C:\checkout\openii3\project\java\examples\org\any_openeai_enterprise\gateways\wareho
May 10, 2006 7:03:49 AM

```
/*****************************************************************************
 $Source: /cvs/repositories/openii3/project/java/examples/org/any_openeai_ente
rise/gateways/warehouse/EnterpriseWarehouseCommand.java,v $
 $Revision: 1.4 $
*****************************************************************************/

/*****************************************************************
This file is part of the OpenEAI sample, reference implementation,
and deployment management suite created by Tod Jackson
(tod@openeai.org) and Steve Wheat (steve@openeai.org) at
the University of Illinois Urbana-Champaign.

Copyright (C) 2002 The OpenEAI Software Foundation

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

For specific licensing details and examples of how this software
can be used to implement integrations for your enterprise, visit
http://www.OpenEai.org/licensing.
*/


package org.any_openeai_enterprise.gateways.warehouse;

import javax.jms.*;
import java.util.*;
import java.io.*;
import org.jdom.Document;
import org.jdom.Element;
import org.jdom.JDOMException;
import org.jdom.output.XMLOutputter;

import org.openeai.dbpool.*;
import org.openeai.xml.*;
import org.openeai.jms.producer.*;
import org.openeai.jms.consumer.*;
import org.openeai.jms.consumer.commands.*;
import org.openeai.layouts.*;
import org.openeai.config.*;
import org.openeai.moa.*;
import org.openeai.moa.objects.resources.*;
import org.openeai.moa.objects.testsuite.*;

import com.any_erp_vendor.moa.jmsobjects.person.v1_0.BasicPerson;
import com.any_erp_vendor.moa.objects.resources.v1_0.Date;
```

```
 57  import com.any_erp_vendor.moa.objects.resources.v1_0.Name;
 58  import com.any_erp_vendor.moa.objects.resources.v1_0.Address;
 59  import com.any_erp_vendor.moa.objects.resources.v1_0.Phone;
 60
 61  import java.sql.*;
 62  import java.net.URL;
 63  import java.io.*;
 64
 65  /**
 66  * This command is part of the OpenEAI Getting Started guide.  It is an example
     f
 67  * a gateway that consumes Synchronizaton messages from our ficticious ERP syst
 68  * from the ficticious "any-erp-vendor.com" company.  That ERP (com.any-erp-ven
     r.Gateway) is authoritative for the
 69  * following messages:
 70  * <ul>
 71  * <li>com.any-erp-vendor.Person/BasicPerson/1.0
 72  * <li>com.any-erp-vendor.Person/EmergencyContact/1.0
 73  * <li>com.any-erp-vendor.Employee/BasicEmployee/1.0
 74  * </ul>
 75  * This command will consume Synchronization messages for the
 76  * com.any-erp-vendor.Person.BasicPerson/1.0 message object.
 77  * This includes:  Create, Update, Delete.  When it consumes a Sync message, it
 78  * will pull relevant information from the message it consumes and store it in
     's
 79  * own repository.  This information will be stored in a more 'traditional' rep
     itory
 80  * meaning it will actually store data in individual columns in its database.
 81  * <P>
 82  * You can use the SelfServiceApplication or the TestSuite Application (also in
     uded in the Getting Started
 83  * distribution) to create, update, delete and query for people, employees and
     ergency
 84  * contacts in the com.any-erp-vendor.Gateway.  When you create, delete or upda
 85  * BasicPerson objects synchronization messages will be published and routed to
     his
 86  * gateway.
 87  * <P>
 88  * Note, this command is purposely simple.  It does not attempt to factor out m
     h of the
 89  * logic that could or should be.  This is so most of the work being done is co
     ained in this single
 90  * file.  Wile it is completely and fully functional, it should not be consider
 91  * the best way to actually code some of this stuff.  It is thouroughly comment
      so you
 92  * should be able to get a pretty good idea about what's going on at all the cr
     ical points
 93  * within the command execution.
 94  * <P>
 95  * To start this gateway you must execute the org.openeai.jms.consumer.MessageC
     sumerClient
 96  * class passing the path to the properties file you wish to use.
 97  * <P>
 98  * e.g. - java -cp [classpath] org.openeai.jms.consumer.MessageConsumerClient c
     GettingStarted.properties
 99  * <P>
100  * In the GettingStarted.properties file, you must un-comment the line that con
     ins:
101  * <P>
102  * <i>messageComponentName=org.any-openeai-enterprise.EnterpriseWarehouse</i>
```

```
103  * <P>
104  * This tells AppConfig which Messaging Component (gateway) it's configuring.
105  * Refer to the GettingStarted guide and installation instructions for more inf
     mation.
106  * <P>
107  * <B>Gateway Configuration</B>
108  * <P>
109  * These are the configuration parameters required by the sample org.any-openea
     enterprise.EnterpriseWarehouse.
110  * They are specified in the AnyOpenEaiEnterprise.xml Deployment Descriptor whi
      is provided
111  * in the sample enterprise distribution.
112  * This gateway is identified in the AnyOpenEaiEnterprise.xml Deployment
113  * Descriptor as <b>org.any-openeai-enterprise.EnterpriseWarehouse</b>.
114  * <P>
115  * Configuration for a gateway can be broken into two pieces:
116  * <ul>
117  * <li>Configuring the consumer to connect to a topic or queue and telling it w
     t commands it executes.
118  * <li>Configuring the command(s) executed by that consumer.
119  * </ul>
120  * <P>
121  * <b>Configuring the Consumer</b>
122  * <P>
123  * The items listed below can be found in the 'Configuration' Element associate
     to
124  * the org.any-openeai-enterprise.EnterpriseWarehouse gateway in the deployment
     escriptor.
125  * This Configuration element becomes the AppConfig
126  * object associated to this gateway <b>(remember, all OpenEAI gateways are sta
     ed through
127  * the org.openeai.jms.consumer.MessageConsumerClient class)</b>.
128  * <P>
129  * The consumer's Configuration is fairly simple.  It contains a LoggerConfig o
     ect that specifies
130  * how the log4j logger will behave and it contains a ConsumerConfigs element.
     elow is a description
131  * of the Consumer configuration.  Note, not all configurable items are mention
      but that
132  * information can be reviewed by looking at the core OpenEAI API Javadoc.
133  * <P>
134  * <TABLE BORDER=2 CELLPADDING=5 CELLSPACING=2>
135  * <TR>
136  * <TH>Configuration Object Name</TH>
137  * <TH>Required</TH>
138  * <TH>Description</TH>
139  * </TR>
140  * <TR HALIGN="left" VALIGN="top">
141  * <TD>WarehouseConsumer1</TD>
142  * <TD>yes</TD>
143  * <TD>An OpenEAI PubSubConsumer object configured to consume JMS messages off
144  * the 'cn=EnterpriseWarehouseTopic' topic.  When the com.any-erp-vendor.Gatewa
     publishes
145  * Sync messages, the EnterpriseTransRouter will route messages that contain me
     ageObjects of
146  * type BasicPerson to this topic.  This consumer will consume those messages a
      execute
147  * the org.any_openeai_enterprise.gateways.warehouse.EnterpriseWarehouseCommand
     this command)
148  * which is specified as a Command executed by this consumer.  As a matter of f
```

```
     t, it is the ONLY command
149  * that will ever be executed by this consumer since its 'isAbsolute' attribute
     s 'true'.
150  * That Command will perform the necessary business logic.  If any errors occur
     the command
151  * will publish an org.openeai.CoreMessaging/Sync/1.0/Error-Sync message that w
     l be logged
152  * by the 'Enterprise Logging Service'.
153  * <P>
154  * The Consumer's configuration specifies that it will connect to the broker
155  * and be ready to consume messages when it is initialized ('startOnInitializat
     n=true') use the
156  * InitialContextFactory 'org.exolab.jms.jndi.rmi.RmiJndiInitialContextFactory'
     o retrieve the JMS
157  * administered objects from the 'rmi://localhost:1099/JndiServer' ProviderURL
     he directory server normally).
158  * Then it will connect to the JMS Provider using the 'cn=EnterpriseWarehouseCo
     umerTCF' JMS TopicConnectionFactory
159  * administered object.
160  * <P>
161  * NOTE:  These are the parameters you change when you wish to switch JMS provi
     rs or use different JNDI stores
162  * to store your administered objects.
