

```

1  package ballmerpeak.turtlenet.server;
2
3  import ballmerpeak.turtlenet.shared.Message;
4  import ballmerpeak.turtlenet.shared.Conversation;
5  import java.security.*;
6  import java.sql.*;
7  import java.security.*;
8  import java.util.List;
9  import java.io.File;
10 import java.util.Vector;
11 import java.util.Arrays;
12
13 public class Database {
14     public static String path = "./db"; //path to database directory
15     private Connection dbConnection;
16     private String password = "UNSET";
17
18     public Database (String pw) {
19         password = pw;
20         dbConnection = null;
21         if (DBExists()) dbConnect(true); else dbCreate();
22     }
23
24     public static boolean DBDirExists() {
25         File dir = new File(path);
26         return dir.exists();
27     }
28
29     public static boolean DBExists() {
30         File edb = new File(path + "/turtlenet.db.aes");
31         File db = new File(path + "/turtlenet.db");
32         return db.exists() || edb.exists();
33     }
34
35     public static boolean createDBDir() {
36         return (new File(path)).mkdirs();
37     }
38
39     //Creates a database from scratch
40     public void dbCreate() {
41         Logger.write("INFO", "DB", "Creating database");
42         try {
43             if (!Database.DBDirExists())
44                 Database.createDBDir();
45             dbConnect(false);
46             for (int i = 0; i < DBStrings.createDB.length; i++)
47                 execute(DBStrings.createDB[i]);
48         } catch (Exception e) {
49             Logger.write("FATAL", "DB", "Failed to create database: " + e);
50         }
51     }
52
53     //Connects to a pre-defined database
54     public boolean dbConnect(boolean dbexists) {
55         if (dbexists)
56             if (!Crypto.decryptDB(password))
57                 Logger.write("FATAL", "DB", "failed to decrypt database");
58
59         Logger.write("INFO", "DB", "Connecting to database");
60         try {
61             Class.forName("org.sqlite.JDBC");
62             dbConnection = DriverManager.getConnection("jdbc:sqlite:db/turtlenet.db");
63             return true;
64         } catch (Exception e) { //Exception logged to disk, program allowed to crash naturally
65             Logger.write("FATAL", "DB", "Could not connect: " + e.getClass().getName() + ": " + e.getMessage());
66             return false;
67         }
68     }
69
70     //Disconnects the pre-defined database
71     public void dbDisconnect() {
72         Logger.write("INFO", "DB", "Disconnecting from database");
73         try {
74             dbConnection.close();
75         } catch (Exception e) { //Exception logged to disk, program allowed to continue
76             Logger.write("FATAL", "DB", "Could not disconnect: " + e.getClass().getName() + ": " + e.getMessage());
77         }
78
79         if (!Crypto.encryptDB(password))
80             Logger.write("FATAL", "DB", "failed to encrypt database");
81     }
82
83     public void execute (String query) throws java.sql.SQLException {
84         try {
85             /*
86             if (query.indexOf('(') != -1)
87                 Logger.write("VERBOSE", "DB", "execute(\"" + query.substring(0, query.indexOf('(')) + "...\"");
88             else
89                 Logger.write("VERBOSE", "DB", "execute(\"" + query.substring(0, 20) + "...\"");
90             */
91             Logger.write("VERBOSE", "DB", "execute(\"" + query + "\"");
92
93             Statement statement = dbConnection.createStatement();

```

```

94         statement.setQueryTimeout(30);
95         dbConnection.setAutoCommit(false);
96         statement.executeUpdate(query);
97         dbConnection.commit();
98         dbConnection.setAutoCommit(true);
99     } catch (java.sql.SQLException e) {
100         logger.write("ERROR", "DB", "SQLException: " + e);
101         throw e;
102     }
103 }
104
105 public ResultSet query (String query) throws java.sql.SQLException {
106     /*
107     if (query.indexOf('(') != -1)
108         logger.write("VERBOSE", "DB", "query(\"" + query.substring(0,query.indexOf('(')) + "...\"");
109     else
110         logger.write("VERBOSE", "DB", "query(\"" + query.substring(0,20) + "...\"");
111     */
112     logger.write("VERBOSE", "DB", "query(\"" + query + "\")");
113
114     try {
115         Statement statement = dbConnection.createStatement();
116         statement.setQueryTimeout(30);
117         ResultSet r = statement.executeQuery(query);
118         return r;
119     } catch (java.sql.SQLException e) {
