NAME>

  </d>
```

- This information will be converted to the header of the field-nominated format file in the second instance:

```
^job jobname printername
```

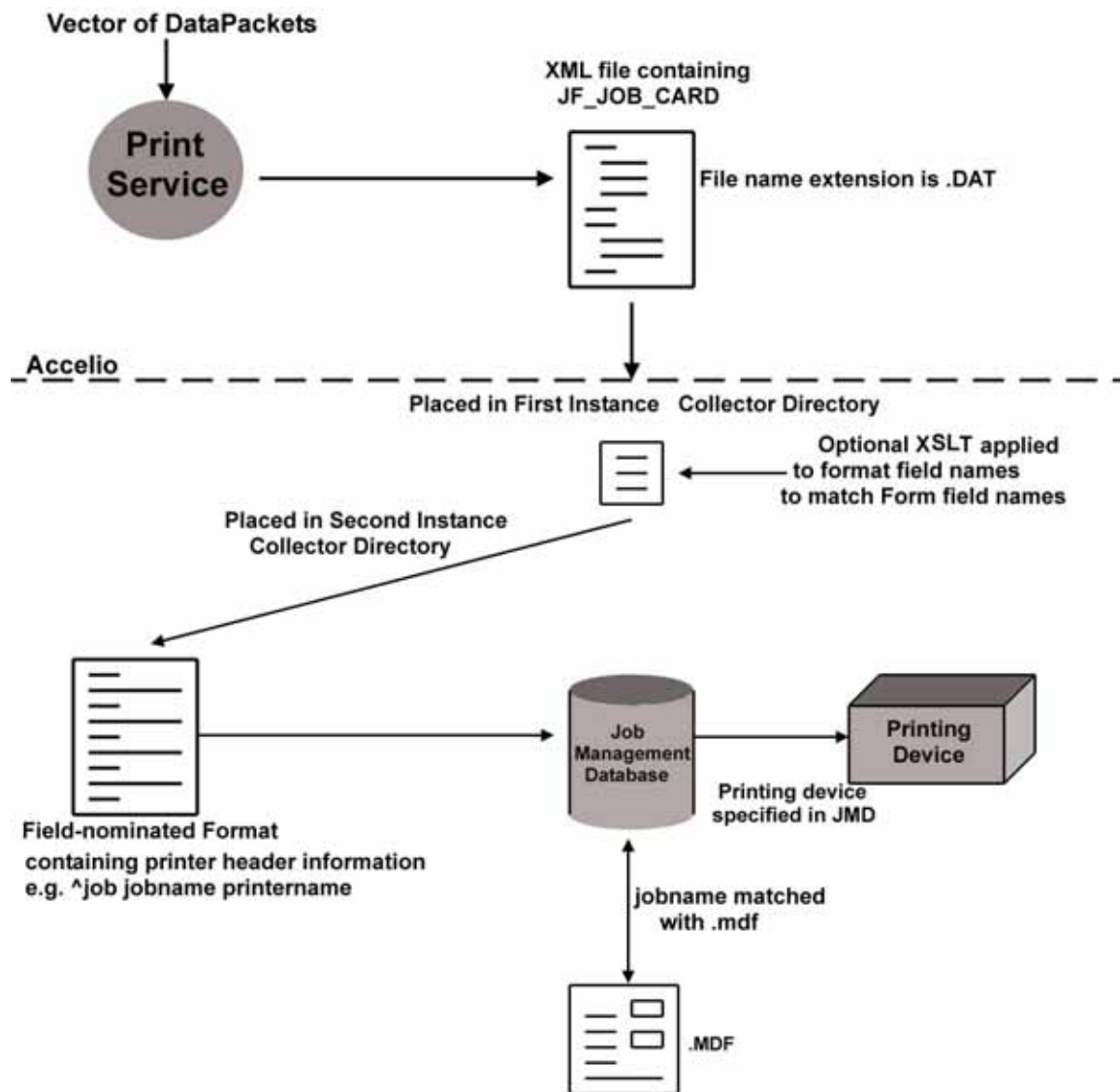
- The `jobname` then identifies the task to be carried out on the Printer task table of the second instance of Accelio, for example,

```
!f jobname HPLJETU d:\mca\printing\forms\xmltest4.mdf * 1 T JFMERGE * * C "test
print"
```

- This identifies the task to be carried out `jobname`, the printer name `printername`, the path of the form name used for example, `d:\mca\printing\forms\xmltest4.mdf` and other printing task information, and the standard layout is as follows:

```
# !f <Job name> <Printer name> <Form file> <Preamble file> <Macro number> <Load
flag> <Task id> <Input file> <Output file> <On error> <Comments>
```

Printing Framework Architecture Overview



Caching Framework

This topic describes the generic caching framework provided by MCA Services.

Uses of caching

Caching of data can be used anytime it is expensive (in terms of time) to access some data. By caching data in local memory unnecessary expensive data accesses can be avoided. Below are some examples of where caching is used in MCA:

Creating JNDI initial contexts	Creating JNDI initial contexts is very expensive. By caching initial contexts they can be re-used, meaning that each initial context only needs to be created once
EJB Home references	Looking up EJB home references is also expensive, so again caching EJB Home references reduces the number of lookups that have to be done, thus increasing performance
Financial Process Integrator	Typically communicating with legacy systems is an expensive process, therefore it makes sense to try and cache data received from hosts, in order to minimise the communication required with the legacy system
Configuration information	Configuration information is stored in a file called <code>BankframeResource.properties</code> . Reading from a file is an expensive process so the contents of the file are cached in memory to improve performance.

In Memory and Persistent Caches

Caches can be divided into two broad categories:

In Memory Cache

Local cache. This kind of cache only uses data stored in local memory, that is, the data in the cache is never stored in a persistent store. The initial context and EJB home caches are examples of this kind of cache. This kind of cache is typically used to cache objects that are expensive to instantiate.

Persistent Cache

This kind of cache is used to cache data that is stored in some persistent store. The Financial Process Integrator and configuration information are examples of this kind of cache. This kind of cache is used to cache objects that are expensive to read from the persistent store. This category of cache can be sub-divided into two more categories:

Read-only caches. Read-only caches contain data that is only ever read and can never be updated.

Read-write caches. Read-write caches contain data that can be read, and also re-written to the persistent store

Functionality of a Cache

Any cache must provide the following functions:

- Associate an object with a key that can be used to retrieve the object at a later time
- Provide a means to iterate over the contents of the cache
- Provide a means to manage the size of the cache, by removing expired data from the cache

In addition persistent caches must provide the following functions:

- Maintain consistency between the in memory cache and the data stored in the persistent store, that is, if the data in the persistent store changes, the cache must be updated.

- Read-write caches must provide a means to flush changes made to cached objects to the persistent store

What Does the Generic Framework Provide?

The generic cache framework provides:

- A generic implementation of an in memory cache.
- A pluggable [CachePolicy](#) interface that allows the policy used for removing expired objects to be customized.
- A clean up interval to define how often the CachePolicy will be asked to check for expired objects
- A framework for implementing persistent caches that supports maintaining the cache consistency and flushing updates to the persistent store.
- An easy to use API; the [Cache](#) class implements the [java.util.Map](#) interface so that its API will be familiar to all Java programmers, and so that it can be easily integrated into code that previously used [Hashtables](#) or [HashMaps](#) for caching data.
- Non key indexing of caches to facilitate retrieval of data in the cache if the primary key value is not known.

com.bankframe.services.cache

The generic caching framework is implemented in the [com.bankframe.services.cache](#) package. This package contains the following interfaces:

- [Cache](#): this interface defines the basic methods that all cache implementations must provide.
- [PersistentCache](#): this interface extends the [Cache](#) interface and must be implemented by all persistent caches that are configured via [BankframeResource.properties](#).
- [CachePolicy](#); this interface defines a mechanism for customizing the policy used for removing expired objects from the cache.
- [ConfigurableCachePolicy](#): This interface extends [CachePolicy](#) and provides a means for policy objects to be configured via the [BankframeResource.properties](#) file. This interface must be implemented by all policy objects that can be configured via [BankframeResource.properties](#)
- [NamedCache](#); this interface ensures implementing cache classes can be identified by [String](#) names.
- [CacheIndexer](#); this interface defines the basic methods that all cache index implementations must provide.
- [CacheListener](#); this interface ensures implementing classes can be notified of events in a cache that affect a particular key.

com.bankframe.services.cache.Cache

This interface defines all the methods that all caches must implement. It extends the [java.util.Map](#) interface which means that all caches implementing this interface must also implement the map interface.

com.bankframe.services.cache.GenericCache

This class provides a generic implementation of a local in memory cache. It also provides the means for this class to be extended to provide a persistent cache. To establish the clean up interval, this class refers to `cache.cleaninterval` property. If not defined, the value defaults to 10000 milliseconds. This class implements the `com.bankframe.services.cache.PersistentCache` interface. The constructors and most commonly used methods unique to the `GenericCache` class are listed below.

Constructors

The `GenericCache` class has a number of constructors, each of which allows the `GenericCache` class to be used in a different fashion. If a constructor does not specify a `CachePolicy` object then the default behaviour will be to keep an entry in the `GenericCache` until it is removed by calling one of: `remove()`, `removeAll()` or `clear()`.

GenericCache(). This constructor can be used to create an in memory cache that has no caching policy. When a `GenericCache` is created with this constructor its behaviour will be the same as the `java.util.HashMap` class.

GenericCache(CachePolicy policy). This constructor can be used to create an in memory cache that uses the specified caching policy.

GenericCache(Map persistentMap). This constructor can be used to create a persistent cache. The `persistentMap` parameter specifies a `java.util.Map` implementation that accesses the persistent store directly. A cache created with this constructor will have no caching policy.

GenericCache(Map persistentMap, CachePolicy policy). This constructor can be used to create a persistent cache that uses the specified caching policy.

put() method

The `put()` method is used to store an object in the `Cache`. This method is declared by the `java.util.Map` interface and has the following signature:

```
public Object put(Object key, Object value);
```

- The `key` parameter specifies the key for the object to store in the cache
- The `value` parameter is the object to store in the cache
- This method returns the previous value associated with the key, or `null` if there was no previous value

get() method

The `get()` method is used to retrieve values from the cache. This method is declared by the `java.util.Map` interface and has the following signature:

```
public Object get(Object key);
```

- The `key` parameter specifies the object to retrieve from the cache
- This method returns the cached object or `null` if the object was not found in the cache

enableCaching() method

The `enableCaching()` method is used with persistent caches, it can be used to enable or disable caching. This method is declared in the `com.bankframe.services.cache.Cache` interface, it has the following signature:

```
public void enableCaching(boolean enableCache);
```

- The `enableCache` parameter specifies whether to enable or disable caching.
- When caching is disabled the cache operates in pass-thru mode; it passes `get()` or `put()` calls straight through to the persistent store. This method can be used when it is critical to read or write values directly from or to the persistent store.

remove() method

The `remove()` method is used to remove an object from the cache. With persistent caches the object is removed from the persistent store as well. The remove method notifies `CacheListeners` of the removed key(s) by calling `cacheChanged(CacheEvent)`.

The `remove()` method has two forms as follows:

Method Form	Description
<code>public Object remove(Object key);</code>	The <code>key</code> parameter specifies the key of the object to remove from the cache. This method removes a single object from the cache and from the persistent store. This method is declared in the <code>java.util.Map</code> interface.
<code>public void remove(Set keySet);</code>	The <code>keySet</code> parameter specifies a <code>Set</code> of keys that identify the objects to remove from the cache. This method removes the objects from the cache and the persistent store. This method is declared in the <code>Cache</code> interface.

removeAll() method

The `removeAll()` method is used to remove all objects from the cache. With persistent caches all objects are removed from the persistent store as well. To remove objects from the cache only use the `clear()` method. The `removeAll()` method is specific to the `Cache` interface and has the following signature:

```
public void removeAll();
```

cleanup() method

This protected method is used to remove expired objects from the cache. This method uses the `CachePolicy` object to determine what objects should be removed. This method is specific to the `Cache` interface and has the following signature:

```
protected void cleanup();
```

- This method is called whenever the following `Cache` methods are called:

```
put()
```

```
putAll()

remove()
```

The cleanup method determines if the number of milliseconds specified by the `cache.cleaninterval` property have passed before investigating the cache to remove expired items. This is for performance reasons, allowing users determine appropriate cleanup times according to the requirements of the specific application data.

The `cache.cleaninterval` setting is configured in the `BankframeResource.properties` file.

createCacheMapInstance() method

This method is used to create the `java.util.Map` instance that is used to store cached values. In the `GenericCache` class, the implementation of this method creates an instance of the `java.util.HashMap` class, however this method can be overridden if it is necessary to use another class.

This method is specific to the `GenericCache` class and has the following signature:

```
protected Map createCacheMapInstance();
```

GetCacheName() method

This method returns the name of the group that this cache is a member of.

com.bankframe.services.cache.NullCache

This is a `Cache` class that is used at runtime when caching is not required. It may be preferred to turn off a particular cache in some circumstances. This can be achieved by setting the corresponding cache class property value to `com.bankframe.services.cache.NullCache`. Policy and persistentMap settings will be ignored. This `Cache` class has a substantially less memory overhead than using another `Cache` with short timeout values.

com.bankframe.services.cache.JMSCache

This class extends the `com.bankframe.services.cache.GenericCache` class to provide a JMS (Java Messaging Service) supported distributed caching implementation. This service extends the current caching framework and can be configured with the different caching policies.

In situations where an environment has caches across multiple JVMs (Java Virtual Machines) it can be necessary to have data consistency across all instances. The MCA Services JMS Caching does this when a `remove()` method is called to remove a key from the local cache. This `remove()` method publishes a message onto a JMS Topic to remove all occurrences of this key in caches across the cluster. A JMS Topic is analogous to a list of messages that is shared among multiple JVMs. Each JVM can have a JMS client that publishes messages to the topic and JMS Listeners in other JVMs who are subscribed to this JMS Topic can read these messages from the topic.

The message driven bean `com.bankframe.services.cache.JMSListener` subscribes to this JMS Topic and its `onMessage()` method is called once a message is placed onto the JMS Topic. This `onMessage()` method removes the passed key from its local cache. The `JMSCache` class overrides the following methods in `GenericCache`:

put() method

The `put()` method is used to store an object in the local `Cache` and invalidate objects stored against `key` in all other remote caches. This method is declared by the `java.util.Map` interface and has the following signature:

```
public Object put(Object key, Object value);
```

- The `key` parameter specifies the key for the object to store in the cache
- The `value` parameter is the object to store in the cache
- This method returns the previous value associated with the key, or `null` if there was no previous value

putAll() method

The `putAll()` method is used to store in the local cache all objects represented by the `Map keys` passed as parameters. The method also invalidates objects stored against `keys` in all other remote caches. This method has the following signature:

```
public Object put(Map keys);
```

- The `keys` parameter specifies the `Map` of all object keys to be removed.

remove() method

The `remove()` method is used to remove an object from the cache and invalidate objects stored against `key` in all other remote caches. The `remove()` method has two forms, the first is declared by the `java.util.Map` interface, the second declared in the `Cache` interface:

```
public Object remove(Object key);
```

- The `key` parameter specifies the key of the object to remove from the cache

```
public void remove(Set keySet);
```

- The `keySet` parameter specifies a `Set` of keys that identify the objects to remove from the cache
- This method also removes the set of object keys represented by `keySet` in all remote caches.

removeAll() method

The `removeAll()` method is used to remove all objects from the cache. The method also invalidates all objects in other remote caches. The `removeAll()` method is specific to the `Cache` interface and has the following signature:

```
public void removeAll();
```

The following methods are specific to the `com.bankframe.services.cache.JMSCache` class:

initialiseTopic method

The `initialiseTopic()` method does a JNDI lookup on the Connection Factory which is an object that enables JMS clients (the `JMSCache` class) to create JMS connections. A JNDI lookup on the JMS

Topic is also executed and a connection is made from the JMS client to the JMS Topic so the JMS client can publish messages to the topic. The `initialiseTopic()` method has the following signature:

```
public void initialiseTopic();
```

removeDontSend() method

The `removeDontSend()` method is used to remove an object from the local cache. However, this method does not invalidate objects stored in other remote caches. The `removeDontSend()` method has the following signature:

```
public Object remove(Object key);
```

- The `key` parameter specifies the key of the object to remove from the cache

```
public void remove(Set keySet);
```

- The `keySet` parameter specifies a `Set` of keys that identify the objects to remove from the cache
- This method does not remove any objects in remote caches.

removeAllDontSend() method

The `removeAllDontSend()` method is used to remove all objects from the cache. The method does not invalidate any objects in remote caches. The `removeAllDontSend()` method has the following signature:

```
public void removeAll();
```

addValueToCache method

The `addValueToCache()` method allows one to add an object value under an object key to a specific JMS Topic. The method has the following signature:

```
public Object addValueToCache (String topicName, Object key, Object value);
```

- The `key` parameter specifies the key for the object to store in the cache
- The `value` parameter is the object to store in the cache
- The `topicName` parameter is the JMS Topic to publish the message to.
- This method returns the previous value associated with the key, or `null` if there was no previous value

com.bankframe.services.cache.JMSCache.JMSCacheEvent

The `JMSCacheEvent` class is a holder for all the information necessary to notify JMS Listeners to perform some action. A JMS Listener is any MDB (Message Driven Bean) that has subscribed to a JMS Topic and listens for messages placed onto the topic. The class implements the `java.io.Serializable` interface so it can be serialized when being set on the `javax.jms.ObjectMessage` that is sent to the JMS Topic.

com.bankframe.services.cache.JMSListener

The `JMSListener` class is written as an MDB (Message Driven Bean). An MDB subscribes to a JMS Topic and listens for messages placed onto the topic. The MDBs subscription to a particular JMS Topic is declared in its deployment descriptor. The `JMSListener` listens for messages placed onto its subscribed JMS Topic, and removes the appropriate entries from its local cache according to the message received. The `onMessage()` method of the `JMSListener` class provides the behaviour for handling a message from the JMS Topic and determines which entries to remove from the local cache.

Configuring JMS Caching

There are application server specific issues that arise when changing the JMS Topic and JMS Connection Factory names:

Configuring JMS Caching in WebLogic

When changing the JMS Topic name, it is necessary to change the `weblogic-ejb-jar.xml` of the message driven bean `com.bankframe.services.cache.JMSListener` as the JNDI of the topic is also specified here.

Configuring JMS Caching in WebSphere

When changing the 'Listener Port' name for a 'Message Listener Service' in WebSphere Application Server, the EJB Module `WebSphere-MCAMDBs.jar` must be imported into WebSphere Studio Application Developer. The 'Listener Port Name' that the MDB (Message Driven Bean) `com.bankframe.services.cache.JMSListener` subscribes to must be modified in the 'EJB Deployment Descriptor' for the MDB.

com.bankframe.services.cache.CachePolicy

The `CachePolicy` interface defines a means for custom caching policies to be defined and plugged into the `Cache`. The methods defined by the `CachePolicy` interface, and how the `Cache` class interacts with `CachePolicy` objects is described below.

isCacheEntryValid () method

This method is called to determine if an entry is still valid. An entry is not valid if the caching policy determines that the entry should be removed from the cache.