163  * <P>
164  * It will execute the Commands specified in a ThreadPool that has 30 pre-alloc
     ed Threads available to it and it will
165  * detect when the ThreadPool is busy and only add another Command execution to
     he ThreadPool when the ThreadPool has
166  * available threads to process the transaction.</TD>
167  * </TR>
168  * </TABLE>
169  * <P>
170  * <b>Configuring the Command(s) executed by the Consumer</b>
171  * <P>
172  * The EnterpriseWarehouseCommand command will NOT validate
173  * the JDOM Document created from the JMS Message that's passed to it (inboundX
     Validation=false).
174  * If you'd like it to validate incomming XML, you can change this attribute to
     rue.
175  * Additionally, it will not write the XML Document to a file (writeToFile=fals
      when it receives it.  As stated
176  * before, this command is the 'absolute' command associated to this consumer.
     o, no matter what "COMMAND_NAME" is
177  * associated to the JMS Message as it is consumed by the consumer, this comman
     will be executed.  The
178  * EnterpriseWarehouse (CommandName) is implemented by the
179  * org.any_openeai_enterprise.gateways.warehouse.EnterpriseWarehouseCommand cla
      (CommandClass) and
180  * it is a 'syncMessage' Command (type=syncMessage).
181  * <P>
182  * The items listed below can be found in the 'Configuration' Element associate
     to the
183  * EnterpriseWarehouse in the deployment descriptor.  This Configuration elemen
     becomes the AppConfig
184  * object associated to this Command that is used by the command when it is exe
     ted.
185  * <P>
186  * They are broken apart by Configuration object type.  This command
187  * will Publish Sync-Error messages if it ever has any problems performing the
     siness logic,
```

```
188  * it will use Message Objects to operate on the data contained in the message
     nsumed, it will
189  * use a DbConnectionPool to persist data and it will use several general Prope
     ies object.
190  * Therefore, there are four categories of configuration objects used:  Produce
     onfigs,
191  * MessageObjectConfigs, DbConnectionPoolConfigs and PropertyConfigs.
192  * <P>
193  * <TABLE BORDER=2 CELLPADDING=5 CELLSPACING=2>
194  * <TR>
195  * <TH>Configuration Object Type</TH>
196  * <TH>Configuration Object Name</TH>
197  * <TH>Required</TH>
198  * <TH>Description</TH>
199  * </TR>
200  * <TR HALIGN="left" VALIGN="top">
201  * <TD>ProducerConfigs</TD>
202  * <TD>SyncErrorPublisher</TD>
203  * <TD>yes</TD>
204  * <TD>An OpenEAI PubSubProducer object configured to publish Synchronization m
     sage to the
205  * 'cn=EnterpriseSyncErrorLoggerTopic'.  This will be used any time the command
     ncounters
206  * errors performing its business logic.  That Sync-Error message will then be
     gged
207  * by the 'Enterprise Logging Service' for follow-up by EAI support staff.
208  * <P>
209  * NOTE:  This is a required object for all SyncCommand implementations.  The g
     eway
210  * will not start if you have not specified a PubSubProducer named 'SyncErrorPu
     isher'.</TD>
211  * </TR>
212  * </TABLE>
213  * <P>
214  * <b>MessageObjectConfigs</b>
215  * <P>
216  * These configuration objects are used to configure all Message objects requir
      by this command.
217  * The message objects themselves are part of the com.any-erp-vendor Message Ob
     ct API (MOA) which
218  * was provided by the vendor because they've defined enterprise messages accor
     ng to the OpenEAI
219  * Message Protocol.  These objects were generated off of their enterprise mess
     e definitions (their Segments.dtd file).
220  * <P>
221  * The objects are configured to use the Enterprise Object (EO) documents suppl
     d by the com.any-erp-vendor organization that
222  * were also generated from their Segments.dtd file.
223  * In our sample enterprise, we've added some of our own formatting rules (data
     pes, lengths, scrubbers and translations)
224  * to these EO documents that our sample organization require.
225  * <P>
226  * The objects are used to operate on the data supplied in the sync messages as
     ell
227  * as to persist the relevant information to the database.
228  * <P>
229  * <TABLE BORDER=2 CELLPADDING=5 CELLSPACING=2>
230  * <TR>
231  * <TH>Configuration Object Name</TH>
232  * <TH>Required</TH>
```

```
233  * <TH>Description</TH>
234  * </TR>
235  * <TR HALIGN="left" VALIGN="top">
236  * <TD>BasicPerson.v1_0</TD>
237  * <TD>yes</TD>
238  * <TD>A com.any_erp_vendor.moa.jmsobjects.person.v1_0.BasicPerson Java object
     at was generated
239  * from the com/any-erp-vendor/Resources/1.0/Segments.dtd file.  This object wi
      be built
240  * from the contents of the messages consumed
241  * and used to store relevant information in the repository associated to this
     mmand.</TD>
242  * </TR>
243  * </TABLE>
244  * <P>
245  * <TABLE BORDER=2 CELLPADDING=5 CELLSPACING=2>
246  * <TR>
247  * <TH>Configuration Object Type</TH>
248  * <TH>Configuration Object Name</TH>
249  * <TH>Required</TH>
250  * <TH>Description</TH>
251  * </TR>
252  * <TR HALIGN="left" VALIGN="top">
253  * <TD>DbConnectionPoolConfigs</TD>
254  * <TD>WarehouseDbPool</TD>
255  * <TD>yes</TD>
256  * <TD>This is an OpenEAI org.openeai.dbpool.EnterpriseConnectionPool object.
     e command will
257  * retrieve a pre-established database connection from the pool and execute SQL
     tatements.  By default,
258  * the connections in the pool are configured to work with a MySql database.  H
     ever, scripts
259  * are provided in the Sample Enterprise distribution that will create Oracle s
     uctures as well.  To
260  * use these Oracle structures, you would simply change the pool configuration
     formation here.</TD>
261  * </TR>
262  * </TABLE>
263  * <P>
264  * <TABLE BORDER=2 CELLPADDING=5 CELLSPACING=2>
265  * <TR>
266  * <TH>Configuration Object Type</TH>
267  * <TH>Configuration Object Name</TH>
268  * <TH>Required</TH>
269  * <TH>Description</TH>
270  * </TR>
271  * <TR HALIGN="left" VALIGN="top">
272  * <TD>PropertyConfig</TD>
273  * <TD>WarehouseProperties</TD>
274  * <TD>yes</TD>
275  * <TD>These are 'general' properties that will be associated to this Command.<
     D>
276  * </TR>
277  * </TABLE>
278  * <P>
279  * <TABLE BORDER=2 CELLPADDING=5 CELLSPACING=2>
280  * <TR>
281  * <TH>Configuration Object Type</TH>
282  * <TH>Configuration Object Name</TH>
283  * <TH>Required</TH>
```

```
284  * <TH>Description</TH>
285  * </TR>
286  * <TR HALIGN="left" VALIGN="top">
287  * <TD>PropertyConfig</TD>
288  * <TD>SyncErrorSyncProperties</TD>
289  * <TD>yes</TD>
290  * <TD>These are REQUIRED properties that are used by SyncCommandImpl (this com
     nd's anscestor)
291  * to configure itself to publish Sync-Error messages.  All SyncCommand impleme
     ations must
292  * have a PropertyConfig object named 'SyncErrorSyncProperties' or they will no
     start.  These
293  * Properties specify the 'primed' Sync-Error document location so that resourc
     can be made
294  * available when the Command is first initialized and so it doesn't have to do
     t itself
295  * later when/if there's an error.</TD>
296  * </TR>
297  * </TABLE>
298  * <P>
299  * @author Tod Jackson (tod@openeai.org)
300  **/
301 public class EnterpriseWarehouseCommand
302   extends SyncCommandImpl
303   implements SyncCommand {
304
305   private EnterpriseConnectionPool m_connPool = null;
306
307   public EnterpriseWarehouseCommand(CommandConfig cConfig) throws Instantiatio
     xception {
308     super(cConfig);
309
310     try {
311       // default properties associated to this command (there could be others
     med differently).
312       PropertyConfig pConfig = (PropertyConfig)getAppConfig().getObject("Wareh
     seProperties");
313       setProperties(pConfig.getProperties());
314
315       // initialize db connection pool that will be used to store information
     om the
316       // message into our own repository
317       m_connPool = (EnterpriseConnectionPool)getAppConfig().getObject("Warehou
     DbPool");
318     }
319     catch (Exception e) {
320       logger.fatal("Error initializing 'EnterpriseWarehouse' command.");
321       logger.fatal(e.getMessage(), e);
322       throw new InstantiationException(e.getMessage());
323     }
324
325     logger.info("EnterpriseWarehouseCommand instantiated successfully.");
326   }
327
328   /**
329   * Takes the BasicPerson sync messages that have been published by the ERP sy
     em
330   * and routed to this gateway and stores relevant information from the consum
331   * messages in its own repository for "wharehousing" purposes like report wri
     ng etc.
