Brightware™

Integration Development Kit Guide

Version 8.1.4

edocs

Trademark, Copyright, and Patent Acknowledgements

Version Date: November 24, 2004

Table of Contents

Chapter 1

About This Guide

This manual describes how to use the Integration Development Kit (IDK). It is intended for anyone involved in using the IDK. It is assumed that anyone using this guide is an experienced Java and JSP programmer.

How this Guide is Organized

Chapter 1, "About This Guide" provides general information.

Chapter 2, "Overview and Getting Started" provides an overview of the IDK, and parameters for its use.

Chapter 3, "The Event API and Database Interface" provides general usage information for the IDK.

Chapter 4, "Workflow (Agent Desktop) Integration" provides an introduction and specifics for integrating Agent Desktop with other complementary products, third-party applications, and custom components (without the need to modify the core Brightware product).

Appendix A, "Understanding Event Sequences" describes and clarifies event sequences.

Appendix B, "Example Event Handler" describes an example event handler.

Related Documentation

For more information about Brightware please see these documents included in PDF format on the Installation CD:

- *Installation Guide*
- *Agent Guide*
- *Contact Center Console Guide*
- *Knowledge Engineer Handbook*
- *Implementation Guide*
- *Analytics Overview Guide*
- *DB Administrator Guide*
- *Report Developer Guide*

Conventions

The following typographic conventions are used in this document:

- Items that you are instructed to click or select, such as button names and hyperlinks, are bold:
  - Select **Add Response**.
  - Click the **OK** button.
- Documents, headings, and chapter titles are italicized:
  - "Refer to the *Reference Manual* for more information."
- Notes are flagged along the left margin:

This icon indicates noteworthy information.

- Cautions are flagged along the left margin:

This icon indicates critical information.

- Programming code and system messages appear in a fixed-width font:

```
Set-request-condition (<condition>)
```

- Hyperlinks and Cross References - If viewing a document online, you can navigate through it using hyperlinks, which appear in blue text, and cross references. Although not displayed in blue, the Table of Contents and Index entries are also hyperlinks. Cross references are specific page number references. Click the page number to navigate to that page:
  - Refer to "Technical Support", on page 3.
- The term Type usually refers to typing information on your keyboard:
  - Type the number of decimal places you want displayed.
- The term Enter typically refers to the "Enter" key on your keyboard:
  - Type the number of decimal places you want displayed and press the **Enter** key.
- When a directory path is given, the hard drive letter is omitted since it is unknown what hard drive the system is installed on. Only the default install path is supported:
  - Documents are available under edocs\Brightware\docs\.

Technical Support

Assistance is available from edocs Technical Support.

	North America	Europe
Telephone	(508) 652-8400	+44.20.8956.2673
Hours of operation	8:30 AM – 8:00 PM Eastern Time, Monday - Friday	09:00 – 17:00 GMT Monday -Friday
E-mail address	support@edocs.com	support@edocs.com
Please consult your Warranty and Maintenance Attachment for the terms of your technical support.		

Before you call Technical Support, please have the following information available for the representative:

- Your company name.
- Version of software currently being used.
- Exact error message.
- Where the error occurred.
- Exact path for recreation of the error.

Chapter 2

Overview and Getting Started

Introduction

This document explains the process of creating a third-party integration using the Brightware IDK.

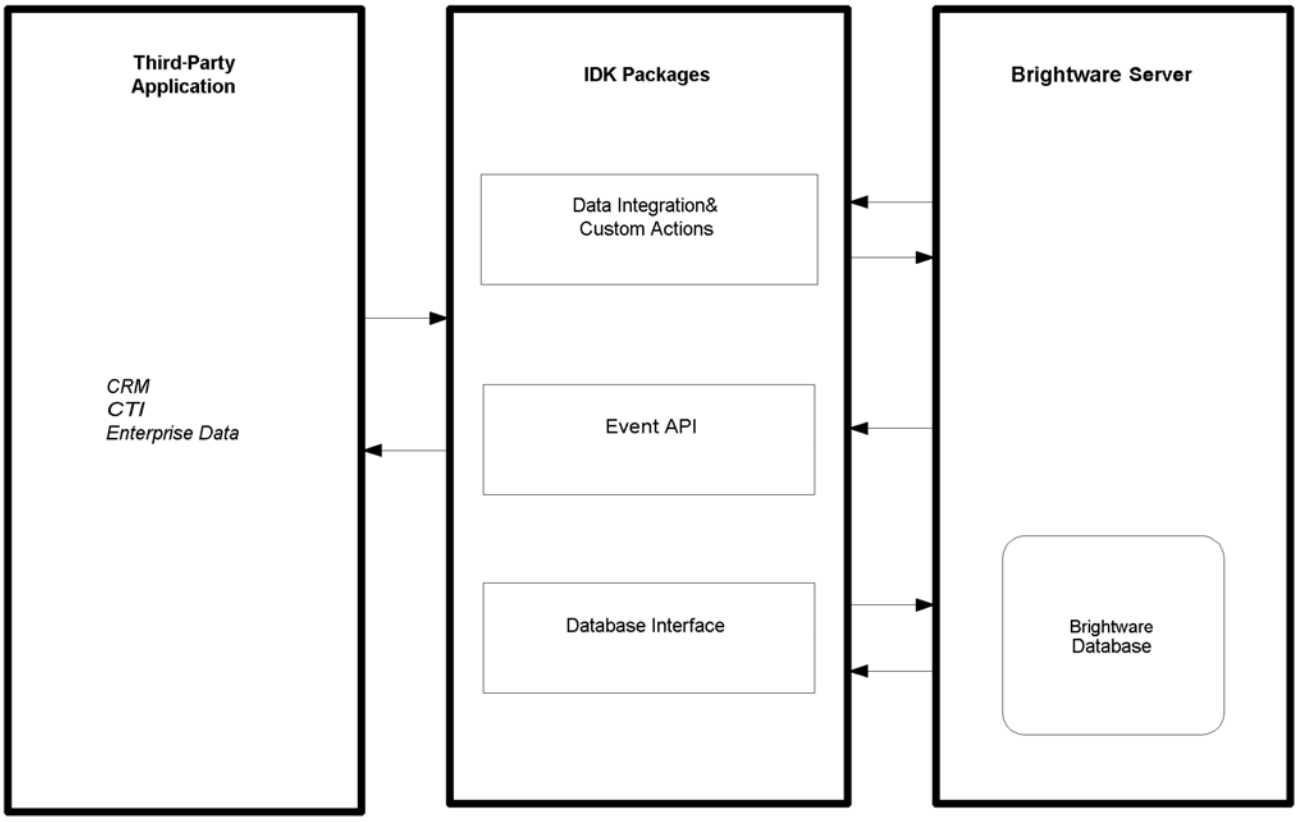

Figure 2-1. Brightware IDK

The Brightware IDK provides a comprehensive software development kit for creating applications that integrate Brightware applications with external third-party systems like CRM or CTI applications. Figure 2-1, “Brightware IDK” shows the typical third-party integration where the Brightware applications are running on an application server. These applications communicate to the third-party system through three main facilities: Event notification, the Database Interface (DBI), and Custom Action / Data Integration Plugins. While Figure 2-1 only shows one third-party system, concurrent integration of many such systems can be supported through the same means.

IDK Components

- The Database Interface (DBI) provides a completely Java Bean based API that provides access to Brightware data objects such as Agents, Queues, and Messages. See "Using the Database Interface", on page 10 for an explanation of the Database API.
- The Events API includes events to notify the partner applications of processes occurring in the components, for example placing an email message in a queue or changing the state of an Agent from online to offline. You code external responses as an event listener, which determines the response to the corresponding event. See "Using the Events API", on page 15 for an explanation of using the Events API.
- The Custom Action / Data Integration Plugins API provides direct interaction with the Intelligence Engine via extension of NLP actions, and extraction / reaction to operations. Unlike the Event API, the Plugins are synchronous to the operations of the Brightware Server.

Development Process Overview

The recommended process of integration between Brightware and third-party applications is as follows:

Installation

The Brightware Server installer installs the IDK and supporting files, providing an environment suitable for IDK development.

Development

Once the IDK is installed, you can begin writing and testing your code. See Chapter 3, "The Event API and Database Interface", on page 9 for an explanation of how to develop using the IDK.

Final Integration

Final integration involves installing the IDK on the production server, along with the integration classes you have developed.

Environment

IDK development should be done on a development computer that is networked to a computer running the Brightware Server. See the *Installation Guide* for installation and configuration instructions.

IDK development requires that Sun's Java Development Kit version 1.3 be installed on the development computer. While the IDK can be installed on the Brightware Server, it is recommended that the IDK be on a separate computer.

Development should never occur on a production server or database!

Installation

The IDK is installed by default during any installation. It is installed from the Brightware installer. To install the Brightware IDK refer to the *Installation Guide*.

Before development can begin, the installed Brightware files must be set in your classpath. In the list below, the `installationDir%` directory represents the directory where you installed Brightware:

- `installationDir%\lib\common\common.jar`
- `installationDir%\lib\common\components.jar`
- `installationDir%\lib\common\components-boot.jar`
- `installationDir%\lib\common\commons-collections.jar`
- `installationDir%\cluster\lib\common\dependent.jar`
- `installationDir%\cluster\lib\common\provider.jar`
- `installationDir%\cluster\lib\common\bcmf.jar`
- `installationDir%\cluster\lib\common\domain.jar`
- `installationDir%\cluster\lib\common\log4j.jar`
- `installationDir%\cluster\lib\common\server.jar`
- `installationDir%\lib\common\xerces.jar`
- `installationDir%\wlserver6.1\lib\weblogic.jar`
- `installationDir%\lib\integration\integration.jar`
- `installationDir%\jdk131\jre\lib\rt.jar`

Once the files are added to your classpath, you are ready to begin development.

Code Sample 2-1. Sample Classpath

```
C:\edocs\Brightware\lib\common\common.jar;C:\edocs\Brightware\lib\
common\components.jar;C:\edocs\Brightware\lib\common\components-
boot.jar;C:\edocs\Brightware\lib\common\commons-
collections.jar;C:\edocs\Brightware\cluster\lib\common\dependent.j
ar;C:\edocs\Brightware\cluster\lib\common\provider.jar;C:\Pedocs\B
rightware\cluster\lib\common\bcmf.jar;C:\edocs\Brightware\cluster\
lib\common\domain.jar;C:\edocs\Brightware\cluster\lib\common\log4j
.jar;C:\edocs\Brightware\cluster\lib\common\server.jar;C:\edocs\Br
ightware\lib\common\xerces.jar;C:\edocs\Brightware\wlserver6.1\lib
\weblogic.jar;C:\edocs\Brightware\lib\integration\integration.jar;
C:\edocs\Brightware\jdk131\jre\lib\rt.jar
```

Chapter 3

The Event API and Database Interface

Introduction

This chapter covers two components: The Events API and the Database Interface (DBI). The Events API is used when coordination between the Brightware application and third-party application is required, while the database interface can be used when access to Brightware data is needed. The sections below explain how to use both components.