```
public boolean isCacheEntryValid(Object key,Object value);
```

updateCacheEntry() method

This method is called every time an entry in the cache is accessed. This enables the `CachePolicy` object to determine which objects in the cache are being used.

```
public void updateCacheEntry(Object key,Object value);
```

updateCacheEntries() method

This method is called when multiple entries in the cache are accessed. This enables the `CachePolicy` object to determine which objects in the cache are being used.

```
public void updateCacheEntries(Map values);
```

remove() method

This method is called when entries are removed from the cache. This enables the [CachePolicy](#) object to stop tracking objects that are no longer in the cache. This method has two forms; the first is used when a single object is removed, the second is used when a [Set](#) of objects is removed:

```
public void remove(Object key);
```

```
public void remove(Set keySet);
```

removeAll() method

This method is called when all entries are removed from the cache. This enables the [CachePolicy](#) object to reset itself.

cleanup() method

This method is called to determine what entries should be removed from the cache. This method has the following signature:

```
public Set cleanup();
```

- This method returns a [java.util.Set](#) containing the keys of the objects that should be removed.
- If no entries should be removed an empty [Set](#) is returned.
- If all entries should be removed [null](#) is returned.

Cache and Cache Index Interaction

A cache index instance is assigned to a single cache. A cache can have multiple cache indices but is not directly aware of them. The cache indices register their interest in certain keys held in the cache by adding themselves as Cache Listeners to the cache.

The default cache indexer implementation [CacheIndex](#) expects keys and entries in the data cache to be instances of [com.bankframe.bo.DataPacket](#). The cache indexer has the following two main methods for populating the index and retrieving data via the index:

Method	Description
public Object put(Object key)	Adds a record to the index. Using the key value, this method gets the corresponding data from dataCache. The data is then used to build an index key from indexStructure; this method stores the given key for the data in a collection. If the key has not been added before, the index registers as a cache listener for that key. This method takes the key only, to make sure that the data has been put into the dataCache first. This

Method	Description
	method expects the data in the dataCache to be an instance of DataPacket. If it is not, an index key cannot be created and therefore the given key will not be added to the index.
public Collection get(Object data)	This method returns a collection of keys to the dataCache that match the indexKey produced from the given data object. The data given must be a DataPacket and may be a superset of the indexKey. If nothing is found, or the indexKey cannot be created from the data object, an empty set is returned. If a collection of keys is found for the indexKey the cache is asked to confirm that it contains each key. Only confirmed keys are returned in the collection. The added benefit of this approach is that by using containsKey, the cache makes sure that the key does not timeout before eventual retrieval.

Similar to CacheFactory, the CacheIndexFactory is the factory class for instantiating cache indexer classes. The CacheIndexFactory uses settings in the BankframeResource.properties file to determine which class to use. The key is cache.index.<index name>. If no cache index is specified or there is an exception when instantiating the given class, CacheIndex is used. CacheIndex determines the index structure and name of the cache to index from IndexMetaData.

Cache and CachePolicy Interaction

Whenever the state of the `Cache` changes the `Cache` informs the `CachePolicy` object.

- When the `Cache.get()` method is called the `CachePolicy.updateEntry()` method is called
- When the `Cache.put()` method is called the `CachePolicy.updateEntry()` method is called
- When the `Cache.putAll()` method is called the `CachePolicy.updateEntries()` method is called
- When the `Cache.remove()` method is called the `CachePolicy.remove()` method is called
- When the `Cache.removeAll()` method is called the `CachePolicy.removeAll()` method is called
- When the `Cache.clear()` method is called the `CachePolicy.removeAll()` method is called

After the following methods are called the `CachePolicy.cleanup()` method is called:

- `Cache.put()`
- `Cache.putAll()`
- `Cache.remove()`

The `Cache` takes the following actions depending on the return value from the `CachePolicy.cleanup()` method:

- If the returned value is `null` all entries in the `Cache` are removed

- If the returned value is an empty [Set](#) no entries are removed from the [Cache](#)
- Otherwise the specified objects identified by the returned [Set](#) are removed from the [Cache](#).

Creating persistent caches

Creating a persistent cache requires creating a class that implements the [java.util.Map](#) interface and implements all its methods. This class must interact with the persistent store, for example a call to the class' [get\(\)](#) method should read the requested object from the persistent store. For an example of a persistent cache implementation see the [com.bankframe.resource.cache](#) package, and the [com.bankframe.resource.cache.BankFrameResourcePersister](#) class.

Configuring Cache Settings

The [com.bankframe.services.cache.CacheFactory](#) class enables the configuration of all MCA caches to be controlled via the [BankframeResource.properties](#) file. Caching settings need to be added to, and configured in, [BankframeResource.properties](#) for every MCA cache that is to be used. All cache settings are of the format `cache.<CashName>.<ConfigurationParameter>`, for example, `cache.txnPersister.policy`.

Configuring the Cache Class

The fully qualified name of the cache class needs to be added to, and configured in, [BankframeResource.properties](#), for every cache to be used. The class must implement the [com.bankframe.services.cache.Cache](#) interface. If the cache requires a persister it must implement the [com.bankframe.services.cache.PersistentCache](#) interface.

Configuring the Cache Policy

The fully qualified name of the cache policy class needs to be added to, and configured in, [BankframeResource.properties](#), for every cache to be used. This class must implement the [com.bankframe.services.cache.ConfigurableCachePolicy](#) interface. The cache policy for each MCA cache to be used should be configured to one of the following MCA cache policies:

LruCachePolicy

This policy uses a least recently used algorithm to limit the cache to a specified maximum size. This policy has the following configurable settings:

- **maxSize**: This specifies the maximum number of entries permitted in the cache. When this is exceeded the least recently used entries are removed from the cache until the cache size is reduced to the maximum size.
- **thrashAmount**: When the maximum size of the cache is exceeded this policy tries to remove just enough entries to reduce the cache to the maximum size. This setting can be used to force the policy to reduce the number of cache entries to `maxSize` less `thrashAmount`. This means that when the cache size is exceeded and the least recently used entries are removed space will be left for new entries to be added.

TimeoutCachePolicy

This policy removes entries that have not been used for more than a specified period of time. This policy has the following configurable setting:

- **timeout**: This value indicates the maximum time in seconds that an entry can remain in the cache without being used.

PerEntryTimeoutCachePolicy

This policy is similar to the [TimeoutCachePolicy](#) except that each individual entry in the cache can have its own timeout setting. This timeout value needs to be specified programmatically for each entry in the cache by calling the `setTimeout(Object key, long timeout)` or `setTimeout(Set keys, long timeout)` methods of this class. Therefore this policy has no configurable settings.

Configuring the Cache Persister

The fully qualified name of the persister class that is to be used to retrieve data from the data store should be configured for each cache in [BankframeResource.properties](#). This class must implement the `java.util.Map` interface. Some caches do not have a persistent store associated with them and the `persister` setting should be omitted for these caches. Refer to *Siebel Retail Finance MCA Services API Specification* for details of which caches use persisters.

NOTE: This caching persister class is not related to the Financial Process Integrator persister.

Sample Cache Configuration

The following example lists the cache settings that need to be added to, and configured in, [BankframeResource.properties](#) to use the TxnPersister cache. All values are sample values and should be adjusted for your environment.

Cache Class Configuration