```

```
332     **/
333     public void execute(int messageNumber, Message aMessage) throws CommandExcep
    on {
334       String errMessage = "";
335       Document inDoc = null;
336       try {
337         inDoc = initializeInput(messageNumber, aMessage);
338       }
339       catch (Exception e) {
340         errMessage = "Exception occurred processing input message in ConsumerCom
    nd.  Exception: " + e.getMessage();
341         ArrayList errors = new ArrayList();
342         errors.add(buildError("application", "SYNC-1001", errMessage));
343         publishSyncError(new Element("EmptyControlArea"), errors, e);
344         throw new CommandException(e.getMessage());
345       }
346
347       Element eControlArea = getControlArea(inDoc.getRootElement());
348       Element eDataArea = inDoc.getRootElement().getChild(DATA_AREA);
349
350       String msgAction = eControlArea.getAttribute(MESSAGE_ACTION).getValue();
351       String msgObject = eControlArea.getAttribute(MESSAGE_OBJECT).getValue();
352       String msgRelease = eControlArea.getAttribute(MESSAGE_RELEASE).getValue();
353       String msgCategory = eControlArea.getAttribute(MESSAGE_CATEGORY).getValue(
354       String msgType     = eControlArea.getAttribute(MESSAGE_TYPE).getValue();
355
356       // if the message object isn't one we're interested in (BasicPerson) publi
357       // a sync error and return...
358
359       String dataAreaChild = NEW_DATA;    // Default, this will work for Create
    d Update
360       if (msgAction.equalsIgnoreCase(DELETE_ACTION)) {
361         dataAreaChild = DELETE_DATA;
362       }
363
364       // Get the DataArea element out of the document
365       // This should correspond to one of our message objects.
366       Element eMessageObject = eDataArea.getChild(dataAreaChild).getChild(msgObj
    t);
367
368       if (eMessageObject == null) {
369         // Error!
370         errMessage = "Could not find an element at DataArea/" + dataAreaChild +
    " + msgObject +
371                      "in the " + msgObject + "-" + msgAction + "-Sync Document p
    sed in";
372         logger.fatal(errMessage);
373         logger.fatal("Message sent in is: \n" + getMessageBody(inDoc));
374
375         ArrayList errors = new ArrayList();
376         errors.add(buildError("application", "MSG-1001", errMessage));
377         publishSyncError(eControlArea, errors);
378         return;
379       }
380
381       // if the msgAction is Update, we'll need to get the baseline object out o
    the document as well.
382       Element eBaselineMessageObject = null;
383       if (msgAction.equalsIgnoreCase(UPDATE_ACTION)) {
384         // we need to create a baseline BasicPerson also so we have to get the b
```

```
    eline element...
385         eBaselineMessageObject = inDoc.getRootElement().
386                         getChild(DATA_AREA).
387                         getChild(BASELINE_DATA).
388                         getChild(msgObject);
389
390         if (eBaselineMessageObject == null) {
391           // Error!
392           errMessage = "Could not find a " + msgObject + " + element at DataArea
    aselineData/" + msgObject +
393                       " in the "+msgObject+"-"+msgAction+"-" + msgAction + " Do
    ment passed in";
394
395           ArrayList errors = logErrors("MSG-1012", errMessage, inDoc);
396           publishSyncError(eControlArea, errors);
397           return;
398         }
399       }
400
401       // get the object(s) we need to build from AppConfig
402       BasicPerson bPerson = null;
403       BasicPerson baselinePerson = null;
404
405       // build the message object from the element retrieved from the XML docume
    .
406       // this is done via the default XmlLayout manager.
407       try {
408         bPerson = (BasicPerson)getAppConfig().getObject(msgObject + "." + genera
    Release(msgRelease));
409         bPerson.buildObjectFromInput(eMessageObject);
410         if (msgAction.equalsIgnoreCase(UPDATE_ACTION)) {
411           baselinePerson = (BasicPerson)getAppConfig().getObject(msgObject + "."
     generateRelease(msgRelease));
412           baselinePerson.buildObjectFromInput(eBaselineMessageObject);
413         }
414       }
415       catch (Exception e) {
416         errMessage = "Exception occurred building the " + eMessageObject.getName
     + " Java object from the Element passed in.  Exception: " + e.getMessage();
417
418         ArrayList errors = logErrors("MSG-1003", errMessage, inDoc);
419         publishSyncError(eControlArea, errors, e);
420         return;
421       }
422
423       // now, perform the actual database activity based on the action associate
    to
424       // the message consumed.
425       if (msgAction.equalsIgnoreCase(CREATE_ACTION)) {
426         try {
427           createPerson(msgCategory, msgObject, msgRelease, bPerson);
428           return;
429         }
430         catch (Exception e) {
431           errMessage = "Exception occurred Creating the " + msgObject + " Java o
    ect in the database.  Exception: " + e.getMessage();
432           logger.fatal(errMessage, e);
433
434           ArrayList errors = logErrors("MSG-1003", errMessage, inDoc);
435           publishSyncError(eControlArea, errors, e);
```

```
436           return;
437         }
438       }
439       else if (msgAction.equalsIgnoreCase(DELETE_ACTION)) {
440         try {
441           String deleteAction = "delete";
442           deletePerson(msgCategory, msgObject, msgRelease, deleteAction, bPerson
443           return;
444         }
445         catch (Exception e) {
446           errMessage = "Exception occurred Deleting the " + msgObject + " Java o
    ect from the database.  Exception: " + e.getMessage();
447           logger.fatal(errMessage, e);
448
449           ArrayList errors = logErrors("MSG-1003", errMessage, inDoc);
450           publishSyncError(eControlArea, errors, e);
451           return;
452         }
453       }
454       else if (msgAction.equalsIgnoreCase(UPDATE_ACTION)) {
455         try {
456           // since we're not authoritative, we just want to do the update if the
    erson
457           // exists, if it doesn't exist, we could create the person?
458           updatePerson(msgCategory, msgObject, msgRelease, bPerson, baselinePers
    );
459         }
460         catch (Exception e) {
461           errMessage = "Exception occurred Updating the " + msgObject + " Java o
    ect in the database.  Exception: " + e.getMessage();
462           logger.fatal(errMessage, e);
463
464           ArrayList errors = logErrors("MSG-1003", errMessage, inDoc);
465           publishSyncError(eControlArea, errors, e);
466           return;
467         }
468       }
469       else {
470         // error, invalid message action.
471       }
472     }
473
474     protected BasicPerson retrievePerson(String instId) throws SQLException {
475       PreparedStatement stmt = null;
476       String sqlString =  "SELECT INST_ID, FIRST_NAME, MIDDLE_NAME, LAST_NAME, "
477                           "GENDER, ETHNICITY, BIRTH_DATE, SSN FROM T_PERSON " +
478                           "WHERE INST_ID=?";
479
480       java.sql.Connection conn = null;
481       EnterprisePooledConnection p = null;
482       try {
483         p = m_connPool.getExclusiveConnection();
484         conn = p.getConnection();
485         conn.setAutoCommit(true);
486       }
487       catch (Exception e) {
488         logger.fatal(e.getMessage(), e);
489         throw new SQLException(e.getMessage());
490       }
491
```

```
492     try {
493       stmt = conn.prepareStatement(sqlString);
494       stmt.clearParameters();
495       stmt.setString(1, instId);
496       logger.info("Retrieving Person record for InstitutionalId: " + instId);
497       ResultSet rs = stmt.executeQuery();
498       logger.info("Retreived Person record for InstitutionalId: " + instId);
499 
500       BasicPerson bPerson = (BasicPerson)getAppConfig().getObject("BasicPerson
    1_0");
501       Name aName = bPerson.newName();
502       boolean foundPerson = false;
503       while (rs.next()) {
504         foundPerson = true;
505         bPerson.setInstitutionalId(rs.getString(1));
506         aName.setFirstName(rs.getString(2));
507         aName.setMiddleName(rs.getString(3));
508         aName.setLastName(rs.getString(4));
509         bPerson.setName(aName);
510         bPerson.setGender(rs.getString(5));
511         bPerson.setEthnicity(rs.getString(6));
512         com.any_erp_vendor.moa.objects.resources.v1_0.Date bDate = bPerson.new
    rthDate();
513         java.sql.Date dbDate = rs.getDate(7);
514 
515         if (dbDate != null) {
516           Calendar calendar = new GregorianCalendar();
517           calendar.setTime(dbDate);
518 
519           bDate.setYear(Integer.toString(calendar.get(Calendar.YEAR)));
520           bDate.setMonth(Integer.toString(calendar.get(Calendar.MONTH)+1));
521           bDate.setDay(Integer.toString(calendar.get(Calendar.DAY_OF_MONTH)));
522 
523           logger.info("BirthDate from db is: " + bDate.toString());
524           bPerson.setBirthDate(bDate);
525         }
526         bPerson.setSocialSecurityNumber(rs.getString(8));
527       }
528 
529       stmt.close();
530 
531       // finally, release the connection.