Additional information and an example to aid your integration has been provided in the appendices:

- Appendix A, “Understanding Event Sequences”
- Appendix B, “Example Event Handler”

IDK Java Packages

The IDK Events API and DBI includes the following packages:

Table 3-1. Brightware IDK Packages

Package	Description
`com.firepond.dbi`	Database Interface (DBI) classes
`com.firepond.event`	base event classes
`com.firepond.event.agent`	agent-related event classes
`com.firepond.event.group`	group-related event classes
`com.firepond.event.queue`	queue-related event classes
`com.firepond.event.request`	request-related event classes
`com.firepond.event.sponsor`	sponsor-related event classes
`com.firepond.example`	example package explained in Appendix B

JavaDocs are installed with the IDK Events API and DBI. They can be accessed via the Start menu. Click the **Start** menu and select **Programs**. Then select **Brightware**, then **Documentation**, **Java Docs**, and finally click on **IDK** to display the JavaDocs.

When running Brightware DBI or Event application make sure to set the system property com.firepond.provider.library.url to *<Brightware Installation Dir>/platform\weblogic-platform.jar*. For example: **java -Dcom.firepond.provider.library.url=file:\\\<Brightware Installation Dir>\platform\weblogic-platform.jar myDBIapp.class**

Using the Database Interface

The DBI is a JavaBean-based interface to the Brightware system. It consists of Value Beans and Helper Beans. Value Beans are used to store the value of a stored object's representation, while the Helper Beans perform utility functions, including returning Value Beans when required. Each Value Bean class has a corresponding helper and key class. The key class is simply the unique identifier for a Value Bean. It is used by the users of the system as well as internally for accessing specific Value Beans.

Because the DBI package is a pure Java API, the steps required to use it are the same as those required to use any Java package: install the IDK and set the classpath, create an instance of the DBI, then write and execute your own implementation of the DBI classes.

Task 1: Install the IDK and set your classpath

Follow the instructions in Chapter 2, "Overview and Getting Started", on page 5 to install the IDK files and set your classpath.

Task 2: Create an Instance of the DBI

In order to use the DBI you must first initialize it. This is accomplished by creating a new instance of the DBI class and supplying the constructor the proper values for `dnsName` and `port`. This will create a new instance of the DBI class as well as initialize the rest of the system. You do not need to store the newly created DBI instance, as it will never be used by you again. The code sample below performs this task:

Code Sample 3-1. Initializing the DBI Class

```
DBI dbi = null;
dbi = new DBI("localhost", "7001");
```

In the example above, `localhost` should be replaced with the name of the server where the Brightware Server is running and the `port` represents the port on which the Brightware Server is listening, which will most likely be the same value as shown in the sample (7001).

Task 3: Implement the DBI Classes

Once the DBI is initialized (and the server is up and running properly), all classes should be fully operational. What's next depends on what you need to do. Consult the JavaDocs for a complete listing of all classes, attributes, and methods provided with the DBI.

In order to use the DBI you must import the `com.firepond.dbi` package in your code using the following line:

```
import com.firepond.dbi.*;
```

The following examples illustrate some of the tasks your implementation might perform.

Example 1: Add an Agent

This example will add an agent profile to the Brightware database. The comments explain each step as it occurs.

In order to add an agent you must first create an instance of the `AgentBean` (a value bean) and fill it with all required columns.

Code Sample 3-2. Adding an Agent

```
AgentBean anAgentBean = null;
AgentHelperBean anAgentHelperBean = null;
// Create an empty instance of the value bean class
anAgentBean = new AgentBean();
// Fill the required fields
anAgentBean.setUserName("example agent");
anAgentBean.setFirstName("Example");
anAgentBean.setMiddleName("C.");
anAgentBean.setLastName("Agent");
anAgentBean.setEmailAddress("example.c.agent@somecompany.com");
anAgentBean.setNote("This is an agent created through the DBI");
// Create a helper and execute the add
anAgentHelperBean = new AgentHelperBean();
try {anAgentHelperBean.add(anAgentBean);
catch (HelperException e) {
System.out.println("Helper exception : " + e.toString());
}
catch (InvalidObjectStateException e) {
System.out.println("you didn't fill in all the required fields." +
e.toString());
}
```

The `add` method will fill in the primary key for the newly added agent so you can find it again if you need to refresh the state.

Example 2: Retrieve Data

All helper beans have getter methods to get instances of the value beans. Lets go through an example of getting some data from the system. To expand on the example above, this example will retrieve a vector of the agent bean created in the first example.

Code Sample 3-3. Retrieve Agent Data

```
// Declare a vector to store the results
Vector      aVector = null;
Enumeration anEnum  = null;
// Ask the helper bean to return all the agents in the system
aVector = anAgentBeanHelper.getAgents();
anEnum = aVector.elements();
while (anEnum.hasMoreElements()) {
   anAgentBean = (AgentBean) anEnum.nextElement();
/*
* Do something with the AgentBean instance
*/
}
```

All helper beans work in this same way: they return a vector of the beans that are requested. If no beans are found, an empty Vector will be returned. All helper beans check the validity of the value bean passed to them if you are trying to add, update, or remove a bean and will throw an `InvalidObjectStateException` if something is incorrect and it can't save the bean contents.

Example 3: Searching with Keys

These methods require you to pass the key class from the desired bean. To accommodate this feature, all Value Beans have a method on them to get the key class from the bean instance. This allows you to navigate through the object hierarchy without having to create a key instance yourself.

Continuing with the example above, if you wanted to find all the requests associated with one of the agents returned by the `getAgents()` method that was called above you could do the following:

Code Sample 3-4. Searching with Keys

```
Vector requests = null;
RequestHelperBean aRequestHelperBean = new RequestHelperBean();
requests = aRequestHelperBean.getRequestByAgent(anAgentBean.getKey());
```

This will return a Vector of `RequestBeans` that are assigned to the agent (`anAgentBean`).

Example 4: Search Using Status

Some helper methods require a list of states or status codes in order to get the desired beans returned. For example the `RequestHelperBean` allows you to find requests by their current state. To find requests matching more than one state you logically OR the states together to get the desired output. For example, the code below will retrieve all the requests that are open and all the requests that are pending in the system.

Code Sample 3-5. Specifying Requests Types

```
aVector = aRequestHelperBean. getRequests(
RequestBean.OPEN | RequestBean.PENDING);
```

All methods that can accept this type of parameter are marked in the JavaDoc.

Example 5: Search Using Actor Key

The `RequestHelperBean` has some methods that allow you to assign requests to queues and agents. These all accept an `ActorKey` as their initial parameter. An `ActorKey` represents the Agent or Supervisor that is performing the action requested. Code Sample 3-6,"Using the ActorKey" below demonstrates use of the `ActorKey` to reassign a request to a specific queue.

Code Sample 3-6. Using the ActorKey

```
/* Assign a request to a queue specifying an agent as
 * the one performing the assignment
 */
aRequestHelperBean.assignRequestToQueue(anAgentBean.getKey(),
      aRequestBean.getKey(),
      aQueueBean.getKey(), aReasonCodeBean,"initial assignment of this
      request to a queue");
```

Notice that this method has a few more parameters than previous examples. These are `ReasonCode` and a free form comment used to explain the reason for the action. You can get a list of `AssignmentReasonCodeBeans` from the `AssignmentReasonCodeHelperBean`.

Example 6: Using the AttachmentHelperBean

The `AttachmentHelperBean` deserves further explanation. This bean represents a file attachment on a `MessageBean` and can be used to get the actual file down from the persistent store, storing it on the local hard drive. Code Sample 3-7,"Using the AttachmentHelperBean" below demonstrates the use of the `AttachmentHelperBean`. In this example, a message bean key is used to find all the attachments for the message specified. It then downloads all the files associated with the message to the local disk and places them in the `c:\temp` directory. The bean uses the original name of the file which is stored in the description attribute of the `MessageBean` object.

The `downloadAttachmentFile` method will throw an Exception if the file is already there.

Code Sample 3-7. Using the AttachmentHelperBean

```
Vector aVector = null;
Enumeration anEnum = null;
/*
 * Create an instance of the helper and declare
 * a place to hold our returned value bean.
 */
AttachmentHelperBean anAttachmentHelperBean =
new AttachmentHelperBean();
AttachmentBean anAttachmentBean = null;
/* This is where we actually get the bean.  You must locate
* it based on the message that it is attached to.
*/
aVector = anAttachmentHelperBean.getAttachments(aMessage.getKey());
anEnum = aVector.elements();
while (anEnum.hasMoreElements()) {
anAttachmentBean = (AttachmentBean) anEnum.nextElement();
   /*
   * This will write the file out to the directory
   * c:\temp using the original name for the file.
   */
try {
   anAttachmentHelperBean.downloadAttachmentFile(
   anAttachmentBean.getKey(), "c:\temp\"+ anAttachmentBean.getDescription());
} catch (HelperException e) {System.out.println(e.toString());
}
}
```

Using the Events API

Before attempting to integrate with other applications, you must have a thorough understanding of how Brightware processes mail. See the *Contact Center Console Guide* for an overview of message flow through the Brightware system.

The Events API is intended to be used to allow one or more third-party applications to be notified of the actions occurring within a running Brightware system. During normal operation, Queue Manager, the Contact Center Console, the Agent desktop, and the Intelligence Engine all communicate with the Brightware Server. Actions that agents, supervisors, Queue Manager, and the Intelligence Engine perform cause events to "fire" from the Brightware Server to the third-party system. These events are asynchronous to the operation of the Brightware Server, and are also queued to allow the third-party system to handle them at its own pace. The events, small Java objects, are transmitted or signaled to the third-party's `EventListeners` via Remote Method Invocation (RMI). You write `EventListeners` that will listen for and respond to the events when they are signaled. For example, an event listener could invoke the Database Interface to access or modify Brightware data objects.

Basic Integration Tasks

While your integration will depend upon your needs, there are certain tasks that all integrations must perform to get up and running. These are:

- Import the IDK classes.
- Declare an `EventListener`.
- Initialize the Event Registry and your `EventListener`.
- Unregister the `EventListener`.

Additional information and an example to aid your integration has been provided in the appendices:

- Appendix A, "Understanding Event Sequences"
- Appendix B, "Example Event Handler"

Task 1: Import the IDK Classes

To use the Events API you must import the following packages in your code:

Code Sample 3-8. Import Statements for `EventListener`

```
import com.firepond.event.*;
import com.firepond.event.agent.*;
import com.firepond.event.queue.*;
import com.firepond.event.group.*;
import com.firepond.event.request.*;
import com.firepond.event.sponsor.*;
```

Task 2: Declare an EventListener

Declaring an event listener for the specific event of interest is as easy as declaring a new class that implements the `com.firepond.event.EventListener` interface. The code sample below declares a new `ServerStartedEventListener`:

Code Sample 3-9. Declaring an `EventListener`

```
public class ServerStartedEventListener implements EventListener, Serializable
{

   /** Listen for ServerStartedEvent. If received, it will call log it to
   * the console
   * @param event The event that is being signalled
   * @since 1.0
   */
public void signal( Event event ) {
   try {if ( ! (event instanceof ServerStartedEvent)) {
System.out.println( "ServerStartedEventListener: Wrong event,"
+ event.toString() );
return;
}
System.out.println( "ServerStartedEventListener: received" );
}     catch (Exception ex) {System.out.println(
      "ServerStartedEventListener: " + ex );
}
}
}
```

Task 3: Initialize the EventListener Registry and Event Listeners

After declaring event listeners for each event of interest, the integration component should initialize the `EventRegistry` and register the event listeners. Initializing the `EventRegistry` also connects the integration component to the Brightware Server.

Be sure to specify the DNS name of the computer running the queue manager and the port on which that server is listening.