120         logger.write("RED", "DB", "Failed to query database: " + e);
121         throw e;
122     }
123 }
124
125 //Get from DB
126 public String getPDATA(String field, PublicKey key) {
127     logger.write("VERBOSE", "DB", "getPDATA(" + field + ",...)");
128     String value = "";
129     try {
130         String strKey = Crypto.encodeKey(key);
131         String sqlStatement = DBStrings.getPDATA.replace("__FIELD__", field);
132         sqlStatement = sqlStatement.replace("__KEY__", strKey); //mods SQL template
133
134         ResultSet results = query(sqlStatement);
135         if(results.next())
136             value = results.getString(field); //gets current value in 'field'
137         else
138             value = "<No Value>";
139     } catch (java.sql.SQLException e) {
140         logger.write("ERROR", "DB", "SQLException: " + e);
141     }
142
143     if (value != null)
144         return value;
145     else
146         return "<no value>";
147 }
148
149 //Set the CMD to POST in the Message constructor
150 public Message[] getWallPost (PublicKey key) {
151     logger.write("VERBOSE", "DB", "getWallPost(...)");
152     Vector<Message> posts = new Vector<Message>();
153     try {
154         String sqlStatement = DBStrings.getWallPostSigs.replace("__KEY__", Crypto.encodeKey(key) );
155         ResultSet results = query(sqlStatement);
156
157         while (results.next()) {
158             Vector<String> visibleTo = new Vector<String>();
159             ResultSet currentPost = query(DBStrings.getPost.replace("__SIG__", results.getString("sig")));
160             ResultSet currentPostVisibleTo = query(DBStrings.getVisibleTo.replace("__SIG__", results.getString("sig")));
161             while(currentPostVisibleTo.next())
162                 visibleTo.add(currentPostVisibleTo.getString("key") );
163
164             if(currentPost.next()) {
165                 Message m = new MessageFactory().newPOST(currentPost.getString("msgText"), currentPost.getString
166 ("recieverKey"), (visibleTo.toArray(new String[0])) );
167                 m.timestamp = Long.parseLong(currentPost.getString("time"));
168                 m.signature = currentPost.getString("sig");
169                 m.command = "POST";
170                 posts.add(m);
171             }
172         } catch (java.sql.SQLException e) {
173             logger.write("ERROR", "DB", "SQLException: " + e);
174         }
175
176         return posts.toArray(new Message[0]);
177     }
178
179 public String getWallPostSender (String sig) {
180     logger.write("VERBOSE", "DB", "getWallPostSender(...)");
181     try {
182         ResultSet sendersKey = query(DBStrings.getPostSender.replace("__SIG__", sig));
183         if (sendersKey.next())
184             return sendersKey.getString("sendersKey");
185         else

```

```

186         return "<POST DOESN'T EXIST>";
187     } catch (java.sql.SQLException e) {
188         Logger.write("ERROR", "DB", "SQLException: " + e);
189         return "ERROR";
190     }
191 }
192
193 public Message[] getComments (String sig) {
194     Vector<Message> comments = new Vector<Message>();
195     Logger.write("VERBOSE", "DB", "getComments(...)");
196
197     try {
198         ResultSet commentSet = query(DBStrings.getComments.replace("__PARENT__", sig));
199         while (commentSet.next()) {
200             Message cmnt = new MessageFactory().newCMNT(sig, commentSet.getString("msgText"));
201             cmnt.timestamp = Long.parseLong(commentSet.getString("creationTime"));
202             cmnt.signature = commentSet.getString("sig");
203             comments.add(cmnt);
204         }
205     } catch (java.sql.SQLException e) {
206         Logger.write("ERROR", "DB", "SQLException: " + e);
207     }
208
209     return comments.toArray(new Message[0]);
210 }
211
212 public Long timeMostRecentWallPost (PublicKey key) {
213     Logger.write("VERBOSE", "DB", "timeMostRecentWallPost(...)");
214     try {
215         ResultSet mostRecent = query(DBStrings.mostRecentWallPost.replace("__KEY__", Crypto.encodeKey(key)));
216         if (mostRecent.next())
217             return Long.parseLong(mostRecent.getString("maxtime"));
218     } catch (java.sql.SQLException e) {
219         Logger.write("ERROR", "DB", "SQLException: " + e);
220     }
221     return 0L;
222 }
223
224 public boolean isLiked (String sig) {
225     Logger.write("VERBOSE", "DB", "isLiked(...)");
226     int ret = 0;
227
228     try {
229         ResultSet row = query(DBStrings.getLike.replace("__SIG__", sig));
230         return row.next();