532       m_connPool.releaseExclusiveConnection(p);
533       if (foundPerson) {
534         return bPerson;
535       }
536       else {
537         return null;
538       }
539     }
540     catch (Exception e) {
541       if (stmt != null) {
542         stmt.close();
543       }
544       logger.fatal(e.getMessage(), e);
545       throw new SQLException(e.getMessage());
546     }
547     finally {
548       m_connPool.releaseExclusiveConnection(p);
549     }
```

```
550     }
551     /**
552     * This method updates the information in T_PERSON for the BasicPerson object
    assed in.
553     * Since this is a SyncCommand (it's not authoritative), it simply takes the
    ta provided
554     * and updates its repository.  If anything had been changed in the local rep
    itory, it would
555     * be overridden.  Since's this is a warehouse gateway, data is only viewed,
    's not changed
556     * in the local repository.
557     * <P>
558     * In addition to updating the T_PERSON table, this method also looks at each
    ddress and Phone
559     * object contained within the newData BasicPerson object and compares them a
    inst the
560     * Address and Phone objects contained withing the baselinePerson BasicPerson
    bject.  Depending
561     * on that comparison, it may call any of the create, delete or update method
    associated to
562     * Addresses and Phones.  It uses the 'key' fields associated to the Address
    d Phone objects
563     * to determine if the Address or Phone transaction needs to be an insert, up
    te or delete.
564     * <P>
565     * @param msgCategory the 'messageCategory' attribute extracted from the
566     * ControlAreaRequest of the enterprise message.  Example: 'com.any-erp-vendo
    Person'.
567     * @param msgObject the 'messageObject' attribute extracted from the
568     * ControlAreaRequest of the enterprise message.  Example: 'BasicPerson'.
569     * @param msgRelease the 'messageRelease' attribute extracted from the
570     * ControlAreaRequest of the enterprise message.  Example: '1.1'.
571     * @param deleteAction the delete action associated to the request that was p
    led
572     * from the 'DataArea/DeleteData/DeleteAction' element in the enterprise mess
    e.
573     * @param newData the BasicPerson Java object that was built from the content
    of the
574     * enterprise message passed in (DataArea/NewData).  This is the new state of
    he BasicPerson.  When
575     * this method is complete, the state of the object in our local repository (
    PERSON, T_ADDRESS and T_PHONE)
576     * should match this object.
577     * @param baselinePerson the BasicPerson Java object that was built from the
    ntents of the
578     * enterprise message passed in (DataArea/BaselineData).  This is the baselin
    state of the BasicPerson.  This
579     * object is used to determine what transaction should be perfored on the Add
    ss and Phone
580     * objects (insert, update or deletes)
581     * @throws SQLException if errors occur deleting the message object from the
    tabase or if
582     * errors occur publishing the resulting Delete-Sync message.
583     **/
584     protected void updatePerson(String msgCategory, String msgObject, String msg
    lease, BasicPerson newData, BasicPerson baselinePerson) throws SQLException {
585       BasicPerson dbPerson = retrievePerson(newData.getInstitutionalId());
586       if (dbPerson == null) {
587         logger.info("Person did not exist in our repository, we'll just create t
    m.");
```

```
588         createPerson(msgCategory, msgObject, msgRelease, newData);
589         return;
590       }
591       PreparedStatement stmt = null;
592       String sqlString =  "UPDATE T_PERSON " +
593                           "SET FIRST_NAME=?, MIDDLE_NAME=?, LAST_NAME=?, GENDER=
     " +
594                           "ETHNICITY=?, BIRTH_DATE=?, SSN=?, MOD_DATE=?, MOD_USE
    ? " +
595                           "WHERE INST_ID=? AND MESSAGE_CATEGORY=? AND MESSAGE_OB
    CT=? AND MESSAGE_RELEASE=?";
596
597       String instId = newData.getInstitutionalId();
598       java.sql.Connection conn = null;
599       EnterprisePooledConnection p = null;
600       try {
601         p = m_connPool.getExclusiveConnection();
602         conn = p.getConnection();
603         conn.setAutoCommit(false);
604       }
605       catch (Exception e) {
606         logger.fatal(e.getMessage(), e);
607         throw new SQLException(e.getMessage());
608       }
609
610       try {
611         stmt = conn.prepareStatement(sqlString);
612         stmt.clearParameters();
613         stmt.setString(1, newData.getName().getFirstName());
614         stmt.setString(2, newData.getName().getMiddleName());
615         stmt.setString(3, newData.getName().getLastName());
616         stmt.setString(4, newData.getGender(getAppName()));
617         stmt.setString(5, newData.getEthnicity(getAppName()));
618         String sBirthDate = "";
619         if (newData.getBirthDate() != null && newData.getBirthDate().isEmpty() =
    false) {
620           sBirthDate = newData.getBirthDate().getMonth() + "/" +
621                        newData.getBirthDate().getDay() + "/" +
622                        newData.getBirthDate().getYear();
623           java.util.Date d = new java.util.Date();
624           java.text.SimpleDateFormat formatter = new java.text.SimpleDateFormat(
    MM/dd/yyyy" );
625           d = formatter.parse(sBirthDate);
626           stmt.setDate(6, new java.sql.Date(d.getTime()));
627         }
628         else {
629           stmt.setDate(6, null);
630         }
631         stmt.setString(7, newData.getSocialSecurityNumber());
632         stmt.setTimestamp(8, new java.sql.Timestamp(System.currentTimeMillis()))
633         stmt.setString(9, conn.getMetaData().getUserName());
634         stmt.setString(10, instId);
635         stmt.setString(11, msgCategory);
636         stmt.setString(12, msgObject);
637         stmt.setString(13, msgRelease);
638         logger.info("Attempting to Update " + msgObject + " record for Instituti
    alId: " + instId);
639         int rc = stmt.executeUpdate();
640         logger.info("Updated " + msgObject + " record for InstitutionalId: " + i
    tId + " rc=" + rc);
```

```
641         stmt.close();
642 
643         // now the address(s), using the same connection
644         // have to determine if we need to insert, delete or update any
645         logger.info("Starting Address update process...");
646         java.util.List newAddresses = newData.getAddress();
647         java.util.List oldAddresses = baselinePerson.getAddress();
648         for (int i=0; i<newAddresses.size(); i++) {
649           Address newAddress = (Address)newAddresses.get(i);
650           String newAddressKey = newAddress.getCombinedKeyValue();
651           boolean foundMatch = false;
652           for (int j=0; j<oldAddresses.size(); j++) {
653             Address oldAddress = (Address)oldAddresses.get(j);
654             String oldAddressKey = oldAddress.getCombinedKeyValue();
655             if (oldAddressKey.equalsIgnoreCase(newAddressKey)) {
656               // found a match, might have to update...
657               foundMatch = true;
658               if (newAddress.equals(oldAddress) == false) {
659                 // need to update individual address
660                 logger.info("updating existing address object.");
661                 updateAddress(conn, msgCategory, msgObject, msgRelease, instId,
    wAddress);
662               }
663             }
664           }
665           if (foundMatch == false) {
666             // need to create address
667             logger.info("creating new address object.");
668             createAddress(conn, msgCategory, msgObject, msgRelease, instId, newA
    ress);
669           }
670         }
671         // now we need to check the oldAddresses and delete any that don't
672         // exist in the newAddress list
673         for (int i=0; i<oldAddresses.size(); i++) {
674           Address oldAddress = (Address)oldAddresses.get(i);
675           String oldAddressKey = oldAddress.getCombinedKeyValue();
676           boolean foundMatch = false;
677           for (int j=0; j<newAddresses.size(); j++) {
678             Address newAddress = (Address)newAddresses.get(j);
679             String newAddressKey = newAddress.getCombinedKeyValue();
680             if (oldAddressKey.equalsIgnoreCase(newAddressKey)) {
681               // found a match, don't delete
682               foundMatch = true;
683             }
684           }
685           if (foundMatch == false) {
686             // must delete the oldAddress
687             logger.info("deleting old address object.");
688             deleteAddress(conn, msgCategory, msgObject, msgRelease, instId, oldA
    ress);
689           }
690         }
691         logger.info("Finished Address update process...");
692 
693         // now the phone(s), using the same connection
694         // have to determine if we need to insert, delete or update any
695         logger.info("Starting Phone update process...");
696         java.util.List newPhones = newData.getPhone();
697         java.util.List oldPhones = baselinePerson.getPhone();
```

```
698         for (int i=0; i<newPhones.size(); i++) {
699           Phone newPhone = (Phone)newPhones.get(i);
700           String newPhoneKey = newPhone.getCombinedKeyValue();
701           boolean foundMatch = false;
702           for (int j=0; j<oldPhones.size(); j++) {
703             Phone oldPhone = (Phone)oldPhones.get(j);
704             String oldPhoneKey = oldPhone.getCombinedKeyValue();
705             if (oldPhoneKey.equalsIgnoreCase(newPhoneKey)) {
706               // found a match, might have to update...