The events registry is initialized as illustrated below:

Code Sample 3-10. Initializing the Event Registry

```
// Initialize the Event registry.  Connects to the server.
EventRegistry registry = null;
String serverName = "myServer";
int port = 7001;
try {System.out.println( "start: Connecting to " + serverName );
DependentParameters.initProvidersForStandaloneApplication(serverName,PORT);
registry = new EventRegistry( "example", serverName, port, false );
```

```
System.out.println( "start: Connected to " + serverName );
} catch (Exception e) {
System.out.println( "start: Can not locate the server, " + serverName
+ ": " + e.toString() );
throw e;
}
```

Registering event listeners is also a simple process. Code Sample 3-11,"Registering EventListener" below registers an example event listener. In the sample code below `registry.register(CreatedAgentEvent.class, new CreatedAgentEventListener())` registers an instance of the class `CreatedAgentEventListener` to listen only for the `CreatedAgentEvent`. The `CreatedAgentEventListener`'s `signal(Event event)` method will only be invoked if the `CreatedAgentEvent` occurs.

The first argument in the above call can be any IDK Event located in the following packages: `com.firepond.event.agent, com.firepond.event.queue, com.firepond.event.group, com.firepond.event.request, com.firepond.event.sponsor`

Code Sample 3-11. Registering `EventListener`

```
// Register event handlers
try {registry.register(ServerStartedEvent.class,new
ServerStartedEventListener());
registry.register(CreatedAgentEvent.class, new CreatedAgentEventListener());
} catch (Exception e) {System.out.println( "start: Failed to register
listener: " + e.toString() );
throw e;
}
```

Once registered, a listener will be invoked when the event occurs by calling the `signal(Event event)` method with the event that occurred as shown in Code Sample 3-9, on page 16.

Task 4: Unregister your EventListener

When the Brightware Server is shut down, the event listeners should be unregistered and the event registry shut down as well. Code Sample 3-12,"Unregister Event Listener and Quit the Event Registry" below uses the `unregister` method to unregister the listener and the terminate method to shut down the registry.

Code Sample 3-12. Unregister Event Listener and Quit the Event Registry

```
public void stop()throws Exception
{
// Unregister event handlers
try {registry.unregister( ServerStartedEvent.class );
registry.unregister( CreatedAgentEvent.class );
} catch (Exception e) {
System.out.println( "stop: Failed to unregister from event" );
throw e;
}
// Terminate the registry - disconnects it from the server
registry.terminate();
}
```

Debugging

Edit the following section in the \cluster\lib\common\log-config.xml file installed on the Brightware Server and set priority value to "info". This will cause the Brightware Server to log each event to the console.

```
<category name="fp.integration">
     <priority value="debug" />
</category>
```

You can generate logging on the client by passing true as the last parameter to the `EventRegistry` constructor. Initialized this way, `EventRegistry` will generate an `event.log` file on disk where information about the event registry is displayed. This log file will be created in the current directory. The log will show initialization of the registry, each event that the client receives, and termination of the registry.

Chapter 4

Workflow (Agent Desktop) Integration

Introduction

Brightware provides the tools and information necessary to assist an agent in quickly and accurately processing a message for a customer. The Desktop Agent interface provides access to relevant information that is managed and maintained “inside” the system (for example: suggested responses and action history). There may be times, however, when an agent is processing a message and requires access to information maintained “outside” the Brightware product.

Access to outside information can be used to:

1. Extract information from external information stores or systems and use it to embellish the current response.
2. Take information supplied by the customer and update other information stores or systems.

The use of variables, custom actions, and the IDK in general can often be used to provide access to outside information when possible without Agent interaction. It is worth recognizing that companies often have considerable investments in existing legacy applications that could be harnessed to help an Agent compose an appropriate response or process information. It is the goal of the Brightware product to make integration with existing third-party applications as smooth and seamless as possible. Part of fulfilling this goal involves allowing the Agent Desktop application to be integrated with other applications.

The Approach

The Agent Desktop has been developed as a thin-client browser-based application. This aspect is one reason it is possible to integrate the Agent Desktop with other applications. For example, since the content rendered within the browser is all HTML, it is possible to use HTML framesets consisting of multiple frames, some of which derive their content from the standard Brightware product, while other frames in the same screen display relevant content rendered by other complementary applications. Furthermore, JavaScript and XML, downloaded within the HTML, can also be used in countless ways.

Examples of how JavaScript, downloaded within a page, can be employed to create an enhanced Agent Desktop environment:

- Popup a new window linked to a third-party browser-based application.
- Pass data between different frames or windows on the client side.
- Execute methods on a downloaded applet or ActiveX object designed to interact with stand-alone desktop applications or client-server systems.

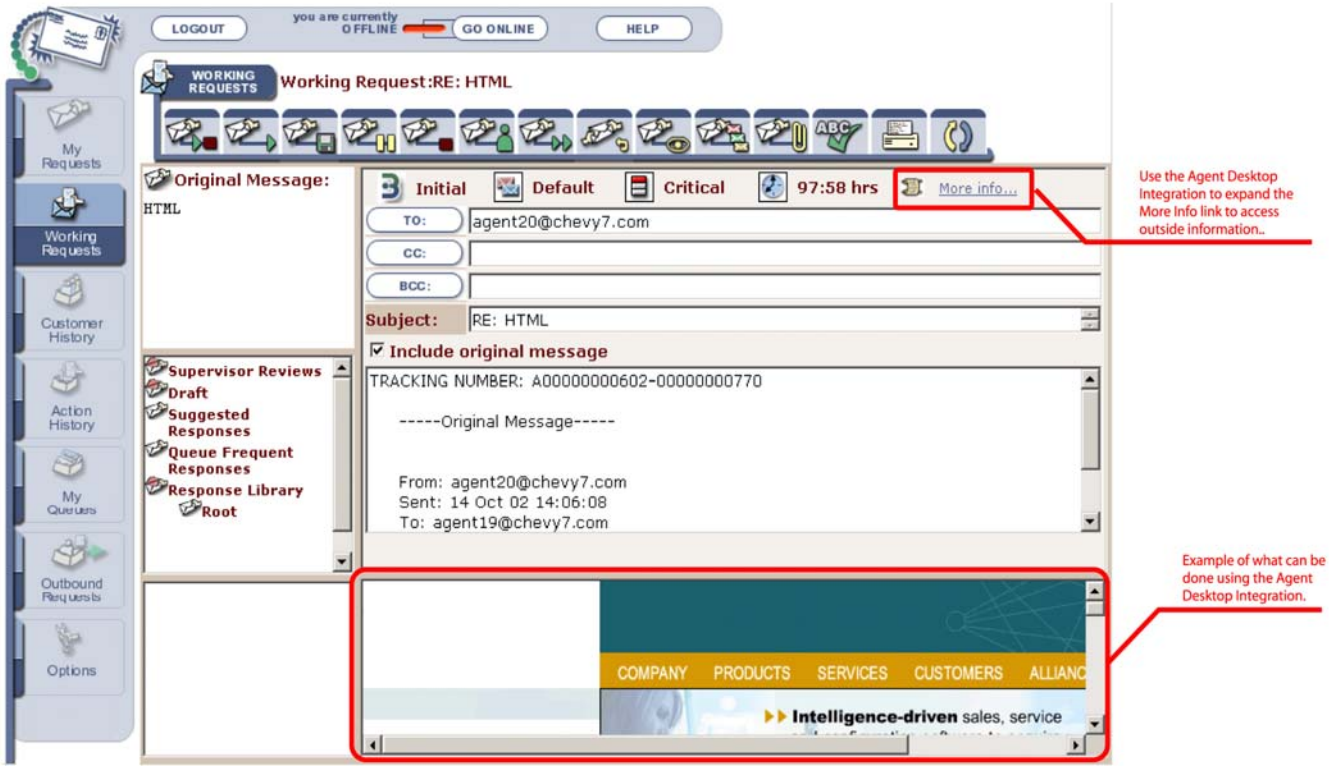

Figure 4-1. Example of Agent Desktop Integration

Considering the flexibility of HTML, JavaScript, and XML, along with the ability to download ActiveX objects and proprietary applets within the HTML, the integration possibilities on the Agent Desktop client computer are endless.

Besides the power and flexibility of HTML and JavaScript, three distinct aspects of the Brightware architecture play an important role in making a meaningful integration within the Agent Desktop possible:

1. Use and distribution of XSL templates to render content.
2. Access to the “Agent Desktop context” (for use by external applications).
3. Access to the database integration API provided by the IDK.

Use and Distribution of XSL

XSL is a J2EE standard that is used in the Agent Desktop architecture. Instead of having content rendered by servlets streamed directly to the browser, most of the servlets within the Agent Desktop generate data in the form of XML. The XML is then merged with XSL templates of the server (using XSLT) to create the final HTML content which is streamed back to the client’s browser. While the XML data is generated dynamically for each request to the server, the XSL is in the form of static text files, in a readable form, and are distributed with the application. It is therefore both possible and practical for an integrator to modify these XSL files after installing the product with the help of any text editor. Integrators could use this as a means of modifying the effective layout, content, and the “look and feel” of the Agent Desktop application.

Examples of changes made through modifying the XSL include (but are not limited to):

- Modification of framesets
- Changing references to images
- Colors
- Fonts
- Addition of both HTML and JavaScript

To locate a list of all XSL files used within the product, see the contents of the install folder, under \config\eservice\applications\DefaultWebApp_myserver.

Some of these XSL files are used to generate corresponding HTML files at the time the server is started (you can tell these files apart by looking at the file timestamps). These files are not context-sensitive to a particular agent or to a particular message being processed. However, the remainder of the XSL files are used to generate HTML at runtime, and could therefore be context-sensitive to a specific agent and the message they are currently processing. The naming convention of the XSL files makes it easy to correlate them with where they are used within the Agent Desktop.

Access to the “Agent Desktop Context”

Modifications to the XSL is in itself sufficient to allow changing the branding or linking to external applications from within the Agent Desktop. However, in most cases that action alone is unlikely to be enough to achieve a meaningful integration of the Agent Desktop. This is because third-party applications will often require information about the state of a particular Agent’s desktop in order to be able to display relevant information. In other words, the “integrated third-party application” needs to be provided the “context” of the agent’s desktop in order to allow it to function in a “context-sensitive” manner.

For example, to integrate an external application which displays a customer’s account information would require some means of establishing the customer’s identity; such as the customer’s “from address” (which is part of the “Agent Desktop Context”).

Two separate means are provided to make the “Agent Desktop Context” available:

1. Through elements published in XML.
2. Through defined variables within a dynamic JavaScript file.

Elements published in XML

This technique has not been made widely available–it is limited for now to use within WorkingRequest.xsl and WorkingRequestForm.xsl. As alluded to earlier, the XSL standard allows one to extract the values supplied in XML elements and to use them to control the generated HTML output. Different XML elements are generated dynamically by different servlets within the product, depending on the context in which they are invoked. Whenever one of the two XSL files (referred to above) is invoked, the following XML elements are also defined, and are therefore available for referencing within the XSL:

- MESSAGE_ID
- FROM

- REQUEST_ID
- QUEUE_ID
- SPONSOR_ID
- ORIGINAL_MSG_ID
- AGENT_ID

These elements are child elements of the HEADER element. For example, part of the XML generated at runtime might look something like this:

```
<HEADER>
   <MESSAGE_ID> 802 </MESSAGE_ID>
   <FROM> customer@myclient.com </FROM>
   <REQUEST_ID> 107 </REQUEST_ID>
   ...
   ...
</HEADER>
```

Refer to the WorkingRequest_X.xsl file as an example of how XSL tags can be used to extract the contextual information from the XML elements described above. This file is not actually used by the system, but could be used to replace WorkingRequest.xsl to provide a demonstration of what is possible. In the example, an additional frame is created within the “working request view”. The frame is designed to receive its content from an external URL (in this case www.edocs.com), and to have specific information from the Agent Desktop Context passed to it as parameters. See Figure 4-1, “Example of Agent Desktop Integration”, on page 20.