231     } catch (java.sql.SQLException e) {
232         Logger.write("ERROR", "DB", "SQLException: " + e);
233     }
234
235     return false;
236 }
237
238 //Return all conversations
239 public Conversation[] getConversations () {
240     Vector<Conversation> convoList = new Vector<Conversation>();
241     Logger.write("VERBOSE", "DB", "getConversations()");
242
243     try {
244         ResultSet convoSet = query(DBStrings.getConversations);
245         while (convoSet.next())
246             convoList.add(getConversation(convoSet.getString("convoID")));
247     } catch (java.sql.SQLException e) {
248         Logger.write("ERROR", "DB", "SQLException: " + e);
249     }
250
251     return convoList.toArray(new Conversation[0]);
252 }
253
254 //Get keys of all people in the given conversation
255 public PublicKey[] getPeopleInConvo (String sig) {
256     Logger.write("VERBOSE", "DB", "getPeopleInConvo(...)");
257     Vector<PublicKey> keys = new Vector<PublicKey>();
258
259     try {
260         ResultSet keySet = query(DBStrings.getConversationMembers.replace("__SIG__", sig));
261         while (keySet.next())
262             keys.add(Crypto.decodeKey(keySet.getString("key")));
263     } catch (java.sql.SQLException e) {
264         Logger.write("ERROR", "DB", "SQLException: " + e);
265     }
266
267     return keys.toArray(new PublicKey[0]);
268 }
269
270 //Return a conversation object
271 public Conversation getConversation (String sig) {
272     Logger.write("VERBOSE", "DB", "getConversation(...)");
273     try {
274         ResultSet convoSet = query(DBStrings.getConversation.replace("__SIG__", sig));
275         if (convoSet.next()) {
276             String timestamp = convoSet.getString("time");
277             ResultSet messages = query(DBStrings.getConversationMessages.replace("__SIG__", sig));
278             String firstMsg;

```

```

279         if (messages.next())
280             firstMsg = messages.getString("msgText");
281         else
282             firstMsg = "<no messages yet>";
283         PublicKey[] keys = getPeopleInConvo(sig);
284         String[] keystings = new String[keys.length];
285         String[] users = new String[keys.length];
286         for (int i = 0; i < keys.length; i++) {
287             keystings[i] = Crypto.encodeKey(keys[i]);
288             users[i] = getName(keys[i]);
289         }
290         return new Conversation(sig, timestamp, firstMsg, users, keystings);
291     } else {
292         Logger.write("WARNING", "DB", "getConversation(...) empty conversation: " + sig);
293     }
294 } catch (java.sql.SQLException e) {
295     Logger.write("ERROR", "DB", "SQLException: " + e);
296 }
297 return new Conversation();
298 }
299
300 //Return all messages in a conversation
301 //{{username, time, msg}, {username, time, msg}, etc.}
302 //Please order it so that element 0 is the oldest message
303 public String[][] getConversationMessages (String sig) {
304     Logger.write("VERBOSE", "DB", "getConversationMessages(...)");
305     Vector<String[]> messagesList = new Vector<String[]>();
306
307     try {
308         ResultSet messageSet = query(DBStrings.getConversationMessages.replace("__SIG__", sig));
309         while(messageSet.next() ) {
310             String[] message = new String[3];
311             message[0] = getName(Crypto.decodeKey(messageSet.getString("sendersKey")));
312             message[1] = messageSet.getString("time");
313             message[2] = messageSet.getString("msgText");
314
315             messagesList.add(message);
316         }
317     } catch (java.sql.SQLException e) {
318         Logger.write("ERROR", "DB", "SQLException: " + e);
319     }
320
321     return messagesList.toArray(new String[0][0]);
322 }
323
324 //If multiple people have the same username then:
325 //Logger.write("FATAL", "DB", "Duplicate usernames");
326 //System.exit(1);
327 public PublicKey getKey (String userName) {
328     Logger.write("VERBOSE", "DB", "getKey(" + userName + ")");
329     int nameCount = 0;
330     String key = "<No Key>";
331
332     try {
333         ResultSet results = query(DBStrings.getKey.replace("__USERNAME__", userName) );
334         while(results.next()) {
335             nameCount++;
336             key = results.getString("key");
337         }
338     } catch (java.sql.SQLException e) {
339         Logger.write("ERROR", "DB", "SQLException: " + e);
340     }
341
342     if(nameCount == 0)
343         Logger.write("ERROR", "DB", "getKey(" + userName + ") - No keys found for userName");
344     else if (nameCount > 1 )