707               foundMatch = true;
708               if (newPhone.equals(oldPhone) == false) {
709                 // need to update individual phone
710                 logger.info("updating existing phone object.");
711                 updatePhone(conn, msgCategory, msgObject, msgRelease, instId, ne
    hone);
712               }
713             }
714           }
715           if (foundMatch == false) {
716             // need to create phone
717             logger.info("creating new address object.");
718             createPhone(conn, msgCategory, msgObject, msgRelease, instId, newPho
    );
719           }
720         }
721         // now we need to check the oldPhones and delete any that don't
722         // exist in the newAddress list
723         for (int i=0; i<oldPhones.size(); i++) {
724           Phone oldPhone = (Phone)oldPhones.get(i);
725           String oldPhoneKey = oldPhone.getCombinedKeyValue();
726           boolean foundMatch = false;
727           for (int j=0; j<newPhones.size(); j++) {
728             Phone newPhone = (Phone)newPhones.get(j);
729             String newPhoneKey = newPhone.getCombinedKeyValue();
730             if (oldPhoneKey.equalsIgnoreCase(newPhoneKey)) {
731               // found a match, don't delete
732               foundMatch = true;
733             }
734           }
735           if (foundMatch == false) {
736             // must delete the oldPhone
737             logger.info("deleting old address object.");
738             deletePhone(conn, msgCategory, msgObject, msgRelease, instId, oldPho
    );
739           }
740         }
741         logger.info("Finished Phone update process...");
742
743         // now the email(s), using the same connection
744
745         // finally, commit it all if everything went okay.
746         conn.commit();
747         m_connPool.releaseExclusiveConnection(p);
748       }
749       catch (Exception e) {
750         conn.rollback();
751         if (stmt != null) {
752           stmt.close();
753         }
754         logger.fatal(e.getMessage(), e);
```

```
755         throw new SQLException(e.getMessage());
756       }
757       finally {
758         m_connPool.releaseExclusiveConnection(p);
759       }
760     }
761     /**
762     * This method first deletes all Address and Phone objects contained within t
     BasicPerson
763     * object passed in by calling the deleteAddress and deletePhone methods.  Th
     it
764     * deletes the T_PERSON row associated to the BasicPerson message object pass
     in.
765     * <P>
766     * @param msgCategory the 'messageCategory' attribute extracted from the
767     * ControlAreaRequest of the enterprise message.  Example: 'com.any-erp-vendo
    Person'.
768     * @param msgObject the 'messageObject' attribute extracted from the
769     * ControlAreaRequest of the enterprise message.  Example: 'BasicPerson'.
770     * @param msgRelease the 'messageRelease' attribute extracted from the
771     * ControlAreaRequest of the enterprise message.  Example: '1.1'.
772     * @param deleteAction the delete action associated to the request that was p
    led
773     * from the 'DataArea/DeleteData/DeleteAction' element in the enterprise mess
    e.
774     * @param bPerson the BasicPerson Java object that was built from the content
    of the
775     * enterprise message passed.  The InstitutionalId from this object will be u
    d
776     * to delete data from the T_PERSON table.  Additionally, Address and Phone o
    ects will
777     * be extracted from the object and deleted from their respective tables.
778     * @throws SQLException if errors occur deleting the message object from the
    tabase or if
779     * errors occur publishing the resulting Delete-Sync message.
780     **/
781     protected void deletePerson(String msgCategory, String msgObject, String msg
    lease, String deleteAction, BasicPerson bPerson) throws SQLException {
782       PreparedStatement stmt = null;
783       String sqlString = "delete from T_PERSON " +
784                          "where inst_id=? AND MESSAGE_CATEGORY=? AND MESSAGE_
    JECT=? AND MESSAGE_RELEASE=?";
785
786       String instId = bPerson.getInstitutionalId();
787       java.sql.Connection conn = null;
788       EnterprisePooledConnection p = null;
789       try {
790         p = m_connPool.getExclusiveConnection();
791         conn = p.getConnection();
792         conn.setAutoCommit(false);
793       }
794       catch (Exception e) {
795         logger.fatal(e.getMessage(), e);
796         throw new SQLException(e.getMessage());
797       }
798
799       try {
800         // first the address(s), using the same connection
801         for (int i=0; i<bPerson.getAddress().size(); i++) {
802           Address a = bPerson.getAddress(i);
```

```
803           deleteAddress(conn, msgCategory, msgObject, msgRelease, instId, a);
804         }
805
806         // now the phone(s), using the same connection
807         for (int i=0; i<bPerson.getPhone().size(); i++) {
808           Phone phone = bPerson.getPhone(i);
809           deletePhone(conn, msgCategory, msgObject, msgRelease, instId, phone);
810         }
811
812         // now the email(s), using the same connection
813
814         // now the actual person record
815         stmt = conn.prepareStatement(sqlString);
816         stmt.clearParameters();
817         stmt.setString(1, instId);
818         stmt.setString(2, msgCategory);
819         stmt.setString(3, msgObject);
820         stmt.setString(4, msgRelease);
821         logger.info("Attempting to delete " + msgObject + " record for Instituti
    alId: " + instId);
822         int rc = stmt.executeUpdate();
823         logger.info("Deleted " + msgObject + " record for InstitutionalId: " + i
    tId);
824         stmt.close();
825
826         // finally, commit it all if everything went okay.
827         conn.commit();
828         m_connPool.releaseExclusiveConnection(p);
829       }
830       catch (Exception e) {
831         conn.rollback();
832         if (stmt != null) {
833           stmt.close();
834         }
835         logger.fatal(e.getMessage(), e);
836         throw new SQLException(e.getMessage());
837       }
838       finally {
839         m_connPool.releaseExclusiveConnection(p);
840       }
841     }
842     /**
843     * Inserts the InstitutionalId, FirstName, MiddleName, LastName, Gender, Ethn
    ity
844     * BirthDate and SocialSecurityNumber fields into the T_PERSON table.  Note,
    at the
845     * values stored for fields like Gender and Ethnicity are reverse translated
    om
846     * "enterprise values" to "application values" that are specified in the Basi
    ersonEO.xml
847     * file (EO doc) for the BasicPerson object being used by this command.
848     * <P>
849     * Then it goes through each Address object contained within the BasicPerson
    ject passed in
850     * and inserts it into the T_ADDRESS table by calling the 'createAddress' met
    d.
851     * <P>
852     * Then it goes through each Phone object containe within the BasicPerson obj
    t passed in and
853     * inserts it into the T_PHONE table calling the 'createPhone' method.
```

```
854     * <P>
855     * @param msgCategory the 'messageCategory' attribute extracted from the
856     * ControlAreaRequest of the enterprise message.  Example: 'com.any-erp-vendo
    Person'.
857     * @param msgObject the 'messageObject' attribute extracted from the
858     * ControlAreaRequest of the enterprise message.  Example: 'BasicPerson'.
859     * @param msgRelease the 'messageRelease' attribute extracted from the
860     * ControlAreaRequest of the enterprise message.  Example: '1.1'.
861     * @param bPerson the BasicPerson Java object that was built from the content
    of the
862     * enterprise message passed.  Individual fields from this object will be ext
    cted
863     * and inserted into the T_PERSON table.  Additionally, Address and Phone obj
    ts will
864     * be extracted from the object and inserted into their respective tables.
865     * @throws SQLException if errors occur deleting the message object from the
    tabase or if
866     * errors occur publishing the resulting Delete-Sync message.