Variable Defined within Dynamic JavaScript

This technique is more widely available–it involves accessing and using the attributes of the Agent Desktop Context and is actually easier to use. A special servlet has been created to provide access to attributes of the Agent Desktop Context. This is done through JavaScript variables defined within a JS file. Since the context is different for each agent, this is not a static JavaScript file, but is instead generated dynamically, and therefore dependent upon the HTTP session. On any particular server (<myserver>) this file can be accessed at:

```
http://<myserver>:portnumber/WadContext.js
```

or for example:

```
http://localhost:7001/WadContext.js
```

An example of what you might expect to be returned by a request to WadContext.js:

```
WAD_MESSAGE_ID = 771;
WAD_REQUEST_ID = 602;
WAD_FROM = 'agent20@mycompany.com';
WAD_ORIGINAL_MSG_ID = 770;
WAD_QUEUE_ID = 102;
WAD_SPONSOR_ID = 2;
WAD_AGENT_ID = 120;
```

The response effectively defines several JavaScript variables (WAD_MESSAGE_ID, WAD_REQUEST_ID, etc). By including a link to this JavaScript in any HTML or XSL file, you can reference and use the defined values in a variety of ways. For example, here's the HTML code to create a link to WadContext.js:

```
<SCRIPT SRC="WadContext.js" language="JavaScript"></SCRIPT>
```

Here is how you might use the defined variables:

```
<SCRIPT>
   function useWadContext()
   {
      sURL = "http://www.yahoo.com?from=" + WAD_FROM +
      "&amp;agent=" + WAD_AGENT_ID;
      window.open(sURL, 'my_new_window');
   }
</SCRIPT>
```

The function could then be called in a variety of ways, for example:

```
<a href="#" onclick="javascript:useWadContext();">
   More info...
</a>
```

The WadContext.js file could be referenced in HTML files served up by any application on any application server–not just the set of XSL files within the Brightware product. The right set of values will be retrieved as long as the HTTP request is made from the same session as is used by the Agent Desktop. The variables will all be initialized to zero if the HTTP request is not made from the same session as used by Agent Desktop.

The WadContext.js file can also be used to acquire the values of all the *variables* (Extraction/Match/Phrase/External) defined within the context of a request. This can be done by passing it a parameter named "*variables*". For example:

```
<SCRIPT SRC="WadContext.js?variables" language="JavaScript"></SCRIPT>
```

The resulting generated JavaScript will then include JavaScript variables for all the defined system variables. This feature is highly significant because the values of variables extracted from an incoming message are often the most relevant information that needs to be passed on to external systems. Variable values can be accessed and passed on in this manner without the need to write additional supporting IDK code.

Access to Database Integration API Provided by the IDK

The IDK's database integration kit (DBI) provides the ability to retrieve data from different kinds of Brightware databases. The Agent Desktop Context provides access to the values of certain fields (such as the Request ID) which can then be used to retrieve any amount of more detailed information, if necessary, through the DBI. Without access to the Agent Desktop Context, it is not possible to use the IDK alone to suitably parameterize other applications so that they provide context-sensitive content. At the same time, the set of keys provided in the context will not provide

access to the depth of information that is sometimes required–it merely provides access to the “tip of the iceberg”. Therefore, the Agent Desktop Integration complements the functionality available through the IDK.

Examples of Integration Scenarios and Possibilities

- A new frame can be created to link to a browser-based third-party application, and pass relevant information from the desktop context to parameterize it appropriately.
- A hyperlink can be added into one of the existing pages that, when clicked, pops up a new window that is linked to an external application (that once again is suitably parameterized with the agent desktop context).
- JavaScript can be added into an existing page (WorkingRequestForm.xsl, for example) that will cause a new window to pop up automatically as soon as the page is loaded.
- Instead of linking directly to an external application, one could link to a servlet deployed with the rest of the Brightware application. This custom-built servlet could use IDK to extract more data, perform some processing, and then perform one of two things:
    - Serve up the response itself, or
    - Redirect to a different URL (possibly external) with an updated set of parameters

In all the examples, it is largely assumed that the application being integrated with is deployed on a web server capable of accepting HTTP requests and serving up browser-based content. That however, does not always need to be the case. It is possible to integrate with non web-based applications as well–it is just more cumbersome. To launch a “thick client” application on the agent’s desktop, an applet or ActiveX object can be embedded within the downloaded HTML. The ActiveX object could be custom built using a tool like Microsoft Visual Basic, and designed to launch and then interact with any kind of application on the agent’s computer. For example, each time a new message is processed by the agent, JavaScript can be used to invoke methods on the downloaded ActiveX object that will in turn update the context of the complimentary stand-alone third-party application.

Even though modification of branding functionality through changes to the XSL files are made technically possible, arbitrary modifications to the XSL by third-party implementers and system integrators could *possibly* detract from the functionality of the product, or even introduce defects! Further, such modifications (if made) cannot be supported by Brightware Technical Support, or remain intact when the product is upgraded from one version to a newer one. However, it is believed that to achieve a meaningful integration, minimal changes to a *very small subset* of the available XSL files are generally required, and system integrators will need to reintroduce the changes to a newly installed upgraded version of the product.

Chapter 5

HTML Formatting of Outbound Messages

This chapter describes the concept of HTML Formatting of Outbound Messages and how to configure a template to allow this feature to be used within the Agent Desktop application.

Introduction

Branding plays an important part in building every company's image. The email messages that contain responses to customers also play an important role in creating and building perceptions of the company that sends them. The more professional looking the email, the more positive its effect on the image of the company. Allowing outgoing email messages to be formatted as HTML opens up a world of possibilities for creating professional looking messages. HTML formatting allows the use of background images, company logos, and fine control over colors and fonts. It also offers the use of forms or the inclusion of hyperlinks to a company's corporate website (for example).

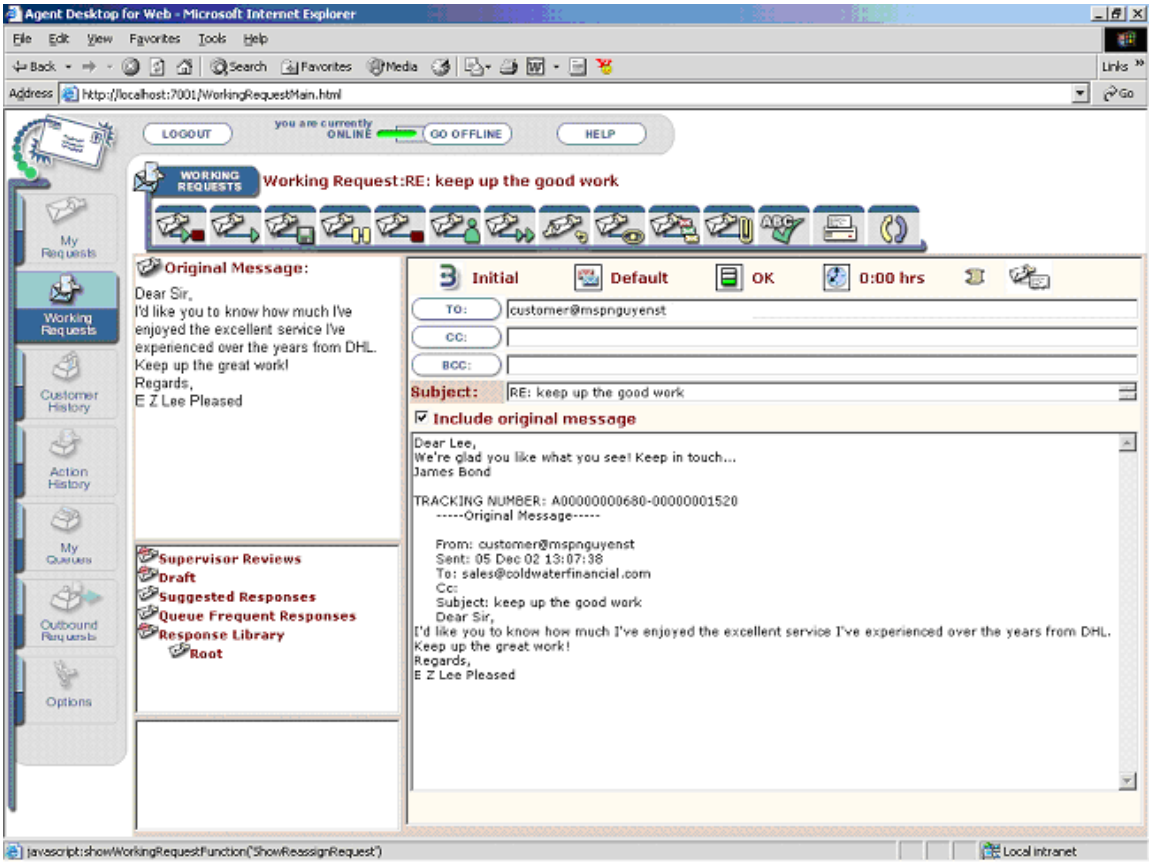

Figure 5-1. Standard plain-text message.

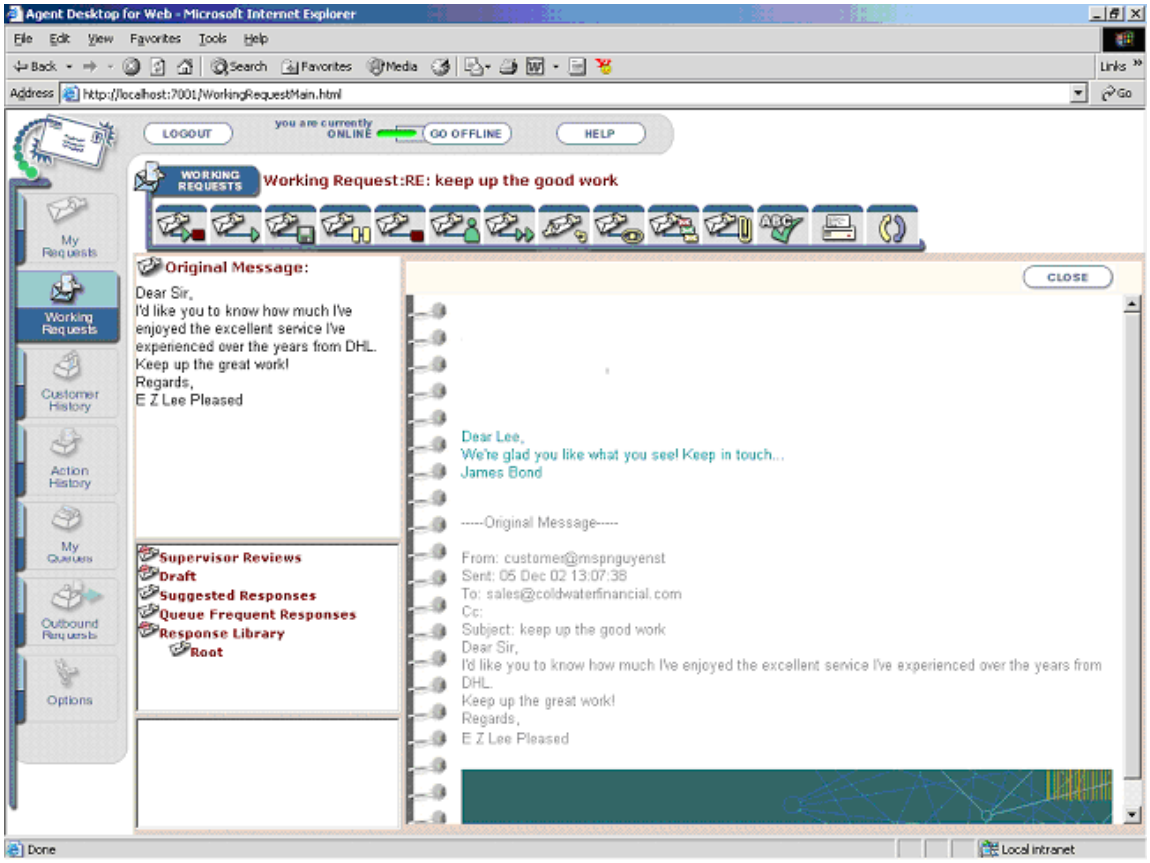

Figure 5-2. HTML enhanced message.