345         Logger.write("ERROR", "DB", "getKey(" + userName + ") - Multiple userNames found for key; Server OPs are evil!");
346
347     return Crypto.decodeKey(key);
348 }
349
350 public boolean canSeePDATA (String category) {
351     Logger.write("VERBOSE", "DB", "canSeePDATA()");
352
353     try {
354         ResultSet categorySet = query(DBStrings.canSeePDATA.replace("__CATID__", category));
355         if (categorySet.next()) {
356             return categorySet.getInt("canSeePDATA") == 1 ? true : false;
357         }
358     } catch (java.sql.SQLException e) {
359         Logger.write("ERROR", "DB", "SQLException: " + e);
360     }
361
362     return false;
363 }
364
365 //Return the name of each member and if it can see your profile info
366 //In this format: {{ "friends", "false"}, {"family", "true"}, etc.}
367 public String[][] getCategories () {
368     Logger.write("VERBOSE", "DB", "getCategories()");
369     Vector<String[]> catList = new Vector<String[]>();
370     String catName;
371     String canSeePDATA;

```

```

372
373     try {
374         ResultSet categorySet = query(DBStrings.getCategories);
375         while(categorySet.next() ) {
376             String[] category = new String[2];
377             category[0] = categorySet.getString("catID");
378             category[1] = categorySet.getInt("canSeePDATA") == 1 ? "true" : "false";
379             catList.add(category);
380         }
381     } catch (java.sql.SQLException e) {
382         Logger.write("ERROR", "DB", "SQLException: " + e);
383     }
384
385     Logger.write("VERBOSE", "DB", "getCategories() returning " + catList.toArray().length + " categories");
386     return catList.toArray(new String[0][0]);
387 }
388
389 //Return the keys of each member of the category
390 //if(category.equals("all")) //remember NEVER to compare strings with ==
391 //    return every key you know about
392 public PublicKey[] getCategoryMembers (String catID) {
393     Logger.write("VERBOSE", "DB", "getCategoryMembers(" + catID + ")");
394     String queryStr = "";
395
396     if(catID.toLowerCase().equals("all"))
397         queryStr = DBStrings.getAllKeys;
398     else
399         queryStr = DBStrings.getMemberKeys.replace("__CATNAME__", catID);
400
401     Vector<PublicKey> keyList = new Vector<PublicKey>();
402
403     try {
404         ResultSet keySet = query(queryStr);
405         while(keySet.next()) {
406             if(catID.toLowerCase().equals("all"))
407                 keyList.add(Crypto.decodeKey(keySet.getString("key")));
408             else
409                 keyList.add(Crypto.decodeKey(keySet.getString("userKey")));
410         }
411     } catch (java.sql.SQLException e) {
412         Logger.write("ERROR", "DB", "SQLException: " + e);
413     }
414
415     Logger.write("VERBOSE", "DB", "getCategoryMembers(" + catID + ") returning " + keyList.toArray().length + " members");
416     return keyList.toArray(new PublicKey[0]);
417 }
418
419 //Given the sig of a post or comment return the keys which can see it
420 public PublicKey[] getVisibilityOfParent(String sig) {
421     Logger.write("VERBOSE", "DB", "getVisibilityOfParent(" + sig + ")");
422
423     try {
424         ResultSet postWithSig = query(DBStrings.getPost.replace("__SIG__", sig));
425         if (postWithSig.next()) { //sig is a post
426             Logger.write("VERBOSE", "DB", "parent is a wall post: " + sig);
427             return getPostVisibleTo(sig);
428         } else { //sig is a comment
429             ResultSet commentWithSig = query(DBStrings.getComment.replace("__SIG__", sig));
430             if (commentWithSig.next())
431                 return getVisibilityOfParent(commentWithSig.getString("parent"));
432             else
433                 Logger.write("ERROR", "DB", "getVisibilityOfParent has no root");
434         }
435     } catch (java.sql.SQLException e) {
436         Logger.write("ERROR", "DB", "SQLException: " + e);
437     }
438
439     return null;
440 }
441
442 public PublicKey[] getPostVisibleTo (String sig) {
443     Logger.write("VERBOSE", "DB", "getPostVisibleTo(...)");
444     Vector<PublicKey> keyList = new Vector<PublicKey>();
445
446     try {
447         ResultSet keyRows = query(DBStrings.getVisibleTo.replace("__SIG__", sig));
448         while(keyRows.next())
449             keyList.add(Crypto.decodeKey(keyRows.getString("key")));
450     } catch (java.sql.SQLException e) {
451         Logger.write("ERROR", "DB", "SQLException: " + e);
452     }
453
454     return keyList.toArray(new PublicKey[0]);
455 }
456