867     **/
868     protected void createPerson(String msgCategory, String msgObject, String msg
    lease, BasicPerson bPerson) throws SQLException {
869       PreparedStatement insertStmt = null;
870       String insertString = "INSERT INTO T_PERSON (INST_ID, MESSAGE_CATEGORY, ME
    AGE_OBJECT, MESSAGE_RELEASE, FIRST_NAME, " +
871                             "MIDDLE_NAME, LAST_NAME, GENDER, ETHNICITY, " +
872                             "BIRTH_DATE, SSN, " +
873                             "CREATE_DATE, CREATE_USER, MOD_DATE, MOD_USER) " +
874                             "VALUES (?,?,?,?,?,?,?,?,?,?,?,?,?,?,?)";
875
876       // get the institutional id from the BasicPerson
877       String instId = bPerson.getInstitutionalId();
878
879       // get an exclusive connection from our OpenEAI database connection pool.
880       // this is required if we're going to perform database activities within
881       // a transaction (autocommit=false)
882       java.sql.Connection conn = null;
883       EnterprisePooledConnection p = null;
884       try {
885         p = m_connPool.getExclusiveConnection();
886         conn = p.getConnection();
887         conn.setAutoCommit(false);
888       }
889       catch (Exception e) {
890         logger.fatal(e.getMessage(), e);
891         throw new SQLException(e.getMessage());
892       }
893
894       // Now, prepare our SQL statement setting values that should be inserted i
    o
895       // the T_PERSON table.  Note, when fields like Gender and Ethnicity are re
    ieved
896       // from the BasicPerson object, we're passing our application name.  This
    ll
897       // result in the enterprise values contained in the BasicPerson object bei
     reverse
898       // translated into application specific values for our system.
899       try {
900         insertStmt = conn.prepareStatement(insertString);
901         insertStmt.clearParameters();
902         insertStmt.setString(1, instId);
```

```
903         insertStmt.setString(2, msgCategory);
904         insertStmt.setString(3, msgObject);
905         insertStmt.setString(4, msgRelease);
906         insertStmt.setString(5, bPerson.getName().getFirstName());
907         insertStmt.setString(6, bPerson.getName().getMiddleName());
908         insertStmt.setString(7, bPerson.getName().getLastName());
909         insertStmt.setString(8, bPerson.getGender(getAppName()));
910         insertStmt.setString(9, bPerson.getEthnicity(getAppName()));
911         String sBirthDate = "";
912         if (bPerson.getBirthDate() != null && bPerson.getBirthDate().isEmpty() =
      false) {
913           sBirthDate = bPerson.getBirthDate().getMonth() + "/" +
914                        bPerson.getBirthDate().getDay() + "/" +
915                        bPerson.getBirthDate().getYear();
916           java.util.Date d = new java.util.Date();
917           java.text.SimpleDateFormat formatter = new java.text.SimpleDateFormat(
      MM/dd/yyyy" );
918           d = formatter.parse(sBirthDate);
919           insertStmt.setDate(10, new java.sql.Date(d.getTime()));
920         }
921         else {
922           insertStmt.setDate(10, null);
923         }
924         insertStmt.setString(11, bPerson.getSocialSecurityNumber());
925         insertStmt.setTimestamp(12, new java.sql.Timestamp(System.currentTimeMil
      s()));
926         insertStmt.setString(13, conn.getMetaData().getUserName());
927         insertStmt.setTimestamp(14, new java.sql.Timestamp(System.currentTimeMil
      s()));
928         insertStmt.setString(15, conn.getMetaData().getUserName());
929         logger.info("Attempting to insert " + msgObject + " record for Instituti
      alId: " + instId);
930         logger.debug("Gender appValue: '" + bPerson.getGender(getAppName()) + "'
      ;
931         logger.debug("Ethnicity appValue: '" + bPerson.getEthnicity(getAppName()
      + "'");
932         int insertRc = insertStmt.executeUpdate();
933         logger.info("Inserted " + msgObject + " record for InstitutionalId: " +
      stId);
934         insertStmt.close();
935
936         // now the address(s), using the same connection
937         for (int i=0; i<bPerson.getAddress().size(); i++) {
938           Address a = bPerson.getAddress(i);
939           createAddress(conn, msgCategory, msgObject, msgRelease, instId, a);
940         }
941
942         // now the phone(s), using the same connection
943         for (int i=0; i<bPerson.getPhone().size(); i++) {
944           Phone phone = bPerson.getPhone(i);
945           createPhone(conn, msgCategory, msgObject, msgRelease, instId, phone);
946         }
947
948         // now the email(s), using the same connection
949
950         // finally, commit it all if everything went okay.
951         conn.commit();
952         m_connPool.releaseExclusiveConnection(p);
953       }
954       catch (Exception e) {
```

```
955         conn.rollback();
956         if (insertStmt != null) {
957           insertStmt.close();
958         }
959         logger.fatal(e.getMessage(), e);
960         throw new SQLException(e.getMessage());
961       }
962       finally {
963         m_connPool.releaseExclusiveConnection(p);
964       }
965     }
966
967     /**
968     * Deletes a single Address row from the T_ADDRESS table.  Uses the msgCatego
      ,
969     * msgObject, msgRelease and instId parameters passed in as well as the Addre
       type associated
970     * to the Address object passed in as the delete criteria.
971     * <P>
972     * @param conn a pre-established database connection.  Since an Address is on
       ever
973     * deleted as part of a larger transaction (BasicPerson-Delete or Update) thi
974     * delete must occur within that same transaction, therefore, this method doe
      not
975     * commit the transaction.
976     * @param msgCategory the 'messageCategory' attribute extracted from the
977     * ControlAreaRequest of the enterprise message.  Example: 'com.any-erp-vendo
      Person'.
978     * @param msgObject the 'messageObject' attribute extracted from the
979     * ControlAreaRequest of the enterprise message.  Example: 'BasicPerson'.
980     * @param msgRelease the 'messageRelease' attribute extracted from the
981     * ControlAreaRequest of the enterprise message.  Example: '1.1'.
982     * @param instId associated to the person we're deleting an address for.
983     * @param a the Address Java object that was pulled from the BasicPerson obje
       that was
984     * built from the enterprise message passed.
985     * @throws SQLException if errors occur deleting the Address from the databas
986     **/
987     protected void deleteAddress(java.sql.Connection conn, String msgCategory, S
      ing msgObject, String msgRelease, String instId, Address a) throws SQLExceptio
      {
988       PreparedStatement stmt = null;
989       String sqlString = "delete from T_ADDRESS " +
990                          "where inst_id=? and MESSAGE_CATEGORY=? AND " +
991                          "MESSAGE_OBJECT=? AND MESSAGE_RELEASE=? AND " +
992                          "ADDR_TYPE=?";
993
994       try {
995         // delete the individual address record passed in
996         stmt = conn.prepareStatement(sqlString);
997         stmt.clearParameters();
998         stmt.setString(1, instId);
999         stmt.setString(2, msgCategory);
1000        stmt.setString(3, msgObject);
1001        stmt.setString(4, msgRelease);
1002        stmt.setString(5, a.getType(getAppName()));
1003        logger.info("Attempting to delete Address record " + a.toString() + " fo
      InstitutionalId: " + instId);
1004        int rc = stmt.executeUpdate();
1005        logger.info("Deleted Address record " + a.toString() + " for Institution
```

```
      Id: " + instId + " rc=" + rc);
1006        stmt.close();
1007      }
1008      catch (Exception e) {
1009        conn.rollback();
1010        if (stmt != null) {
1011          stmt.close();
1012        }
1013        logger.fatal(e.getMessage(), e);
1014        throw new SQLException(e.getMessage());
1015      }
1016    }
1017    /**
1018     * Updates a single Address row from the T_ADDRESS table.  Uses the msgCatego
      ,
1019     * msgObject, msgRelease and instId parameters passed in as well as the Addre
       type associated
1020     * to the Address object passed in as the update criteria.
1021     * <P>
1022     * @param conn a pre-established database connection.  Since an Address is on
       ever
1023     * updated as part of a larger transaction (BasicPerson-Update) this
1024     * update must occur within that same transaction, therefore, this method doe
      not
1025     * commit the transaction.
1026     * @param msgCategory the 'messageCategory' attribute extracted from the
1027     * ControlAreaRequest of the enterprise message.  Example: 'com.any-erp-vendo
      Person'.
1028     * @param msgObject the 'messageObject' attribute extracted from the
1029     * ControlAreaRequest of the enterprise message.  Example: 'BasicPerson'.
1030     * @param msgRelease the 'messageRelease' attribute extracted from the
1031     * ControlAreaRequest of the enterprise message.  Example: '1.1'.
1032     * @param instId associated to the person we're updating an address for.
1033     * @param a the Address Java object that was pulled from the BasicPerson obje
       that was
1034     * built from the enterprise message passed in.