Features

Brightware support for HTML formatting of outbound messages features the following:

- Ability to insert images (to support the use of company logos) and alter fonts and colors.
- Ability to use background images to create the effect of using “stationery”.
- The addition of HTML support does not change the way an Agent or Supervisor interacts with the system (the User Interface design and underlying infrastructure for creating responses, such as reviewing, forwarding, assigning, transferring, and rejecting messages all remain unaffected).
- Ability to use templates to allow automatic compliance with approved company branding and formatting standards of every message leaving the system.
- Allows messages to be uniquely formatted for the different Business Units and queues (if desired) by using different templates for each.
- Allows Agents to preview the formatted version of a message.
- The different parts of the message (original message, tracking text, and the reply) can receive different formatting options.

- Supports plain-text formatting in addition to HTML. Support for plain-text is important because it is the lowest common denominator supported by even the most primitive of mail clients.
- Identifies the mail client signature used to generate an incoming message and provides the ability to format messages based on the mail client used. Allows “optimistic” and “pessimistic” options that dictate if an outgoing message is formatted as HTML or plain-text.
- Allows expansion of recognized mail clients list through the modification of a configuration file (without the need for code changes).
- Allows the use of a formatting template even for messages sent in plain-text. It can, however, be used to enhance the text with headers and footers to provide a suitable alternative to HTML elements (such as images).
- When using HTML formatting, an option is available to send out HTML and plain-text as separate parts of a multipart message – allowing messages to be viewed in both formats.
- Supports the option of using plain-text or HTML at a Business Unit level rather than just as one system-wide setting.

Template Configuration

The approach used in formatting a message as HTML involves the provision and use of a template. This alleviates the need for an Agent to spend time and energy applying formatting to individual messages, and ensures the consistent branding of all outgoing messages across all Agents. The template is used to format a message when an Agent invokes the preview function within the Agent Desktop (working request view) as well as when the Outbound Mail Handler sends out a message.

A single physical file named **OutboundMsg.xsl** represents the template for all outbound messages. This file is located within the install folder:

```
\config\eservice\applications\DefaultWebApp_myserver
```

This one file can contain several different templates that could be applied as needed for use with different Business Units and queues. The file can also contain a template that would be used exclusively to format messages intended for plain-text.

At runtime, the components of the application merge the response in the form of an XML document with the template (an XSL file) to generate the message that is sent to the client. The XML document contains several different kinds of XML elements that represent various parts of the message as well as information about it – such as which Business Unit and queue it came from. The information in these XML elements are used in the XSL template (XSL essentially is a template for generating HTML).

XSL is a well documented non-proprietary standard and it is beyond of the scope of this document to cover. An elementary understanding of the XSL syntax is required to configure or customize a template. An alternative is to use one of many third-party off-the-shelf tools to edit the XSL file. The Brightware product already includes a basic usable out-of-the-box template, as well as a sample of a template file (**OutboundMsg_X.xsl**) that demonstrates many of the formatting possibilities. For example, how to achieve a distinct look and feel for different Business Units or hide the tracking text.

The following is a list of XML elements generated within the XML document that represent an outgoing message: <root>, <reply>, <trackingText>, <originalMessage>, <sponsor>, <queue>, and <plainText>. The <root> element is the parent of all the other elements.

NOTE: The <sponsor> element represents the business unit.

For example, an XML document for a message might look similar to this:

```
<root>
   <reply>
      Dear Sir,
      Thanks for contacting us. We will be responding within the
      next few days with the information you requested.
   </reply>

   <trackingText>
      TRACKING NUMBER: A00000000604-00000001268
   </trackingText>

   <originalMessage>
      -----Original Message-----
         From: customer@edocs.com
         Sent: 25 Nov 02 16:55:28
         To: sales@coldwaterfinancial.com
         Cc:
         Subject: Enquiry
         Hi,
         Could you please enlighten me? (Soon!)
   </originalMessage>

   <sponsor> Initial </sponsor>
   <queue> Sales & Service </queue>
   <plainText> true </plainText>
</root>
```

Support for Mail Clients Incapable of Rendering HTML

There are many reasons why a company would want to send the majority of messages formatted as HTML. However, as mentioned earlier, it is important to have some support for primitive mail clients that cannot recognize HTML content.

Brightware provides several levels of support for primitive mail clients and several configuration options that will determine how a message is formatted:

- The use of HTML in formatting messages can be disabled for any specific Business Unit or all Business Units. This can be specified under the Business Unit tab from the Contact Center Console. HTML formatting is enabled by default. If it is disabled, messages will be sent as plain-text, regardless of the kind of mail client used by the mail recipient.
- A system wide setting is provided to allow a Supervisor or System Administrator to specify that whenever a message is being formatted as HTML, a textual representation of the same message should be included – just in case the mail client cannot recognize HTML. This is done by sending out a multi-part MIME message, one part contains HTML and the other contains plain-text. Different mail clients have different ways of displaying multi-part messages. Outlook Express may display the different parts one below the other, while Outlook would display the plain-text part as an attachment on the primary (HTML) message. This setting is provided through the configuration parameter called **always.include.plaintext.version** and is turned off by default.
- The decision of whether or not to use HTML formatting can be based on the mail client used to send out the original incoming message. The system detects the signature of the mail client used to send an incoming email message. This corresponds to the X-Mailer header of the email. Each mail client has a unique signature. Based upon a knowledge of which common mail clients do or do not support HTML, and their signatures, the best decision can be made about whether or not to use HTML in outbound messages. Three configuration parameters are provided with relation to this feature:
    - The parameter **plain.text.email.clients** should be used to configure a semicolon delimited list of signatures of all mail clients that cannot read HTML. For example, the signature for AOL Version 4.0 will be a member of this list. Each element of the list need only be a sub-string of the entire X-Mailer header. This list is incomplete since there are a large number of obscure or little-used mail clients in use. The more complete the list is, the more effective it will be. The completeness of this list is only relevant when the **optimistic** option is in use (see the third bullet below).
    - The parameter **html.enabled.email.clients** should be used to configure a semicolon delimited list of signatures of all mail clients that are known to read HTML. This list is incomplete. The more complete it is, the more effective it will be. The completeness of this list is only relevant when the **pessimistic** option is in use (see the third bullet below).

- The parameter named **use.html.formatting.optimistically** is what eventually dictates the algorithm used to decide what format should be used. It also dictates which of the two lists becomes relevant.

  Optimistic Approach: If this parameter is set to true, HTML formatting is used unless the mail client of the recipient definitively belongs to the category of known clients that cannot read HTML.

  Pessimistic Approach: If this parameter is set to false, HTML formatting is used only if the mail client of the recipient is one of the clients definitively known to be able to read HTML. This is the safer approach, and what is used if the reliability of email responses is especially important. Note that most clients that cannot render HTML can still read HTML and render it as plain-text anyway – making HTML the more viable choice.

Since the lists of mail client signatures are configurable, they can be progressively updated by System Administrators without changes to the core product. Its important to note, that even if the lists are trivial, they can still be effective. For example, if the only element of the HTML enabled client list consisted only of the word Outlook, and one also chose the pessimistic option, it would still make it possible to reach over 60% of clients with HTML, while the remainder would receive plain-text (which is better than 100% of clients receiving plain-text).

In addition to the above, support for use of templates in formatting and message composition is provided even if a message is being sent as plain-text. This allows an outgoing message to be enhanced with additional content (such as headers and footers) just as is the case for HTML messages. The XML element <plainText> passed to the template holds the value of true or false, and is provided for just this purpose.

Appendix

A

Understanding Event Sequences

Introduction

This appendix is provided to give you a background of what activities occur within the Brightware application during certain actions. Use these scenario explanations to plan your integration, or use the JavaDocs to determine how to best integrate your applications.

Server Startup Processes

The Brightware or third-party systems may be started in any order. The integration must handle this requirement. How this requirement is satisfied is left to the integrator.

When the Brightware system starts, it initializes many components and draws upon the information in the variable-pool.xml file. In terms of integration, the important components are RMI, JMS, the Event Registry, and the Event Topics. During startup, the Brightware Server will attempt to start an external process specified in the configuration file. After the Brightware Server has completed its initialization it will fire the `ServerStarted` event. Your integration code can be started by the `exec()`call. It should register a listener for each event that you find important. The integration code can also initialize the DBI at this time (see Figure A-1, on page 32).

Please note that the variable-pool.xml file contains information the system requires to run. This file is located in the install directory and should not be moved.