457 //In the case of no username for the key: "return Crypto.encode(k);"
458 public String getName (PublicKey key) {
459     Logger.write("VERBOSE", "DB", "getName(...)");
460     String name = "";
461
462     try {
463         ResultSet nameRow = query(DBStrings.getName.replace("__KEY__", Crypto.encodeKey(key)));
464         if (nameRow.next())

```

```

465         name = nameRow.getString("username");
466     } catch (java.sql.SQLException e) {
467         Logger.write("ERROR", "DB", "SQLException: " + e);
468     }
469
470     if (name != null)
471         return name;
472     else
473         return "<no username>";
474 }
475
476 //What key signed this message
477 public PublicKey getSignatory (Message m) {
478     Logger.write("VERBOSE", "DB", "getSignatory(...)");
479     try {
480         ResultSet keys = query(DBStrings.getAllKeys);
481         while (keys.next())
482             if (Crypto.verifySig(m, Crypto.decodeKey(keys.getString("key"))))
483                 return Crypto.decodeKey(keys.getString("key"));
484     } catch (java.sql.SQLException e) {
485         Logger.write("ERROR", "DB", "SQLException: " + e);
486     }
487     Logger.write("WARNING", "DB", "getSignatory() could not find signatory");
488     return null;
489 }
490
491 //Add to DB
492 public boolean addPost (Message post) {
493     Logger.write("VERBOSE", "DB", "addPost(...)");
494
495     try {
496         execute(DBStrings.addPost.replace("__SIG__", post.getSig())
497             .replace("__msgText__", post.POSTgetText())
498             .replace("__time__", Long.toString(post.getTimestamp()))
499             .replace("__recieverKey__", post.POSTgetWall())
500             .replace("__sendersKey__", Crypto.encodeKey(getSignatory(post))));
501         String[] visibleTo = post.POSTgetVisibleTo();
502         for (int i = 0; i < visibleTo.length; i++)
503             execute(DBStrings.addPostVisibility.replace("__postSig__", post.getSig()).replace("__key__", visibleTo[i]));
504         return true;
505     } catch (java.sql.SQLException e) {
506         Logger.write("ERROR", "DB", "SQLException: " + e);
507         return false;
508     }
509 }
510
511 public boolean addKey (Message msg) {
512     return addKey(Crypto.decodeKey(msg.ADDKEYgetKey()));
513 }
514
515 public boolean addKey (PublicKey k) {
516     Logger.write("VERBOSE", "DB", "addKey(...)");
517
518     try {
519         execute(DBStrings.addKey.replace("__key__", Crypto.encodeKey(k)));
520         boolean ret = validateClaims(k);
521         if (!calcRevocationKeys(k))
522             ret = false;
523         return ret;
524     } catch (java.sql.SQLException e) {
525         Logger.write("ERROR", "DB", "SQLException: " + e);
526     }
527
528     return false;
529 }
530
531 //Update k's username by validating claims
532 public boolean validateClaims(PublicKey k) {
533     if (k == null) {
534         Logger.write("ERROR", "DB", "validateClaims(...) called with null key");
535         return false;
536     }
537
538     Logger.write("VERBOSE", "DB", "validateClaims(...)");
539
540     try {
541         ResultSet claimSet = query(DBStrings.getClaims);
542         while (claimSet.next()) {
543             Message msg = new Message("CLAIM",
544                 claimSet.getString("name"),
545                 Long.parseLong(claimSet.getString("claimTime")),
546                 claimSet.getString("sig"));
547
548             Logger.write("VERBOSE", "DB", "Considering Claim for name: \"\" + claimSet.getString("name") + "\"\"");
549             Logger.write("VERBOSE", "DB", "time: \"\" + Long.toString(Long.parseLong
550 (claimSet.getString("claimTime"))) + "\"\"");
551             Logger.write("VERBOSE", "DB", "sig: \"\" + claimSet.getString("sig") + "\"\"");
552
553             PublicKey signatory = getSignatory(msg);
554             if (signatory != null && signatory.equals(k)) {
555                 execute(DBStrings.newUsername.replace("__name__", msg.CLAIMgetName()).replace("__key__", Crypto.encodeKey
556 (k)));
557                 execute(DBStrings.removeClaim.replace("__sig__", msg.getSig()));

```



```

556         Logger.write("INFO", "DB", "Claim for " + msg.CLAIMgetName() + " verified");
557     }
558 }
559 } catch (java.sql.SQLException e) {
560     Logger.write("ERROR", "DB", "SQLException: " + e);
561     return false;
562 }
563 return true;
564 }
565
566 //update keys column in revocations
567 public boolean calcRevocationKeys (PublicKey k) {
568     if (k == null) {
569         Logger.write("ERROR", "DB", "calcRevocationKeys(...) called with null key");