```
cache.txnPersister.class=com.bankframe.services.cache.GenericCache
```

Policy Class Configuration

```
cache.txnPersister.policy=com.bankframe.services.cache.LruCachePolicy|TimeoutCachePolicy|PerEntryTimeoutCachePolicy.
```

LruCachePolicy Configuration

If `cache.txnPersister.policy` has been set to `com.bankframe.services.cache.LruCachePolicy` then the following settings need to be added and configured:

```
cache.txnPersister.policy.maxSize=100
```

```
cache.txnPersister.policy.thrashAmount=10
```

TimeoutCachePolicy Configuration

If `cache.txnPersister.policy` has been set to `com.bankframe.services.cache.TimeoutCachePolicy` then the following setting needs to be added and configured:

```
cache.txnPersister.policy.timeout=100
```

PerEntryTimeoutCachePolicy

This policy is not configurable in `BankframeResource.properties` and is set in the code. See `com.bankframe.services.cache.PerEntryTimeoutCachePolicy` in *Siebel Retail Finance MCA Services API Specification* for further information.

Cache Extensibility

The MCA Services caching framework provides the following extensibility features:

- The `CacheFactory` uses the abstract factory pattern. This enables the plugging in of a proprietary `CacheFactory` implementation.
- The `JMSCache` implemented as an interface so that a pluggable implementation can be used.
- The `JMSListener` class creates a Universally Unique Identifier (UUID) class on startup. Therefore each Application Server node in a cluster has a unique identifier. This UUID is sent with each broadcast message, allowing each node to determine if it was the originator of the message.

The following classes support cache extensibility:

```
com.bankframe.services.cache.DefaultCacheFactoryImpl
com.bankframe.services.cache.JMSCacheEventImpl
com.bankframe.services.cache.JMSCacheImpl
```

Configuring Cache Extensibility

Cache extensibility is configured in the file `BankframeResource.properties`. To configure cache extensibility, remove the value for the `jms.cluster.node.id=` setting so that, by default, a UUID will be used to represent the node ID within a cluster. The

`jms.cache.event.impl=com.bankframe.services.cache.JMSCacheEventImpl` setting should be configured to specify the implementation class to use for the `JMSCacheEvent`. The `mca.services.cache.factoryClass` should be configured to specify the `CacheFactory` implementation class to be used. The default value is:

```
com.bankframe.services.cache.DefaultCacheFactoryImpl
```

To configure *BankframeResource.properties* for cache extensibility

- 1 Set the `jms.cluster.node.id` property to null as follows:


```
jms.cluster.node.id=
```
- 2 Configure the property


```
jms.cache.event.impl=com.bankframe.services.cache.JMSCacheEventImpl
```
- 3 Configure the property


```
mca.services.cache.factoryClass=com.bankframe.services.cache.DefaultCacheFactoryImpl.
```


Dynamic Configuration

This topic describes the dynamic configuration framework in MCA Services. MCA Services provides its own framework for reading `.properties` files. The MCA framework periodically refreshes the in memory cache from the disk file. The end result is that it is possible to make changes to MCA's configuration without requiring the application server to be restarted. However, dynamic configuration does have some performance overheads, primarily because methods have to be synchronized for reloading.

Configuring `com.eontec.mca.bankframeresourcebundle`

By default, the dynamic configuration is not used. To enable dynamic configuration, set the Java system property `com.eontec.mca.bankframeresourcebundle` to `com.bankframe.services.resource.BankFrameResourceBundle`. The default is `com.bankframe.services.resource.NoReloadBankFrameResourceBundle`. The default implementation does not reload property values and the methods for getting property values are not synchronized.

Grouping properties

The standard Java APIs provide no means for grouping related configuration information, the MCA framework adds support for this facility, allowing only the configuration information relating to a particular functional area to be retrieved. How this facility works is explained below.

`com.bankframe.services.resource`

This package defines the MCA dynamic configuration framework; it contains the following classes and interface:

<code>BankFrameResource</code>	Defines the methods that all <code>BankFrameResource</code> implementations must provide
<code>BankFrameResourceSubset</code>	This class provides the functionality for grouping related properties
<code>BankFrameMCAResource</code>	This class provides methods for accessing the standard <code>BankframeResource.properties</code> file
<code>BankFrameResourceBundle</code>	This class implements the <code>BankFrameResource</code> interface and provides functionality for reading data from <code>.properties</code> files.
<code>BankFrameResourceFactory</code>	This class creates instances of <code>BankFrameResource</code> for the specified URL. It will use the Java system property <code>com.eontec.mca.bankframeresourcebundle</code> to determine which resource bundle to use. <code>NoReloadBankFrameResourceBundle</code> is the default value.

<code>NoReloadBankFrameResourceBundle</code>	Performance optimized resource bundle class. Default bundle in framework.
<code>NoReloadBankFrameResource</code>	Performance optimized resource class. Default resource in framework.
<code>ResourceLocator</code>	This class provides methods for manipulating files in the Java class path

BankFrameResource

This interface defines the following methods:

get()

This method gets a value from the resource. This method has the following signature:

```
public Object get(String key);
```

- The `key` parameter specifies the name of the value to retrieve
- The value is returned if found, or `null` if the value is not found

getString()

This method gets a value and converts it to a String. This method has the following signature:

```
public String getString(String key);
```

- The `key` parameter specifies the name of the value to retrieve
- The value is returned if found, or `null` if the value is not found

getSubset()

This method gets a subset of values whose keys all begin with the specified prefix. This method has the following signature:

```
public BankFrameResource getSubset(String prefix);
```

- The `prefix` parameter specifies the prefix that the subset starts with
- A `BankFrameResource` instance is returned containing the requested subset. An empty subset is returned if no values with the specified prefix could be found.

put()

This method adds or updates a value in the resource. This method is used for changing or adding configuration values. Note that not all implementations support this method. This method has the following signature:

```
public Object put(String key, Object value);
```

- The `key` parameter specifies the name of the value

- The `value` parameter contains the value to be stored
- The previous value associated with the specified key is returned, or `null` if the key had no previous association.

remove()

This method removes a value from the resource. Note that not all implementations support this method. This method has two forms:

```
public Object remove(String key);
```

- The `key` parameter specifies the name of the value to remove
- The removed value is returned, or `null` if the key did not exist.