1035     * @throws SQLException if errors occur updating the Address from the databas
1036     **/
1037    protected void updateAddress(java.sql.Connection conn, String msgCategory, S
      ing msgObject, String msgRelease, String instId, Address a) throws SQLExceptio
      {
1038      PreparedStatement stmt = null;
1039      String sqlString = "UPDATE T_ADDRESS SET STREET1=?, STREET2=?, " +
1040                         "STREET3=?, CITY=?, STATE_CODE=?, ZIP_CODE=?, EFFECT
      E_DATE=?, " +
1041                         "MOD_DATE=?, MOD_USER=? " +
1042                         "WHERE INST_ID=? AND MESSAGE_CATEGORY=? AND MESSAGE_
      JECT=? AND " +
1043                         "MESSAGE_RELEASE=? AND ADDR_TYPE=?";
1044
1045      try {
1046        stmt = conn.prepareStatement(sqlString);
1047        stmt.clearParameters();
1048        stmt.setString(1, a.getStreet1(getAppName()));
1049        stmt.setString(2, a.getStreet2(getAppName()));
1050        stmt.setString(3, a.getStreet3(getAppName()));
1051        stmt.setString(4, a.getCityOrLocality(getAppName()));
1052        stmt.setString(5, a.getStateOrProvince(getAppName()));
1053        stmt.setString(6, a.getZipOrPostalCode(getAppName()));
1054        String sEffDate = "";
```

```
1055        if (a.getEffectiveDate().isEmpty() == false) {
1056          sEffDate = a.getEffectiveDate().getMonth() + "/" +
1057                      a.getEffectiveDate().getDay() + "/" +
1058                      a.getEffectiveDate().getYear();
1059          java.util.Date d = new java.util.Date();
1060          java.text.SimpleDateFormat formatter = new java.text.SimpleDateFormat(
      MM/dd/yyyy" );
1061          d = formatter.parse(sEffDate);
1062          stmt.setDate(7, new java.sql.Date(d.getTime()));
1063        }
1064        else {
1065          stmt.setDate(7, null);
1066        }
1067        stmt.setTimestamp(8, new java.sql.Timestamp(System.currentTimeMillis()))
1068        stmt.setString(9, conn.getMetaData().getUserName());
1069        stmt.setString(10, instId);
1070        stmt.setString(11, msgCategory);
1071        stmt.setString(12, msgObject);
1072        stmt.setString(13, msgRelease);
1073        stmt.setString(14, a.getType(getAppName()));
1074        logger.info("Attempting to update Address record " + a.toString() + " fo
      InstitutionalId: " + instId);
1075        int rc = stmt.executeUpdate();
1076        logger.info("Updated Address record " + a.toString() + " for Institution
      Id: " + instId + " rc=" + rc);
1077        stmt.close();
1078      }
1079      catch (Exception e) {
1080        if (stmt != null) {
1081          stmt.close();
1082        }
1083        logger.fatal(e.getMessage(), e);
1084        throw new SQLException(e.getMessage());
1085      }
1086    }
1087    /**
1088     * Creates a single Address row in the T_ADDRESS table.  Uses the msgCategory
1089     * msgObject, msgRelease and instId parameters passed in as well as the Addre
       type associated
1090     * to the Address object passed in as the data to create.
1091     * <P>
1092     * @param conn a pre-established database connection.  Since an Address is on
       ever
1093     * created as part of a larger transaction (BasicPerson-Create or Update) thi
1094     * insert must occur within that same transaction, therefore, this method doe
      not
1095     * commit the transaction.
1096     * @param msgCategory the 'messageCategory' attribute extracted from the
1097     * ControlAreaRequest of the enterprise message.  Example: 'com.any-erp-vendo
      Person'.
1098     * @param msgObject the 'messageObject' attribute extracted from the
1099     * ControlAreaRequest of the enterprise message.  Example: 'BasicPerson'.
1100     * @param msgRelease the 'messageRelease' attribute extracted from the
1101     * ControlAreaRequest of the enterprise message.  Example: '1.1'.
1102     * @param instId associated to the person we're creating an address for.
1103     * @param a the Address Java object that was pulled from the BasicPerson obje
       that was
1104     * built from the enterprise message passed.
1105     * @throws SQLException if errors occur creating the Address in the database.
1106     **/
```

```
1107    protected void createAddress(java.sql.Connection conn, String msgCategory, S
      ing msgObject, String msgRelease, String instId, Address a) throws SQLExceptio
      {
1108      PreparedStatement insertStmt = null;
1109      String insertString = "INSERT INTO T_ADDRESS (INST_ID, MESSAGE_CATEGORY, "
1110                            "MESSAGE_OBJECT, MESSAGE_RELEASE, ADDR_TYPE, STREET1
      " +
1111                            "STREET2, STREET3, CITY, STATE_CODE, ZIP_CODE, EFFEC
      VE_DATE, " +
1112                            "CREATE_DATE, CREATE_USER, MOD_DATE, MOD_USER) " +
1113                            "VALUES (?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?)";
1114
1115      try {
1116        insertStmt = conn.prepareStatement(insertString);
1117        insertStmt.clearParameters();
1118        insertStmt.setString(1, instId);
1119        insertStmt.setString(2, msgCategory);
1120        insertStmt.setString(3, msgObject);
1121        insertStmt.setString(4, msgRelease);
1122        insertStmt.setString(5, a.getType(getAppName()));
1123        insertStmt.setString(6, a.getStreet1(getAppName()));
1124        insertStmt.setString(7, a.getStreet2(getAppName()));
1125        insertStmt.setString(8, a.getStreet3(getAppName()));
1126        insertStmt.setString(9, a.getCityOrLocality(getAppName()));
1127        insertStmt.setString(10, a.getStateOrProvince(getAppName()));
1128        insertStmt.setString(11, a.getZipOrPostalCode(getAppName()));
1129        String sEffDate = "";
1130        if (a.getEffectiveDate().isEmpty() == false) {
1131          sEffDate = a.getEffectiveDate().getMonth() + "/" +
1132                     a.getEffectiveDate().getDay() + "/" +
1133                     a.getEffectiveDate().getYear();
1134          java.util.Date d = new java.util.Date();
1135          java.text.SimpleDateFormat formatter = new java.text.SimpleDateFormat(
      MM/dd/yyyy" );
1136          d = formatter.parse(sEffDate);
1137          insertStmt.setDate(12, new java.sql.Date(d.getTime()));
1138        }
1139        else {
1140          insertStmt.setDate(12, null);
1141        }
1142        insertStmt.setTimestamp(13, new java.sql.Timestamp(System.currentTimeMil
      s()));
1143        insertStmt.setString(14, conn.getMetaData().getUserName());
1144        insertStmt.setTimestamp(15, new java.sql.Timestamp(System.currentTimeMil
      s()));
1145        insertStmt.setString(16, conn.getMetaData().getUserName());
1146        logger.info("Attempting to insert Address record " + a.toString() + " fo
      InstitutionalId: " + instId);
1147        int insertRc = insertStmt.executeUpdate();
1148        logger.info("Inserted Address record " + a.toString() + " for Institutio
      lId: " + instId);
1149        insertStmt.close();
1150      }
1151      catch (Exception e) {
1152        if (insertStmt != null) {
1153          insertStmt.close();
1154        }
1155        logger.fatal(e.getMessage(), e);
1156        throw new SQLException(e.getMessage());
1157      }
```

```
1158    }
1159
1160    /**
1161     * Deletes a single Phone row from the T_PHONE table.  Uses the msgCategory,
1162     * msgObject, msgRelease and instId parameters passed in as well as the Phone
      ype associated
1163     * to the Phone object passed in as the delete criteria.
1164     * <P>
1165     * @param conn a pre-established database connection.  Since an Address is on
       ever
1166     * deleted as part of a larger transaction (BasicPerson-Delete or Update) thi
1167     * delete must occur within that same transaction, therefore, this method doe
      not
1168     * commit the transaction.
1169     * @param msgCategory the 'messageCategory' attribute extracted from the
1170     * ControlAreaRequest of the enterprise message.  Example: 'com.any-erp-vendo
      Person'.
1171     * @param msgObject the 'messageObject' attribute extracted from the
1172     * ControlAreaRequest of the enterprise message.  Example: 'BasicPerson'.
1173     * @param msgRelease the 'messageRelease' attribute extracted from the
1174     * ControlAreaRequest of the enterprise message.  Example: '1.1'.
1175     * @param instId associated to the person we're deleting an address for.
1176     * @param p the Phone Java object that was pulled from the BasicPerson object
      hat was
1177     * built from the enterprise message passed.