570         return false;
571     }
572
573     Logger.write("VERBOSE", "DB", "calcRevocationKeys(...)");
574
575     try {
576         ResultSet revocationSet = query(DBStrings.getRevocations());
577         while (revocationSet.next()) {
578             Message msg = new Message("REVOKE",
579                                     revocationSet.getString("timeOfLeak"),
580                                     Long.parseLong(revocationSet.getString("creationTime")),
581                                     revocationSet.getString("sig"));
582             PublicKey signer = getSignatory(msg);
583             if (signer != null && signer.equals(k)) {
584                 execute(DBStrings.updateRevocationKey.replace("__KEY__", Crypto.encodeKey(k))
585                     .replace("__SIG__", revocationSet.getString("sig")));
586             }
587         }
588     } catch (java.sql.SQLException e) {
589         Logger.write("ERROR", "DB", "SQLException: " + e);
590         return false;
591     }
592     return true;
593 }
594
595 //if this key has already claimed a name, forget the old one
596 public boolean addClaim (Message claim) {
597     Logger.write("VERBOSE", "DB", "addClaim(" + claim.CLAIMgetName() + ")");
598
599     try {
600         execute(DBStrings.addClaim.replace("__sig__", claim.getSig())
601             .replace("__name__", claim.CLAIMgetName())
602             .replace("__time__", Long.toString(claim.getTimestamp())));
603
604         ResultSet everyone = query(DBStrings.getAllKeys());
605         while (everyone.next())
606             validateClaims(Crypto.decodeKey(everyone.getString("key")));
607     } catch (java.sql.SQLException e) {
608         Logger.write("ERROR", "DB", "SQLException: " + e);
609         return false;
610     }
611     return true;
612 }
613
614 public boolean addRevocation (Message revocation) {
615     Logger.write("VERBOSE", "DB", "-----addRevocation(...)-----");
616
617     try {
618         execute(DBStrings.addRevocation.replace("__key__", Crypto.encodeKey(getSignatory(revocation)))
619             .replace("__sig__", revocation.getSig())
620             .replace("__time__", Long.toString(revocation.REVOKEgetTime()))
621             .replace("__creationTime__", Long.toString(revocation.getTimestamp())));
622         return eraseContentFrom(getSignatory(revocation));
623     } catch (java.sql.SQLException e) {
624         Logger.write("ERROR", "DB", "SQLException: " + e);
625         return false;
626     }
627 }
628
629 public boolean isRevoked (PublicKey key) {
630     Logger.write("VERBOSE", "DB", "isRevoked(...)");
631
632     try {
633         return query(DBStrings.isRevoked.replace("__KEY__", Crypto.encodeKey(key))).next();
634     } catch (java.sql.SQLException e) {
635         Logger.write("ERROR", "DB", "SQLException: " + e);
636         return false;
637     }
638 }
639
640 public boolean eraseContentFrom (PublicKey key) {
641     Logger.write("VERBOSE", "DB", "-----eraseContentFrom(...)-----");
642     String keyStr = Crypto.encodeKey(key);
643
644     try {
645         execute(DBStrings.removeMessageAccess.replace("__KEY__", keyStr));
646         execute(DBStrings.removeMessages.replace("__KEY__", keyStr));
647         execute(DBStrings.removePosts.replace("__KEY__", keyStr));
648         execute(DBStrings.removePostVisibility.replace("__KEY__", keyStr));

```

```

649         execute(DBStrings.removeUser.replace("__KEY__", keyStr));
650         execute(DBStrings.removeFromCategories.replace("__KEY__", keyStr));
651         execute(DBStrings.removeLikes.replace("__KEY__", keyStr));
652         execute(DBStrings.removeComments.replace("__KEY__", keyStr));
653         execute(DBStrings.removeEvents.replace("__KEY__", keyStr));
654     } catch (java.sql.SQLException e) {
655         Logger.write("ERROR", "DB", "SQLException: " + e);
656         return false;
657     }
658
659     return true;
660 }
661
662 public boolean addPDATA (Message update) {
663     Logger.write("VERBOSE", "DB", "addPDATA(...)");
664     boolean ret = true;
665
666     String[][] updates = update.PDATAgetValues();
667     for (int i = 0; i < updates.length; i++)
668         if (!updatePDATA(updates[i][0], updates[i][1], getSignatory(update)))
669             ret = false;
670
671     return ret;
672 }
673
674 public boolean updatePDATA (String field, String value, PublicKey k) {
675     Logger.write("VERBOSE", "DB", "updatePDATA(" + field + ", " + value + ", ...)");
676
677     try {
678         execute(DBStrings.addPDATA.replace("__field__", field)