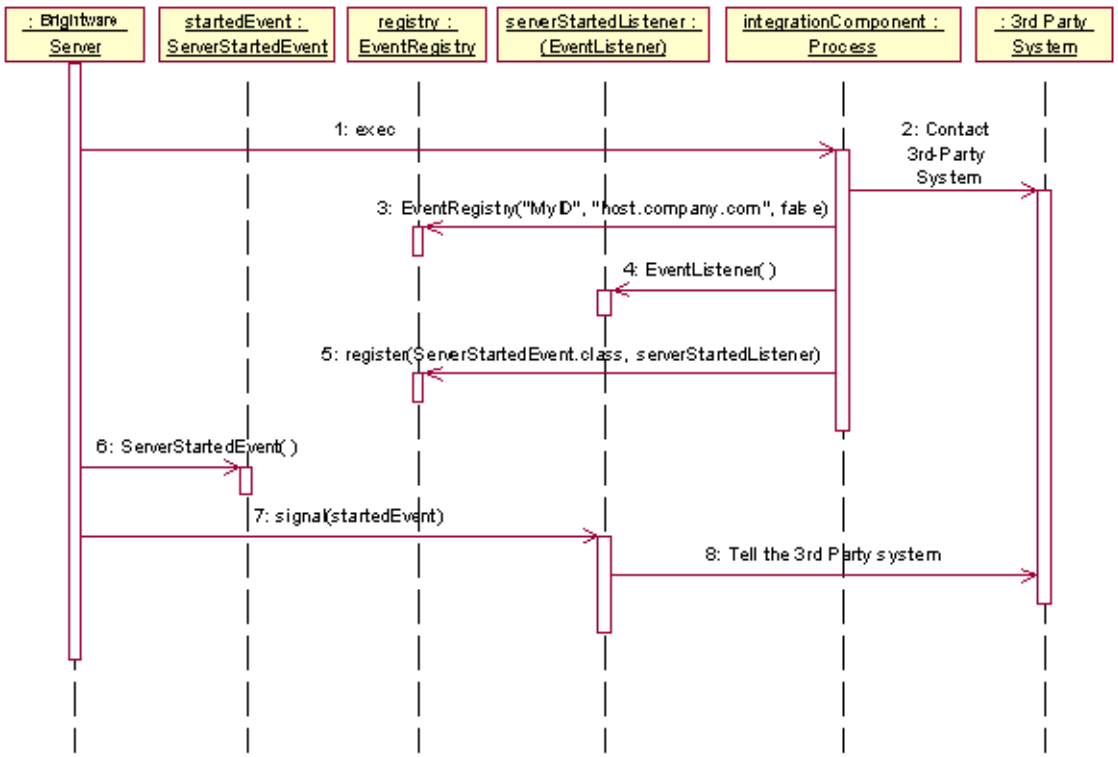

Figure A-1. Server Startup

The external process started by the `exec` call is configurable in the `config.cml` file as is the delay from starting the external process to the signaling of the `ServerStarted` event. Code Sample A-1, “config.cml”, on page 32 is a snippet from `config.cml`, an XML based configuration file for the server. This file is found in the root directory of the Brightware Server installation. The external process is configured using the `eventservice` parameter. It must follow the standards for an executable process. The value of the trace parameter will be appended to the `eventservice` parameter as an argument. You can configure a delay in milliseconds using the `startupdelay` parameter.

Code Sample A-1. config.cml

```
<application name="Firepond.event.application">
      <dependency application="Firepond.bus.application"/>
      <instance instance-of="Firepond.event.registry" name="Firepond.event.handler">
          <parameter-group>
              <parameter-group name="event">
                  <parameter name="shutdowndelay" value="5000"/>
                  <parameter name="eventservice" value="starteventcompatibility.cmd"/>
                  <parameter name="eventorder" value="true"/>
                  <parameter name="trace" value="false"/>
                  <parameter name="startupdelay" value="40000"/>
              </parameter-group>
          </parameter-group>
      </instance>
  </application>
```

The remainder of this discussion assumes that the Integration Component, shown in Figure A-1, on page 32, was started using the `exec()` command from the Brightware Server. The Integration component should create an `EventRegistry` and register an `EventListener` for each event of interest. The remaining event scenario diagrams assume that the startup and registration process is complete.

Server Shutdown Processes

The Brightware or third-party systems may be shut down in any order.

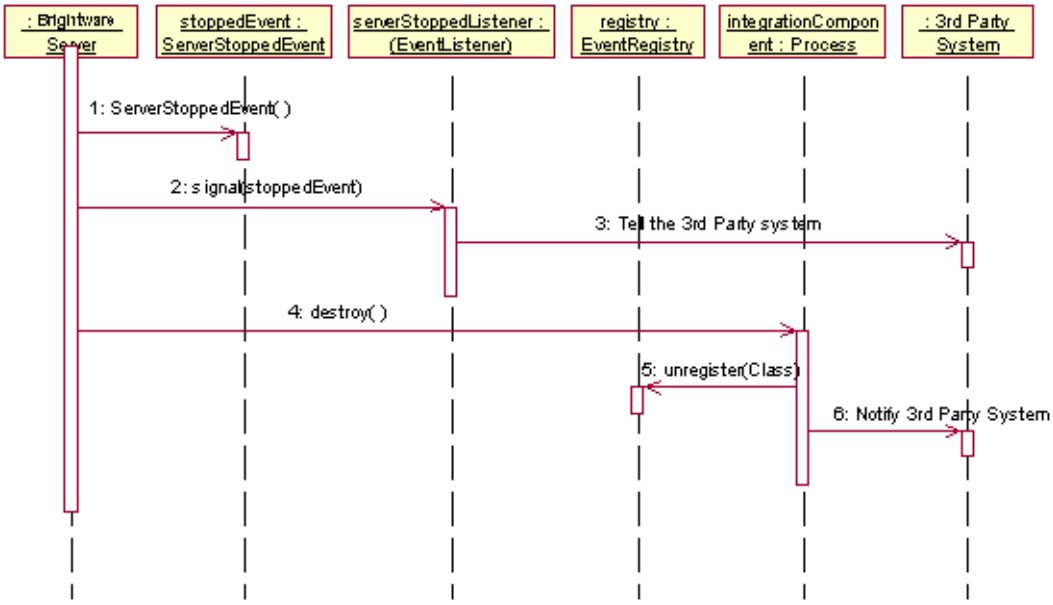

Figure A-2. Brightware Server Shutdown

See Figure A-2, “Brightware Server Shutdown” above. When the Brightware Server stops, it signals the `ServerStopped` event, waits a short time, and then terminates the Integration Component started during the Brightware Server startup procedure as in Figure A-1, “Server Startup”. You can configure the delay between the `ServerStopped` event and the termination of the Integration Component in the `config.cml` file. The `shutdowndelay` parameter in Code Sample A-1,“config.cml” expresses this delay in milliseconds.

Brightware Component Processes and Event Firing

The following section describes the Events API.

Intelligence Engine

1. Start the Intelligence Engine.

   ApplicationStartedEvent("firepond.answeragent.application")

   ServerStartedEvent

2. Retrieves a new email from the mail server, determines which Business Unit the request belongs to and stores the request in the Inbound Queue of that Business Unit.

   EnqueueEvent(Inbound)

3. Analyze the request email content and determine the intent. Upon completing NLP processing the request is stored in the database.

   CreatedRequestEvent

   AssignMessage(AA)

   DeliveredMessageEvent(AA)

   UnAssignMessage(AA)

   DequeueEvent(INBOUND)

   DequeueEvent(OUTBOUND) in case when you have Enqueue(outbound)

4. Based on the intent and action rules defined in the knowledge base, it proceeds to one of the following:

   (a) If the request is Threaded (this event will fire even when threading is disabled).

   ThreadedRequestReceivedEvent

   (b) If the intent is certain and the Auto Response feature is on, then generate the auto reply and close the request.

   AutoReplySentEvent

   ClosedRequestEvent

   EnqueueEvent(OUTBOUND)

   DequeueEvent

   (c) Otherwise bundle the suggested replies, if any, to the request and assign the request to a queue.

   RequestAssignedbyAnswerEvent

   EnqueueEvent(SUBJECT)

   (d) If an incoming request triggers an auto close rule then:

   CloseRequestEvent

   EnqueueEvent(INBOUND)

   DequeueEvent(OUTBOUND)

5. If Auto Acknowledge feature is on then generate auto acknowledge reply for the request.

   AutoAcknowledgeReplySentEvent

   EnqueueEvent(OUTBOUND)

DequeueEvent

6. Cycle repeats starting at step 2.
7. Brightware Intelligence Engine stops.

   ApplicationStoppedEvent("firepond.answeragent.application")

   ServerStoppedEvent

Queue Manager

1. Queue Manager server is started.

   ApplicationStartedEvent("firepond.email.router.application")

   ServerStartedEvent

2. The Contact Center database is polled at a predefined interval.

   (a) If any outbound request replies are seen, then send them out via the connected email server.

   ReplySentEvent

   DequeueEvent(OUTBOUND)

   (b) If an available agent and a request that has been directly assigned to that agent are seen, then deliver the request to the agent.

   DeliveredRequestEvent

   DeliveredMessageEvent

   (c) If an available agent is found, then deliver a request that is waiting in a queue to the agent based on the routing criteria.

   DeliveredRequestEvent

   AssignedMessageEvent

   DeliveredMessageEvent

3. QueueManager detects change in queue status.

   QueueOKEvent

   QueueWarningEvent

   QueueCriticalEvent

4. Queue Manager server is shutdown.

   ApplicationStoppedEvent("firepond.email.router.application")

   ServerStoppedEvent

Agent Desktop

1. Agent Desktop launched by the agent. Agent state changed to *online.*

   AgentOnlineEvent

   MakeAgentOnlineEvent

2. Agent performs one of the actions/tasks as follows:

   (a) Make available – The Agent clicks the *Go Online* button to make himself available.

AgentReadyEvent

MakeAgentBeReadyEvent

If there are open requests waiting for the agent, one will be delivered to the agent.

AgentWorkingEvent

MakeAgentWorkingEvent

(b) Make unavailable – The Agent clicks the *Go Offline* button to make himself unavailable.

AgentUnavailableEvent

MakeAgentUnavailableEvent

(c) Send reply – The Agent clicks *Send* to send a response to the customer.

ReplySentEvent

EnqueueEvent(OUTBOUND)

Dequeue(OUTBOUND)

If the agent is under review (the flag *Send all replies composed by this agent to review* is checked in the Contact Center Console).

AssignedRequestToReviewEvent

(d) Close request – The Agent clicks the *Close* button to close the request.

ClosedRequestEvent

UnassignedMessageEvent

DequeueEvent

AgentReadyEvent

MakeAgentBeReadyEvent

If there are new requests in the agent's queue:

AgentWorkingEvent

MakeAgentWorkingEvent

(e) Send&Close – The Agent clicks the *Send&Close* button to send the response and close the request at the same time.

Fires all applicable events in (d) and (e).

(f) Pend request – The Agent clicks the *Pend* button to pend the currently working request.

PendedRequestEvent

AgentReadyEvent

MakeAgentBeReadyEvent

If the agent has requests waiting in his queue:

AgentWorkingEvent

MakeAgentWorkingEvent

(g) Unpend request – The Agent clicks the *Unpend* button to unpend the request:

UnpendedRequestEvent

AgentWorkingEvent

MakeAgentWorkingEvent

(h) Reassign request – The Agent clicks the *Reassign* button to reassign the request to another agent or queue.

ReassignRequestEvent

UnassignedMessageEvent

DequeueEvent

EnqueueEvent

AssignedMessageEvent (if agent specified)

AgentReadyEvent

MakeAgentBeReadyEvent

If the agent has requests waiting in his queue:

AgentWorkingEvent

MakeAgentWorkingEvent

(i) Forward the request – The Agent clicks the *Forward* button.

ForwardedRequestEvent

DequeueEvent

EnqueueEvent

(j) Pass the current request to the supervisor for review – The Agent clicks the *Review* button to send a response to the supervisor to review.

AssignedRequestToReviewEvent

UnassignedMessageEvent

DequeueEvent

EnqueueEvent

AgentReadyEvent

MakeAgentBeReadyEvent

If the agent has requests waiting in his queue:

AgentWorkingEvent

MakeAgentWorkingEvent

(k) Transfer request. The agent clicks the Transfer button.

TransferRequestEvent

Unassign

Dequeue

CloseRequest

(l) Reopen request. The agent clicks the reopen button.

OpenedRequestEvent

Enqueue

(m) Agent Initiated Mail – Agent works in the Agent Initiated Mail View and clicks the *Send&Close* button to send out the message.

CreatedRequestEvent

AgentInitiatedRequestEvent

EnqueueEvent(outbound)

DequeueEvent(outbound)

(n) Agent goes offline – The Agent clicks the availability button to make himself unavailable.