1178     * @throws SQLException if errors occur deleting the Address from the databas
1179     **/
1180    protected void deletePhone(java.sql.Connection conn, String msgCategory, Str
      g msgObject, String msgRelease, String instId, Phone p) throws SQLException {
1181      PreparedStatement stmt = null;
1182      String sqlString = "delete from T_PHONE " +
1183                         "where inst_id=? and MESSAGE_CATEGORY=? AND " +
1184                         "MESSAGE_OBJECT=? AND MESSAGE_RELEASE=? AND " +
1185                         "PHONE_TYPE=?";
1186
1187      try {
1188        // delete the individual address record passed in
1189        stmt = conn.prepareStatement(sqlString);
1190        stmt.clearParameters();
1191        stmt.setString(1, instId);
1192        stmt.setString(2, msgCategory);
1193        stmt.setString(3, msgObject);
1194        stmt.setString(4, msgRelease);
1195        stmt.setString(5, p.getType(getAppName()));
1196        logger.info("Attempting to delete Phone record " + p.toString() + " for
      stitutionalId: " + instId);
1197        int rc = stmt.executeUpdate();
1198        logger.info("Deleted Phone record " + p.toString() + " for Institutional
      : " + instId + " rc=" + rc);
1199        stmt.close();
1200      }
1201      catch (Exception e) {
1202        conn.rollback();
1203        if (stmt != null) {
1204          stmt.close();
1205        }
1206        logger.fatal(e.getMessage(), e);
1207        throw new SQLException(e.getMessage());
1208      }
1209    }
```

```
1210    /**
1211     * Updates a single Phone row from the T_ADDRESS table.  Uses the msgCategory
1212     * msgObject, msgRelease and instId parameters passed in as well as the Phone
      ype associated
1213     * to the Phone object passed in as the update criteria.
1214     * <P>
1215     * @param conn a pre-established database connection.  Since an Phone is only
      ver
1216     * updated as part of a larger transaction (BasicPerson-Update) this
1217     * update must occur within that same transaction, therefore, this method doe
      not
1218     * commit the transaction.
1219     * @param msgCategory the 'messageCategory' attribute extracted from the
1220     * ControlAreaRequest of the enterprise message.  Example: 'com.any-erp-vendo
      Person'.
1221     * @param msgObject the 'messageObject' attribute extracted from the
1222     * ControlAreaRequest of the enterprise message.  Example: 'BasicPerson'.
1223     * @param msgRelease the 'messageRelease' attribute extracted from the
1224     * ControlAreaRequest of the enterprise message.  Example: '1.1'.
1225     * @param instId associated to the person we're updating an address for.
1226     * @param p the Phone Java object that was pulled from the BasicPerson object
      hat was
1227     * built from the enterprise message passed in.
1228     * @throws SQLException if errors occur updating the Phone from the database.
1229     **/
1230    protected void updatePhone(java.sql.Connection conn, String msgCategory, Str
      g msgObject, String msgRelease, String instId, Phone p) throws SQLException {
1231      PreparedStatement stmt = null;
1232      String sqlString = "UPDATE T_PHONE SET PHONE_AREA=?, " +
1233                         "PHONE_NUMBER=?, MOD_DATE=?, MOD_USER=? " +
1234                         "WHERE INST_ID=? AND MESSAGE_CATEGORY=? AND " +
1235                         "MESSAGE_OBJECT=? AND MESSAGE_RELEASE=? AND PHONE_TY
      =?";
1236
1237      try {
1238        stmt = conn.prepareStatement(sqlString);
1239        stmt.clearParameters();
1240        stmt.setString(1, p.getPhoneArea());
1241        stmt.setString(2, p.getPhoneNumber());
1242        stmt.setTimestamp(3, new java.sql.Timestamp(System.currentTimeMillis()))
1243        stmt.setString(4, conn.getMetaData().getUserName());
1244        stmt.setString(5, instId);
1245        stmt.setString(6, msgCategory);
1246        stmt.setString(7, msgObject);
1247        stmt.setString(8, msgRelease);
1248        stmt.setString(9, p.getType(getAppName()));
1249        logger.info("Attempting to update Phone record " + p.toString() + " for
      stitutionalId: " + instId);
1250        int rc = stmt.executeUpdate();
1251        logger.info("Updated Phone record " + p.toString() + " for Institutional
      : " + instId + " rc=" + rc);
1252        stmt.close();
1253      }
1254      catch (Exception e) {
1255        if (stmt != null) {
1256          stmt.close();
1257        }
1258        logger.fatal(e.getMessage(), e);
1259        throw new SQLException(e.getMessage());
1260      }
```

```
1261    }
1262    /**
1263     * Creates a single Phone row in the T_PHONE table.  Uses the msgCategory,
1264     * msgObject, msgRelease and instId parameters passed in as well as the Phone
      ype associated
1265     * to the Phone object passed in as the data to create.
1266     * <P>
1267     * @param conn a pre-established database connection.  Since a Phone is only
      er
1268     * created as part of a larger transaction (BasicPerson-Create or Update) thi
1269     * insert must occur within that same transaction, therefore, this method doe
      not
1270     * commit the transaction.
1271     * @param msgCategory the 'messageCategory' attribute extracted from the
1272     * ControlAreaRequest of the enterprise message.  Example: 'com.any-erp-vendo
      Person'.
1273     * @param msgObject the 'messageObject' attribute extracted from the
1274     * ControlAreaRequest of the enterprise message.  Example: 'BasicPerson'.
1275     * @param msgRelease the 'messageRelease' attribute extracted from the
1276     * ControlAreaRequest of the enterprise message.  Example: '1.1'.
1277     * @param instId associated to the person we're creating an address for.
1278     * @param p the Phone Java object that was pulled from the BasicPerson object
      hat was
1279     * built from the enterprise message passed.
1280     * @throws SQLException if errors occur creating the Phone in the database.
1281     **/
1282    protected void createPhone(java.sql.Connection conn, String msgCategory, Str
      g msgObject, String msgRelease, String instId, Phone p) throws SQLException {
1283      PreparedStatement insertStmt = null;
1284      String insertString = "INSERT INTO T_PHONE (INST_ID, MESSAGE_CATEGORY, " +
1285                            "MESSAGE_OBJECT, MESSAGE_RELEASE, PHONE_TYPE, PHONE_
      EA, " +
1286                            "PHONE_NUMBER, " +
1287                            "CREATE_DATE, CREATE_USER, MOD_DATE, MOD_USER) " +
1288                            "VALUES (?,?,?,?,?,?,?,?,?,?,?)";
1289
1290      try {
1291        insertStmt = conn.prepareStatement(insertString);
1292        insertStmt.clearParameters();
1293        insertStmt.setString(1, instId);
1294        insertStmt.setString(2, msgCategory);
1295        insertStmt.setString(3, msgObject);
1296        insertStmt.setString(4, msgRelease);
1297        insertStmt.setString(5, p.getType(getAppName()));
1298        insertStmt.setString(6, p.getPhoneArea());
1299        insertStmt.setString(7, p.getPhoneNumber());
1300        insertStmt.setTimestamp(8, new java.sql.Timestamp(System.currentTimeMill
      ()));
1301        insertStmt.setString(9, conn.getMetaData().getUserName());
1302        insertStmt.setTimestamp(10, new java.sql.Timestamp(System.currentTimeMil
      s()));
1303        insertStmt.setString(11, conn.getMetaData().getUserName());
1304        logger.info("Attempting to insert Phone record " + p.toString() + " for
      stitutionalId: " + instId);
1305        int insertRc = insertStmt.executeUpdate();
1306        logger.info("Inserted Phone record " + p.toString() + " for Institutiona
      d: " + instId);
1307        insertStmt.close();
1308      }
1309      catch (Exception e) {
```

```
1310        if (insertStmt != null) {
1311          insertStmt.close();
1312        }
1313        logger.fatal(e.getMessage(), e);
1314        throw new SQLException(e.getMessage());
1315      }
1316    }
1317
1318    final private ArrayList logErrors(String errNumber, String errMessage, Throw
      le e, Document inDoc) {
1319      logger.fatal(errMessage, e);
1320      logger.fatal("Message sent in is: \n" + getMessageBody(inDoc));
1321      ArrayList errors = new ArrayList();
1322      errors.add(buildError("application", errNumber, errMessage));
1323      return errors;
1324    }
1325
1326    final private ArrayList logErrors(String errNumber, String errMessage, Docum
      t inDoc) {
1327      logger.fatal(errMessage);
1328      logger.fatal("Message sent in is: \n" + getMessageBody(inDoc));
1329      ArrayList errors = new ArrayList();
1330      errors.add(buildError("application", errNumber, errMessage));
1331      return errors;
1332    }
1333  }
```