679             .replace("__value__", value)
680             .replace("__key__", Crypto.encodeKey(k)));
681     } catch (java.sql.SQLException e) {
682         Logger.write("ERROR", "DB", "SQLException: " + e);
683         return false;
684     }
685
686     return true;
687 }
688
689 public boolean addConvo (Message convo) {
690     Logger.write("VERBOSE", "DB", "addConvo(...)");
691
692     try {
693         execute(DBStrings.addConvo.replace("__sig__", convo.getSig())
694             .replace("__time__", Long.toString(convo.getTimestamp())));
695         String[] keys = convo.CHATgetKeys();
696         for (int i = 0; i < keys.length; i++) {
697             execute(DBStrings.addConvoParticipant.replace("__sig__", convo.getSig())
698                 .replace("__key__", keys[i]));
699         }
700     } catch (java.sql.SQLException e) {
701         Logger.write("ERROR", "DB", "SQLException: " + e);
702         return false;
703     }
704
705     return true;
706 }
707
708 public boolean addMessageToChat (Message msg) {
709     Logger.write("VERBOSE", "DB", "addMessageToChat(...)");
710
711     try {
712         boolean duplicate = false;
713
714         String[][] messagesInConvo = getConversationMessages(msg.PCHATgetConversationID());
715         for (int i = 0; i < messagesInConvo.length; i++)
716             if (messagesInConvo[i][1].equals(Long.toString(msg.getTimestamp())) && messagesInConvo[i][2].equals
(msg.PCHATgetText()))
717                 duplicate = true;
718
719         if (!duplicate) {
720             execute(DBStrings.addMessageToConvo.replace("__convoID__", msg.PCHATgetConversationID())
721                 .replace("__sendersKey__", Crypto.encodeKey(getSignatory(msg)))
722                 .replace("__msgText__", msg.PCHATgetText())
723                 .replace("__time__", Long.toString(msg.getTimestamp())));
724         }
725     } catch (java.sql.SQLException e) {
726         Logger.write("ERROR", "DB", "SQLException: " + e);
727         return false;
728     }
729
730     return true;
731 }
732
733 public boolean addComment (Message comment) {
734     Logger.write("VERBOSE", "DB", "addComment(...)");
735
736     try {
737         execute(DBStrings.addComment.replace("__sig__", comment.getSig())
738             .replace("__msgText__", comment.CMNTgetText())
739             .replace("__parent__", comment.CMNTgetItemID())
740             .replace("__commenterKey__", Crypto.encodeKey(getSignatory(comment))))

```



```

741         .replace("__senderKey__", Crypto.encodeKey(getSignatory(comment)))
742         .replace("__creationTime__", Long.toString(comment.getTimestamp())));
743     } catch (java.sql.SQLException e) {
744         Logger.write("ERROR", "DB", "SQLException: " + e);
745         return false;
746     }
747
748     return true;
749 }
750
751 public boolean addLike (Message like) {
752     Logger.write("VERBOSE", "DB", "addLike(...)");
753
754     try {
755         execute(DBStrings.addLike.replace("__likerKey__", Crypto.encodeKey(getSignatory(like)))
756             .replace("__parent__", like.LIKEgetItemID()));
757     } catch (java.sql.SQLException e) {
758         Logger.write("ERROR", "DB", "SQLException: " + e);
759         return false;
760     }
761
762     return true;
763 }
764
765 public boolean addEvent (Message event) {
766     Logger.write("VERBOSE", "DB", "addEvent(...)");
767     try {
768         execute(DBStrings.addEvent.replace("__sig__", event.getSig())
769             .replace("__startTime__", Long.toString(event.EVNTgetStart()))
770             .replace("__endTime__", Long.toString(event.EVNTgetEnd()))
771             .replace("__creatorKey__", Crypto.encodeKey(getSignatory(event)))
772             .replace("__accepted__", "0")
773             .replace("__name__", event.EVNTgetName())
774             .replace("__creationTime__", Long.toString(event.getTimestamp())));
775     } catch (java.sql.SQLException e) {
776         Logger.write("ERROR", "DB", "SQLException: " + e);
777         return false;
778     }
779
780     return true;
781 }
782
783 public boolean acceptEvent (String sig) {
784     Logger.write("VERBOSE", "DB", "acceptEvent(...)");
785     try {
786         execute(DBStrings.acceptEvent.replace("__sig__", sig));
787     } catch (java.sql.SQLException e) {
788         Logger.write("ERROR", "DB", "SQLException: " + e);
789         return false;
790     }
791
792     return true;
793 }
794
795 public boolean declineEvent (String sig) {
796     Logger.write("VERBOSE", "DB", "declineEvent(...)");
797     try {