AgentUnAvailableEvent

MakeAgentUnAvailableEvent

AgentOfflineEvent
MakeAgentOfflineEvent

Contact Center Console Events

A supervisor logs on to Contact Center Console to perform the following tasks:

1. Close request – The request is closed using the *Close* button.
   ClosedRequestEvent
   UnassignedMessageEvent
   DequeueEvent
   AgentReadyEvent
   MakeAgentBeReadyEvent
   If the agent has requests waiting in his queue:
   AgentWorkingEvent
   MakeAgentWorkingEvent
2. Reassign request – Supervisor reassigns request via the Contact Center Console.
   ReassignRequestEvent
   UnassignedMessageEvent
   DequeueEvent
   EnqueueEvent
   AssignedMessageEvent
   AgentReadyEvent
   MakeAgentBeReadyEvent
   If the agent has requests waiting in his queue:
   AgentWorkingEvent
   MakeAgentWorkingEvent
3. Approve review – Response in supervisor review is approved via the Contact Center Console.
   CloseRequestEvent()
   ReplySentEvent()
   ApprovedRequestEvent()
   AssignedRequestFromReview ()
   Dequeue(ReviewQueueID)
   Enqueue(OutboundQueueID)
   Dequeue(OutboundQueueID)
4. Close review – Response in supervisor review is closed via the Contact Center Console.
   RejectRequestEvent
   ClosedRequestEvent (If a request in the Review queue had been closed in the regular queue this event will not fire).
   RequestAssignedFromReview(RequestID, -1)-d
   Dequeue(ReviewQueue)-d

5. Reassign review – Response in supervisor review is reassigned via the Contact Center Console.

   RejectRequestEvent

   ReassignRequestEvent

   RequestAssignedFromReview()

   Dequeue(ReviewQueue)

   Enqueue(AgentQueue)

   AssignedMessage(ToAgentID)

   Create a Sponsor (Business Unit)

   If the request was just Sent for review, the following events will fire in addition to events above:

   UnAssignMessage()

   Dequeue()

6. CreatedSponsorEvent

   QueueCreatedEvent (5 times for the 5 system queues that are created for the Business Unit by default)

7. Delete a Sponsor (Business Unit)

   DeletedSponsorEvent

   DeletedQueueEvent (fires the number of times equal to the number queues the sponsor has)

   If any agents were assigned to this Sponsor the following event might fire:

   UnassignedAgentFromSponsorEvent

8. Save modified Sponsor (Business Unit) property value

   UpdatedSponsorEvent

9. Create an agent for the selected Business Unit

   CreatedAgentEvent

   AssignedAgentToSponsorEvent

10. Delete an agent

    UnassignedAgentFromQueueEvent

    UnassignedAgentFromGroupEvent

    DeletedAgentEvent

    UnassignedAgentFromSponsorEvent

11. Save modified agent property values (such Agent's Custom Signature, Email Address, First Name, Last Name, Login ID, Notes and Review Status)

    UpdatedAgentEvent

    AgentPropertiesChangedEvent

    UpdatedAgentEvent

    MakeAgentActiveEvent or MakeAgentInactiveEvent

    AgentActiveEvent or **AgentInactiveEvent**

12. Create a queue

    CreatedQueueEvent

13. Delete a queue

UnassingedAgentFromQueueEvent
UnassignedGroupFromQueueEvent
DeletedQueueEvent

14. Save modified queue property value
    UpdatedQueueEvent
15. Create a group
    CreatedGroupEvent
16. Delete a group
    UnassignedAgentFromGroupEvent
    UnassignedGroupFromQueueEvent
    DeletedGroupEvent
17. Save modified group property value
    UpdatedGroupEvent
18. Assign an agent to queues
    AssignedAgentToQueueEvent
19. Un-assign an agent from queues
    UnassignedAgentFromQueueEvent
20. Assign an agent to groups
    AssignedAgentToGroupEvent
    AssignedAgentToQueueEvent
21. Un-assign an agent from groups
    UnassignedAgentFromGroupEvent
    UnassignedAgentFromQueueEvent
22. Assign a group to queues
    AssignedGroupToQueueEvent
    AssignedAgentToQueueEvent
23. Un-assign a group from queues
    UnassignedGroupFromQueueEvent
    UnassignedAgentFromQueueEvent

New Events

- CreatedSponsorEvent(SponsorKey spk)
- DeletedSponsorEvent(SponsorKey spk)
- UpdatedSponsorEvent(SponsorKey spk)
- AgentPropertiesChangedEvent(AgentKey apk)
- AssignedAgentToSponsorEvent(AgentKey apk, SponsorKey spk)
- UnassignedAgentFromSponsorEvent(AgentKey apk, SponsorKey spk)
- AgentInitiatedRequestEvent(SponsorKey spk, RequestKey rpk, AgentKey apk, MessageKey mpk)

Appendix

B

Example Event Handler

The root directory of the Brightware IDK, here after known as `IDKROOT`, is typically C:\edocs\Brightware. This directory is the place you chose to install the IDK. Please substitute the actual directory location for IDKROOT in the example.

Running the Example

The example source code is installed by the IDK installer to `IDKROOT\lib\integation\Example\ExampleEventHandler.java`. Also, `testExampleEventHandler.cmd` is installed in the `IDKROOT` directory. Running this command file from the IDKROOT directory will execute the example code. The TestExampleEventHandler.cmd file now contains an extra parameter called -DWeblogic.RootDirectory="InstalledDirectrory". This parameter should be implemented in any CMD file used when testing the IDK. After starting the `ExampleEventHandler`, create a new agent using the Contact Center Console and you should see something like the following:

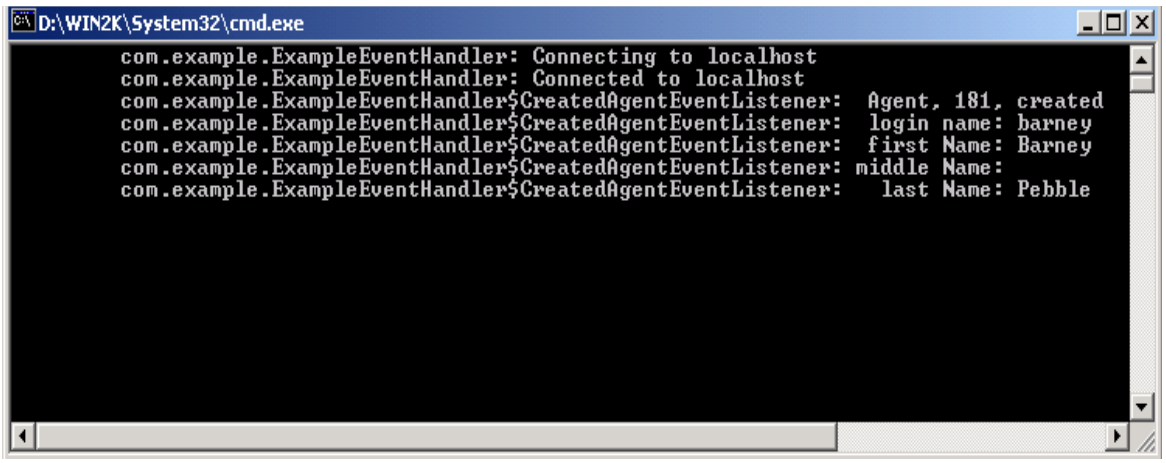

Figure B-1. Example Class Output

The Example Source Code

```
package com.firepond.example;


// The firepond event imports
import com.firepond.event.*;
import com.firepond.event.agent.*;
import com.firepond.event.queue.*;
import com.firepond.event.group.*;
import com.firepond.event.request.*;


import com.firepond.dbi.*;


import java.io.Serializable;


/**
 *
 * Demonstrates a simple event handler.<p>  The example initializes the event registry as
 * well as the DBI, then registers several event handlers with the event API.  These event
 * handlers use the DBI to access information about the objects that the events are
 * notifying about.  Once the information is found, they simply print that information
 * to the screen for the sake of the example.  You may want to do something a little
 * more useful then that.<p>
 *
 * @author  Chris Zielke, Tony Stone
 * @version 2.0
 */
public class ExampleEventHandler extends java.lang.Object {
    public static String    PORT = "7001";

 /**
 *
 * A simple event handler to handle <code>ServerStartedEvent</code> events.<p>
 *
 */
    public class ServerStartedEventListener implements EventListener, Serializable {


        /** Listen for ServerStartedEvent. If received, it will call log it to the console
         * @param event The event that is being signalled
         * @since 1.0
         */
        public void signal( Event event ) {
            try {
                if ( ! (event instanceof ServerStartedEvent)) {
                    System.out.println( this.getClass().getName() + ": Wrong type of Event, " +
event.toString() );
                    return;
                }
                System.out.println( this.getClass().getName() + ":  received" );
            } catch (Exception ex) {
                System.out.println( this.getClass().getName() + ": " + ex );
            }
        }
    }

 /**
```

```
 *
 * A simple event handler to handle <code>CreatedAgentEvent</code> events.<p>
 * This handler simply traps the event and looks up the beans that were involved
 * in the event.  It then prints that information out to the screen.<p>
 *
 */
    public class CreatedAgentEventListener implements EventListener, Serializable {
  private AgentHelperBean anAgentHelperBean = new AgentHelperBean();

        /**
         * Listen for ServerStartedEvent.<p> If received, it will call log it to the console and
         * set the agentCreated flag to true.<p>
         *
         * @param event The event that is being signalled
         * @since 1.0
         */
        public void signal( Event event ) {
        AgentBean anAgentBean = null;

            try {
                if ( ! (event instanceof CreatedAgentEvent)) {
                    System.out.println( this.getClass().getName() + ": Wrong type of Event, " +
event.toString() );
                    return;
                }
                CreatedAgentEvent cae = (CreatedAgentEvent) event;

 // Ask the DBI to get the new agent for you
 anAgentBean = anAgentHelperBean.getAgentByKey(cae.getAgentKey());

 // Do something with the Agent. In this case we just print it to the screen
              System.out.println( this.getClass().getName() + ":  Agent, " + anAgentBean.getKey().getId()
+ ", created" );
 System.out.println( this.getClass().getName() + ":  login name: " + anAgentBean.getUserName());
 System.out.println( this.getClass().getName() + ":  first Name: " + anAgentBean.getFirstName());
 System.out.println( this.getClass().getName() + ": middle Name: " + anAgentBean.getMiddleName());
 System.out.println( this.getClass().getName() + ":   last Name: " + anAgentBean.getLastName());

            } catch (Exception ex) {
                System.out.println( this.getClass().getName() + ": " + ex );
            }
        }
    }

 /**
 *
 * A simple event handler to handle <code>EnqueueEvent</code> events.<p>
 * This handler simply traps the event and looks up the beans that were involved
 * in the event.  It then prints that information out to the screen.<p>
 *
 */
public class EnqueueEventListener implements EventListener, Serializable {
  private QueueHelperBean  aQueueHelperBean= new QueueHelperBean();
     private MessageHelperBean aMessageHelperBean = new MessageHelperBean();

        /**
         * Listen for EnqueueEvent.<p>
```

```
         *
         * @param event The event that is being signalled
         * @since 1.0
         */
        public void signal( Event event ) {
     EnqueueEvent eqe = null;

            try {
                if ( ! (event instanceof EnqueueEvent)) {
                    System.out.println( this.getClass().getName() + ": Wrong type of Event, " +
event.toString() );
                    return;
                }
                eqe = (EnqueueEvent) event;

                /*
                 * In this example we will check the queue type to be sure that
                 * the type is something we want to pay attention to.
                 */
                if (eqe.getQueueType() == com.Firepond.domain.QueueType.SUBJECT) {
                QueueBean aQueueBean = aQueueHelperBean.getQueueByKey(eqe.getQueueKey());
 MessageBeanaMessageBean = aMessageHelperBean.getMessageByKey(eqe.getMessageKey());

 /*
  * Here we simply print the results.
  */
 System.out.println(this.getClass().getName() + ": A message was queued to the " + aQueueBean.getName() +
" queue");
 System.out.println(this.getClass().getName() + ":      the message subject is '" + aMessageBean.getSub-
ject() + "'");

                } else {
                System.out.println(this.getClass().getName() + ": A message was queued to a queue that we
are not interested in");
                }
            } catch (Exception ex) {
                System.out.println( this.getClass().getName() + ": " + ex );
            }
        }
    }

 /**
 *
 * A simple event handler to handle <code>DequeueEvent</code> events.<p>
 * This handler simply traps the event and looks up the beans that were involved
 * in the event.  It then prints that information out to the screen.<p>
 *
 */
public class DequeueEventListener implements EventListener, Serializable {
  private QueueHelperBean  aQueueHelperBean= new QueueHelperBean();
     private MessageHelperBean aMessageHelperBean = new MessageHelperBean();


        /**
         * Listen for DequeueEvent.<p>
         *
         * @param event The event that is being signalled
```

```
         * @since 1.0
         */
        public void signal( Event event ) {
        DequeueEvent dqe = null;

            try {
                if ( ! (event instanceof DequeueEvent)) {
                    System.out.println( this.getClass().getName() + ": Wrong type of Event, " +
event.toString() );
                    return;
                }
                dqe = (DequeueEvent) event;

                /*
                 * In this example we will check the queue type to be sure that
                 * the type is something we want to pay attention to.
                 */
                if (dqe.getQueueType() == com.Firepond.domain.QueueType.SUBJECT) {
                /*
                 * Get the beans associated with this dequeue event
                 */
                QueueBean aQueueBean = aQueueHelperBean.getQueueByKey(dqe.getQueueKey());
 MessageBeanaMessageBean = aMessageHelperBean.getMessageByKey(dqe.getMessageKey());

 /*
  * We simply print them here.
  */
 System.out.println(this.getClass().getName() + ": A message was dequeued from the " + aQueueBean.get-
Name() + " queue");
 System.out.println(this.getClass().getName() + ":      the message subject is '" + aMessageBean.getSub-
ject() + "'");
                } else {
                System.out.println(this.getClass().getName() + ": A message was dequeued to a queue that
we are not interested in");
                }

            } catch (Exception ex) {
                System.out.println( this.getClass().getName() + ": " + ex );
            }
        }
    }

 /**
 *
 * A simple event handler to handle <code>AssignedMessageEvent</code> events.<p>
 * This handler simply traps the event and looks up the beans that were involved
 * in the event.  It then prints that information out to the screen.<p>
 *
 */
public class AssignedMessageEventListener implements EventListener, Serializable {
     private MessageHelperBean aMessageHelperBean = new MessageHelperBean();
  private AgentHelperBeananAgentHelperBean = new AgentHelperBean();

        /**
         * Listen for AssignedMessageEvent.<p>
         *
         * @param event The event that is being signalled
         * @since 1.0
```

```
         */
        public void signal( Event event ) {
AssignedMessageEvent ame = null;

            try {
                if ( ! (event instanceof AssignedMessageEvent)) {
                    System.out.println( this.getClass().getName() + ": Wrong type of Event, " +
event.toString() );
                    return;
                }
                ame = (AssignedMessageEvent) event;


                /*
                 * Get information about the message and the agent that that message
                 * was assigned to.
                 */
 MessageBeanaMessageBean = aMessageHelperBean.getMessageByKey(ame.getMessageKey());
 AgentBeananAgentBean = anAgentHelperBean.getAgentByKey(ame.getAgentKey());

 System.out.println(this.getClass().getName() + ": A message was assigned to the agent '" + anAgent-
Bean.getUserName() + "'");
 System.out.println(this.getClass().getName() + ": the message subject is '" + aMessageBean.getSubject() +
"'");

            } catch (Exception ex) {
                System.out.println( this.getClass().getName() + ": " + ex );
            }
        }
    }

 /**
 *
 * A simple event handler to handle <code>DeliveredMessageEvent</code> events.<p>
 * This handler simply traps the event and looks up the beans that were involved
 * in the event.  It then prints that information out to the screen.<p>
 *
 */
public class DeliveredMessageEventListener implements EventListener, Serializable {
     private MessageHelperBean aMessageHelperBean = new MessageHelperBean();
  private AgentHelperBeananAgentHelperBean = new AgentHelperBean();

        /**
         * Listen for DeliveredMessageEvent.<p>
         *
         * @param event The event that is being signalled
         * @since 1.0
         */
        public void signal( Event event ) {
         DeliveredMessageEvent dme = null;

            try {
                if ( ! (event instanceof DeliveredMessageEvent)) {
                    System.out.println( this.getClass().getName() + ": Wrong type of Event, " +
event.toString() );
                    return;
                }
                dme = (DeliveredMessageEvent) event;
```

```
                /*
                 * Get information about the message and the agent that that message
                 * was delivered to.
                 */
 MessageBeanaMessageBean = aMessageHelperBean.getMessageByKey(dme.getMessageKey());
 AgentBeananAgentBean = anAgentHelperBean.getAgentByKey(dme.getAgentKey());

 System.out.println(this.getClass().getName() + ": A message was delivered to the agent '" + anAgent-
Bean.getUserName() + "'");
 System.out.println(this.getClass().getName() + ":      the message subject is '" + aMessageBean.getSub-
ject() + "'");

            } catch (Exception ex) {
                System.out.println( this.getClass().getName() + ": " + ex );
            }
        }
    }

 /**
 *
 * A simple event handler to handle <code>UnassignedMessageEvent</code> events.<p>
 * This handler simply traps the event and looks up the beans that were involved
 * in the event.  It then prints that information out to the screen.<p>
 *
 */
public class UnassignedMessageEventListener implements EventListener, Serializable {
     private MessageHelperBean aMessageHelperBean = new MessageHelperBean();
  private AgentHelperBeananAgentHelperBean = new AgentHelperBean();

        /**
         * Listen for UnassignedMessageEvent.<p>
         *
         * @param event The event that is being signalled
         * @since 1.0
         */
        public void signal( Event event ) {
         UnassignedMessageEvent ume = null;

            try {
                if ( ! (event instanceof UnassignedMessageEvent)) {
                    System.out.println( this.getClass().getName() + ": Wrong type of Event, " +
event.toString() );
                    return;
                }
                ume = (UnassignedMessageEvent) event;

                /*
                 * Get information about the message and the agent that that message
                 * was assigned to.
                 */
 MessageBeanaMessageBean = aMessageHelperBean.getMessageByKey(ume.getMessageKey());
 AgentBeananAgentBean = anAgentHelperBean.getAgentByKey(ume.getAgentKey());

 System.out.println(this.getClass().getName() + ": A message was unassigned from the agent '" + anAgent-
Bean.getUserName() + "'");
```

```
 System.out.println(this.getClass().getName() + ":      the message subject is '" + aMessageBean.getSub-
ject() + "'");


            } catch (Exception ex) {
                System.out.println( this.getClass().getName() + ": " + ex );
            }
        }
    }


    /**
     *
     * Shutdown handler if the process is terminated externally.<p>
     *
     */
    public class Shutdown extends Thread {
        public Shutdown( ExampleEventHandler handler ) {
            this.handler = handler;
        }

        public void run() {
            try {
                System.out.println("Shutdown: Stopping the integration component" );
                if (handler == null) {
                    System.out.println("Shutdown: no handler specified" );
                    return;
                }
                handler.stop();
            } catch (Exception e) {
                System.out.println("Shutdown: Error stopping integration component: " + e.toString() );
            }
        }

        private ExampleEventHandler handler = null;
    }

    private EventRegistry registry = null;
    /**
     * Creates new ExampleEventHandler.<p>
     *
     * @param serverName The name of the Brightware Server
     *
     */
    public ExampleEventHandler( String serverName )
        throws Exception
    {
        // Setup a shutdown handler to ensure a clean exit
        Shutdown shutdown = new Shutdown( this );
        Runtime.getRuntime().addShutdownHook( shutdown );

        // Initialize the Event registry.  Connects to the server.
        try {
            System.out.println( this.getClass().getName() + ": Connecting to " + serverName );
            registry = new EventRegistry( "example", serverName, Integer.parseInt(PORT), false );
            System.out.println( this.getClass().getName() + ": Connected to " + serverName );
        } catch (Exception e) {
```

```
            System.out.println(this.getClass().getName() + ": Can not locate the server, " + serverName +
": " + e.toString() );
            throw e;
        }

        // Register event handlers
        try {
            registry.register( ServerStartedEvent.class, new ServerStartedEventListener() );
            registry.register( CreatedAgentEvent.class, new CreatedAgentEventListener() );
            registry.register( EnqueueEvent.class, new EnqueueEventListener());
            registry.register( DequeueEvent.class, new DequeueEventListener());
            registry.register( AssignedMessageEvent.class, new AssignedMessageEventListener());
            registry.register( DeliveredMessageEvent.class, new DeliveredMessageEventListener());
            registry.register( UnassignedMessageEvent.class, new UnassignedMessageEventListener());

        } catch (Exception e) {
            System.out.println( this.getClass().getName() + ": Failed to register listener: " +
e.toString() );
            throw e;
        }
    }
    /**
     *
     * A main for the Compatibility Event Service executable.<p>
     *
     */
    public static void main( String args[] )
    {
     DBI dbi = null;

        if (args.length < 1) {
            System.out.println( "Usage:" );
            System.out.println( "java com.firepond.example.exampleEventHandler hostname" );
            return;
        }
        String serverName = args[0];

        try {
         // Initialize the DBI
dbi = new DBI(serverName, PORT);

            // Create a handler
            ExampleEventHandler handler = new ExampleEventHandler( serverName );

            while( true ) {
                try {
                    Thread.sleep( 1000 );
                } catch (Exception e) {
                }
            }


        } catch (Exception e) {
            System.out.println( "main: " + e.toString() );
        }
    }
/**
 * When the JVM shuts down this method is called to unregister the event
```

```
 * listeners from the server.<p>
 *
 */
public void stop() throws Exception {
   // Unregister event handlers
   System.out.println("stop: Stopping the integration component.");
   try {
      registry.unregister(ServerStartedEvent.class);
      registry.unregister(CreatedAgentEvent.class);
      registry.unregister(EnqueueEvent.class);
      registry.unregister(DequeueEvent.class);
      registry.unregister(AssignedMessageEvent.class);
      registry.unregister(DeliveredMessageEvent.class);
      registry.unregister(UnassignedMessageEvent.class);

   } catch (Exception e) {
      System.out.println("stop: Failed to unregister from event");
      throw e;
   }

   // Terminate the registry - disconnects it from the server
   registry.terminate();

   System.out.println("stop: Stopped the integration component.");
}
}
```