```
public void remove(Enumeration keys);
```

- The `keys` parameter specifies an `Enumeration` of one or more keys to remove.

removeAll()

This method removes all values from the resource. Note that not all implementations support this method. This method has the following signature:

```
public void removeAll();
```

removeSubset()

This method removes a subset of values from the resource. Note that not all implementations support this method. This method has the following signature:

```
public void removeSubset(String prefix);
```

The `prefix` parameter specifies the prefix that the subset starts with.

keys()

This method returns an `Enumeration` of key values. This method has the following signature:

```
public Enumeration keys();
```

BankFrameResourceSubset

This class implements the `BankFrameResource` interface and provides a standard mechanism for `BankFrameResource` implementations to implement support for subsets. A subset of properties is defined as one or more properties that start with the same prefix, for example:

```
ldap.default.java.naming.provider.url=ldap://localhost:389
```

```
ldap.default.java.naming.factory.initial=com.sun.jndi.ldap.LdapCtxFactory
```

This class provides the same methods as `BankFrameResource`, in addition it has a single constructor:

BankFrameResourceSubSet()

This constructor creates a subset for the specified prefix. This method has the following signature:

```
public BankFrameResourceSubSet(String prefix, BankFrameResource parent);
```

- The `prefix` value specifies the prefix that all members of the subset begin with
- The `parent` value specifies the resource which contains this subset

BankFrameMCAResource

This class provides methods for easily retrieving values from the standard `BankFrameResource.properties` file. This class replaces the deprecated `com.bankframe.BankframeResource` class. This class has the following methods:

getString()

This method gets the value of the specified property. This method has the following signature:

```
public static String getString(String key);
```

- The `key` parameter specifies the name of the property to retrieve
- The specified property is returned or `null` if the value could not be found

getKeys()

This method returns an `Enumeration` containing all the keys in the `BankframeResource.properties` file. This method has the following signature:

```
public static Enumeration getKeys();
```

getSubset()

This method returns a subset of keys in the `BankframeResource.properties` file. This method has the following signature:

```
public static BankFrameResource getSubset(String prefix);
```

- The `prefix` parameter specifies the prefix that the subset starts with
- A `BankFrameResource` instance is returned containing the requested subset. An empty subset is returned if no values with the specified prefix could be found.

BankFrameResourceBundle

This class provides an implementation of the `BankFrameResource` interface that reads data from `.properties` files. The public methods of this class are the same as those of the `BankFrameResource` interface. This class has the following constructor:

```
public BankFrameResource BankFrameResourceBundle(URL resourceUrl);
```

- The `resourceUrl` parameter specifies the URL of the `.properties` file to read
- This class reads the contents of the `.properties` file the first time a property is requested.

- It caches the entire contents of the `.properties` file for a specified time period. When that time period has passed it re-reads the `.properties` file. This enables changes to the `.properties` file to be detected.
- The time period can be configured as follows:

The time period is specified by adding a property named: `resource.cache.refreshInterval` to the `.properties` file. This property must be an integer indicating the number of seconds in the time period, for example:

```
resource.cache.refreshInterval=120
```

If the `resource.cache.refreshInterval` property is not present in the resource file then the file will be refreshed every 15 minutes.

If the `resource.cache.refreshInterval` property has a value of -1 then the resource file will never be refreshed (This means changes made to the resource file will not be detected).
- This class provides read only access to `.properties` files therefore it does not support the `remove()`, `put()` or `clear()` methods of `BankFrameResource`.

BankFrameResourceFactory

This class is used to create instances of `BankFrameResource` for a specific URL.

getInstance()

This method has the following signature:

```
public static BankFrameResource getInstance(String resourceName);
```

- This method creates a `BankFrameResource` instance for the specified `.properties` file
- The `.properties` file must be in the class path
- The implementation of this method creates an instance of the `BankFrameResourceBundle` class to read from the specified `.properties` file

ResourceLocator

This class provides utility methods for locating resources in the class path, and for accessing resource files. This class contains the following methods:

getClassInClassPath()

This method gets the `URL` for the specified class. This method has three forms:

```
public static URL getClassInClassPath(String className);
```

- The `className` parameter specifies the name of the class
- The `URL` of the class will be returned or `null` if it is not found in the class path
- `public static URL getClassInClassPath(String className, Locale locale);`
- The `className` parameter specifies the name of the class

- The `locale` parameter specifies the locale specific version of this class to locate
- The `URL` of the class will be returned or `null` if it is not found in the class path
- `public static URL getClassInClassPath(Class clazz,String className,Locale locale);`
- The `clazz` parameter specifies the `Class` instance to use to search the class path
- The `className` parameter specifies the name of the class
- The `locale` parameter specifies the locale specific version of this class to locate
- The `URL` of the class will be returned or `null` if it is not found in the class path

getResourceInClassPath()

This method gets the `URL` for the specified resource file. This method has three forms:

```
public static URL getResourceInClassPath(String resourceName);
```

- The `resourceName` parameter specifies the name of the resource
- The `URL` of the resource will be returned or `null` if it is not found in the class path

```
public static URL getResourceInClassPath(String resourceName, locale locale);
```

- The `resourceName` parameter specifies the name of the resource
- The `locale` parameter specifies the locale specific version of this resource to locate
- The `URL` of the resource will be returned or `null` if it is not found in the class path

```
public static URL getResourceInClassPath(Class clazz,String resourceName,Locale locale);
```

- The `clazz` parameter specifies the `Class` instance to use to search the class path
- The `resourceName` parameter specifies the name of the resource
- The `locale` parameter specifies the locale specific version of this resource to locate
- The `URL` of the resource will be returned or `null` if it is not found in the class path

getInputStream()

This method gets an `InputStream` for the specified `URL`. This method has the following signature:

```
public static InputStream getInputStream(URL url) throws IOException;
```

- The `url` parameter specifies the `URL` of the resource
- The `InputStream` for the `URL` is returned or an `IOException` is thrown if the resource cannot be accessed

getOutputStream()

This method gets an `OutputStream` for the specified `URL`. Note that the resource may be read only, in which case calling this method will result in an `IOException` being thrown. This method has the following signature:

```
public static OutputStream getOutputStream(URL url) throws IOException;
```

- The `url` parameter specifies the `URL` of the resource
- The `OutputStream` for the `URL` is returned or an `IOException` is thrown if the resource cannot be accessed

getLastModified()

This method returns the time (in milliseconds) that the resource was last modified. This method has the following signature:

```
public static long getLastModified(URL url);
```

- The `url` parameter specifies the `URL` of the resource
- The time of last modification is returned or zero if an error occurs

isReadOnly()

This method checks if the specified resource is read only. This method has the following signature:

```
public static boolean isReadOnly(URL url);
```

- The `url` parameter specifies the `URL` of the resource
- This method returns `true` if the resource is read only, `false` otherwise

Using the Dynamic Configuration Framework

Accessing BankframeResource.properties

The values in `BankframeResource.properties` are read using the static methods in `com.bankframe.services.resource.BankFrameMCAResource`.

The following sample implementation illustrates how to read a single value:

```
String ldapServer =
    BankFrameMCAResource.getString("ldap.default.java.naming.provider.url");
```

The following sample implementation illustrates how to read a single value:

```
BankFrameResource ldapSubset = BankFrameMCAResource.getSubset("ldap.default");
```

Working with subsets

A subset is a set of values that all start with the same prefix. Prefixes are delimited using the `'.'` character. Below is an example of a subset:

```
ldap.default.java.naming.provider.url=ldap://localhost:389

ldap.default.java.naming.factory.initial=com.sun.jndi.ldap.LdapCtxFactory
```

```

ldap.default.java.naming.security.authentication=simple

ldap.default.java.naming.security.principal=cn=bankframe,dc=eontec,dc=com

ldap.default.java.naming.security.credentials=bankframe

```

This subset can be retrieved by calling `BankFrameMCAResource.getSubset("ldap.default")`. The returned subset will contain all the values starting with 'ldap.default', however the prefix: 'ldap.default' will be removed from the names of the values, so the subset above will contain:

```

java.naming.provider.url=ldap://localhost:389

java.naming.factory.initial=com.sun.jndi.ldap.LdapCtxFactory

java.naming.security.authentication=simple

java.naming.security.principal=cn=bankframe,dc=eontec,dc=com

java.naming.security.credentials=bankframe

```

Getting a single value from a subset

The following code sample creates a subset:

```
BankFrameResource ldapSubset = BankFrameMCAResource.getSubset("ldap.default");
```

To retrieve a key from this subset, its name, less the 'ldap.default' prefix, needs to be supplied, for example to retrieve the key whose full name is 'ldap.default.java.naming.provider.url' the following code could be used:

```
String providerUrl = ldapSubset.getString("java.naming.provider.url");
```

Retrieving a subset of a subset

Subsets can be nested within each other, for example to get the security settings in the example above the following code could be used:

```
BankFrameResource securitySubset = ldapSubset.getSubset("java.naming.security");
```

Accessing arbitrary resource files

The `com.bankframe.services.resource.BankFrameMCAResource` class provides a means to read settings from the `BankframeResource.properties` file. To access properties in other files use the `com.bankframe.services.resource.BankFrameResourceFactory` class.

Accessing a .properties file in the Java class path

To access a file called 'FrontEnd.properties' which is somewhere in the Java class path use the following code:

```

BankFrameResource resource = BankFrameResourceFactory.getInstance("FrontEnd");

String someProperty = resource.getString("someProperty");

```


Note that you do not supply the filename extension of the `.properties` file.

Accessing a `.properties` file not located in the Java class path

To access a file not located in the class path you must provide a complete [URL](#) to the resource file.

To access a file on a http server. Below is an example of accessing a file stored on an http server

```
BankFrameResource resource = BankFrameResourceFactory.getInstance(new
URL("http://webserver/SomePropertyFile.properties"));

String someProperty = resource.getString("someProperty");
```

To access a file on a local file system

```
BankFrameResource resource = BankFrameResourceFactory.getInstance(new
URL("file:///some/path/to/SomePropertyFile.properties"));

String someProperty = resource.getString("someProperty");
```

To access a file in a .JAR on a local file system

```
BankFrameResource resource = BankFrameResourceFactory.getInstance(new
URL("jar:///some/path/to/Some.jar/SomePropertyFile.properties"));

String someProperty = resource.getString("someProperty");
```

Configuring the refresh interval

To configure how often a resource file should be re-read add the following key to the resource file:

```
resource.cache.refreshInterval
```

This value is specified in seconds. If this setting is not specified in the resource file then the default refresh interval of fifteen minutes will be used.

Deprecated Class

The class `com.bankframe.BankframeResource` has been deprecated and the `com.bankframe.services.resource.BankFrameMCAResource` class should be used instead.

To provide backwards compatibility with existing code `com.bankframe.BankframeResource` has been retrofitted to call the methods of `com.bankframe.services.resource.BankFrameMCAResource`.

6

Administering MCA Services

This chapter covers the MCA Services Administration Application and includes the following topics:

- Administering MCA Routes
- Administering MCA Sessions
- Administering MCA Users and Groups
- Using the MCA Monitor Servlet

Launching the MCA Services Administration Application

The MCA Services Administration Application is a browser-based servlet (`ServiceServlet`) which is launched at the following URL:

`http://<servername>:<port>/BankFrameMCA/ServiceServlet`

`<servername>` is the name of the machine on which the Banking Application is deployed.

`<port>` is the port number that the application server is listening on. This is typically 9080 for WebSphere and 7001 for WebLogic.

Administering MCA Routes

Every time a new route or service is developed, it must be added to the list of MCA Routes. This list of routes is stored in the `ROUTES` table in the database. The MCA Services Administration Application can be used to administer Routes, instead of running SQL on the services table. If the Session Managed option is selected then the Route EJB is checked for a SessionID before every request is fulfilled. For more information on MCA Routes see How MCA Services Routing Works.

To create an MCA Route

- 1 Select Configure Route > Create a New Route.
- 2 Enter a `REQUEST_ID`, JNDI Name and description for the new Route.
- 3 Select Session Managed if the Route EJB should be session managed.
- 4 Click Submit.

Administrating MCA Sessions

The MCA Services Administration Application can be used to manage currently active and expired sessions. The Session ID, User ID, and last time the user accessed the system, are displayed for each Session. For more information on MCA Services Session Management see Components of MCA Services Session Management.

To view a list of current sessions

- Select Configure Session > List all Current Sessions.

To remove expired sessions

- Select Configure Session > Remove Expired Sessions.

To remove all sessions

- Select Configure Session > Remove All Sessions.

To delete a session

- Select Configure Session > Delete a Specific Session. Enter the Session ID of the Session to be deleted.

Administrating MCA Users and Groups

The MCA Services Administration Application can be used to manage MCA users and groups. For more information on MCA Services users and groups see Access Control. This topic contains the following sub-topics:

- Administrating MCA Users
- Administrating MCA Groups

Administrating MCA Users

The following tasks list the administration operations that can be executed for MCA users.

To view a list of all registered MCA users

- Select Administer Users and Groups > View All Users.

To create a new MCA user

- Select Administer Users and Groups > Create a New User.

To delete an MCA user

- Select Administer Users and Groups > Delete a User.

To view an MCA user's permissions

- Select Administer Users and Groups > View a User's Permissions.

To assign or extend an MCA user's permissions

- Select Administer Users and Groups > Assign/Extend User Permissions.

To remove an MCA user's permissions

- Select Administer Users and Groups > Remove User Permissions.

To assign an MCA user to a group

- Select Administer Users and Groups > Assign User to Group.

To remove an MCA user from a group

- Select Administer Users and Groups > Delete From Group.

Administrating MCA Groups

The following tasks list the administration operations that can be executed for MCA groups.

To view a list of all registered MCA groups

- Select Administer Users and Groups > View All Groups.

To create a new MCA group

- Select Administer Users and Groups > Create a New Group.

To delete an MCA group

- Select Administer Users and Groups > Delete Group.

To view an MCA group's members

- Select Administer Users and Groups > View a Group's Members.

To view an MCA group's permissions

- Select Administer Users and Groups > View Group Permissions.

To assign or extend an MCA group's permissions

- Select Administer Users and Groups > Assign/Extend Group Permissions.

To remove an MCA group's permissions

- Select Administer Users and Groups > Remove Group Permissions.

To remove an MCA group's permissions

- Select Administer Users and Groups > Remove Group Permissions.

Using the MCA Monitor Servlet

MCA processes DataPackets from various channels, and routes them to the correct Financial Component, based on a REQUEST_ID specified in the DataPackets. The Monitor Servlet is a HTML utility provided with MCA to test the transmission of DataPackets. The MonitorServlet can generate DataPackets of varying formats, forward them to the correct Financial Component, and receive the response DataPacket(s) from the Financial Component.

To test the transmission of DataPackets

- 1 Select MCA Monitor.
The DataPacket fields are displayed.
- 2 If additional DataPacket fields are required select Add a new field.
- 3 Enter the DataPacket values. Sample MCA DataPacket values are provided in the following table:

DATA PACKET NAME	LOGON REQUEST
OWNER	Oracle
REQUEST_ID	50000
userPassword	AAAAA
USER_ID	Kds
LOGON	true

- 4 Click Update to update the DataPacket values.
- 5 Click Send DataPacket.
On successful transmission the response DataPacket for the Financial Component displays.

7

Glossary: MCA Services

DataPacket

A [DataPacket](#) is a Siebel Retail Finance class through which MCA Services organizes data that is passed between clients and Financial Components. It provides a standard format for all data used within SRF applications, which greatly simplifies the task of passing data from clients to Financial Components and from Financial Components to other Financial Components. Information stored in [DataPackets](#) can be transformed into a string representation or a serialized Java Object. This enables [DataPackets](#) to be easily transmitted over various protocols. There are three required keys in every [DataPacket](#): [DATA_PACKET_NAME](#), [OWNER](#) and [REQUEST_ID](#). All keys in a [DataPacket](#) are unique within that [DataPacket](#) and identify corresponding data, as in a hashtable.

DPTP

[DataPacket](#) Transmission Protocol.

Dynamic Configuration

Standard Java APIs for reading configuration information from [.properties](#) files require the application server to be re-started to pick up any configuration changes made. The MCA Services Dynamic Configuration framework enables changing MCA Service's configuration and enabling these changes to take effect without having to re-start the application server. The Dynamic Configuration framework re-reads the [.properties](#) file at set intervals, and the interval period is configurable in [BankframeResource.properties](#).

Financial Component

A stateless session EJB. All Siebel Financial Components implement the [com.bankframe.ejb.ESession](#) interface.

Financial Process Integrator

The Financial Process Integrator provides the facility in MCA Services to map data from SRF Entity Beans and Financial Components to host transactions.

Free Service

A Financial Component that does not involve a user logged into SRF, that is, an EJB session bean that is not session managed, for example, the [GenerateRandomNumbers](#) service bean, which determines which digits of the end-user's password to request (for example, first, third and last) when the user is logging onto SRF applications.

Internationalization Framework

The MCA Internationalization Framework enables messages to be localized on the client-side, supporting localization on a per-client and per-locale basis. The data that needs to be localized is

passed to the client, in addition to the data required for the localization, which is held in resource bundles.

MCA Services

Multi Channel Architecture Services: an infrastructure that can support the delivery of uniform services to all channels, and be able to incorporate new channels as they emerge. It is implemented using open industry standards to facilitate integration with diverse channel technologies.

Metadata

MCA Services Metadata is the set of data that maps SRF Entity Beans to host transactions.

Module

An Oracle Siebel Retail Finance Module is a pre-assembled solution set of Siebel Financial Components, for example, Branch Teller and Internet Banking.

Sample Screen Code and Process Templates

Sample Screen Code and Process Templates are referred to in this documentation as "MCA Extension Point", "Implementation Layer Code", "Swing Front End Code" and "JSP Front End Code".

Store and Forward

When the Financial Process Integrator fails to send a transaction to the host, the host is marked as offline and the transaction is stored for later forwarding. When a host is marked as offline it will remain marked as such for a specified period (this period is configurable). During this period no further attempts will be made to send transactions to that host, all transactions will instead be stored (except for transactions that are not permitted to be stored, these will instead result in an exception being thrown). When the time period has expired the forwarding mechanism will try to send the first entry on the queue to the host. Only data for update to the host is stored, it will not store data retrieved from the host.