798         execute(DBStrings.declineEvent.replace("__sig__", sig));
799     } catch (java.sql.SQLException e) {
800         Logger.write("ERROR", "DB", "SQLException: " + e);
801         return false;
802     }
803
804     return true;
805 }
806
807 public boolean updatePDATApersion (Message msg) {
808     return updatePDATApersion(msg.UPDATECATgetName(), msg.UPDATECATgetValue());
809 }
810
811 public boolean updatePDATApersion (String category, boolean value) {
812     Logger.write("VERBOSE", "DB", "updatePDATApersion(...)");
813     try {
814         execute(DBStrings.updatePDATApersion.replace("__catID__", category)
815             .replace("__bool__", value?"1":"0"));
816     } catch (java.sql.SQLException e) {
817         Logger.write("ERROR", "DB", "SQLException: " + e);
818         return false;
819     }
820
821     return true;
822 }
823
824 public PublicKey[] keysCanSeePDATA () {
825     Logger.write("VERBOSE", "DB", "keysCanSeePDATA()");
826     Vector<PublicKey> keys = new Vector<PublicKey>();
827
828     try {
829         ResultSet categories = query(DBStrings.categoriesCanSeePDATA);
830         while (categories.next()) {
831             String catname = categories.getString("catID");
832             PublicKey[] memberKeys = getCategoryMembers(catname);
833             for (int i = 0; i < memberKeys.length; i++)

```

```

834         if (!keys.contains(memberKeys[i]))
835             keys.add(memberKeys[i]);
836     }
837 } catch (java.sql.SQLException e) {
838     Logger.write("ERROR", "DB", "SQLException: " + e);
839 }
840
841 return keys.toArray(new PublicKey[0]);
842 }
843
844 //no duplicate names
845 public boolean addCategory (Message msg) {
846     return addCategory(msg.ADDCATgetName(), msg.ADDCATgetValue());
847 }
848
849 public boolean addCategory (String name, boolean can_see_private_details) {
850     Logger.write("VERBOSE", "DB", "addCategory(...)");
851     try {
852         execute(DBStrings.addCategory.replace("__catID__", name)
853             .replace("__canSeePDATA__", can_see_private_details?"1":"0"));
854     } catch (java.sql.SQLException e) {
855         Logger.write("ERROR", "DB", "SQLException: " + e);
856         return false;
857     }
858
859     return true;
860 }
861
862 public boolean addToCategory (Message msg) {
863     return addToCategory(msg.ADDTOCATgetName(), Crypto.decodeKey(msg.ADDTOCATgetKey()));
864 }
865
866 public boolean addToCategory (String category, PublicKey key) {
867     Logger.write("VERBOSE", "DB", "addToCategory(" + category + ", ...)");
868
869     PublicKey[] members = getCategoryMembers(category);
870     if (Arrays.asList(members).contains(key)) {
871         return false;
872     }
873
874     try {
875         execute(DBStrings.addToCategory.replace("__catID__", category)
876             .replace("__key__", Crypto.encodeKey(key)));
877     } catch (java.sql.SQLException e) {
878         Logger.write("ERROR", "DB", "SQLException: " + e);
879         return false;
880     }
881
882     return true;
883 }
884
885 public boolean removeFromCategory (Message msg) {
886     return removeFromCategory(msg.REMFROMCATgetCategory(), Crypto.decodeKey(msg.REMFROMCATgetKey()));
887 }
888
889 public boolean removeFromCategory (String category, PublicKey key) {
890     Logger.write("VERBOSE", "DB", "removeFromCategory(" + category + ", ...)");
891     try {
892         execute(DBStrings.removeFromCategory.replace("__catID__", category)
893             .replace("__key__", Crypto.encodeKey(key)));
894     } catch (java.sql.SQLException e) {
895         Logger.write("ERROR", "DB", "SQLException: " + e);
896         return false;
897     }
898
899     return true;
900 }
901
902 public boolean like (String sig) {
903     Logger.write("VERBOSE", "DB", "like(...)");
904     try {
905         execute(DBStrings.addLike.replace("__parent__", sig)
906             .replace("__likerKey__", Crypto.encodeKey(Crypto.getPublicKey())));
907     } catch (java.sql.SQLException e) {
908         Logger.write("ERROR", "DB", "SQLException: " + e);
909         return false;
910     }
911
912     return true;
913 }
914
915 public boolean unlike (String sig) {
916     Logger.write("VERBOSE", "DB", "like(...)");
917     try {
918         execute(DBStrings.removeLike.replace("__parent__", sig)
919             .replace("__likerKey__", Crypto.encodeKey(Crypto.getPublicKey())));
920     } catch (java.sql.SQLException e) {
921         Logger.write("ERROR", "DB", "SQLException: " + e);
922         return false;
923     }
924
925     return true;
926 }

```

---

927     }