```java
//All methods ought to be static
package ballmerpeak.turtlenet.server;

import ballmerpeak.turtlenet.server.FIO;
import ballmerpeak.turtlenet.shared.Message;
import java.io.*;
import java.security.*;
import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
import java.security.spec.X509EncodedKeySpec;
import javax.crypto.spec.SecretKeySpec;
import javax.crypto.spec.IvParameterSpec;
import javax.xml.bind.DatatypeConverter;
import java.util.StringTokenizer;
import java.security.SecureRandom;

public class Crypto {
    public static SecureRandom srand = new SecureRandom(
                                               Long.toString(
                                                   System.currentTimeMillis())
                                               .getBytes());

    public static Boolean keysExist() {
        File publicKey  = new File(Database.path + "/public.key");
        File privateKey = new File(Database.path + "/private.key");
        return publicKey.exists() && privateKey.exists();
    }

    public static void keyGen() {
        try {
            Logger.write("INFO", "Crypto","Generating keys");

            //generate the key
            KeyPairGenerator gen = KeyPairGenerator.getInstance("RSA");
            gen.initialize(1024, srand);
            KeyPair keys = gen.generateKeyPair();

            //create the DB directory if needed
            if (!Database.DBDirExists())
                Database.createDBDir();

            //and save the keys into it
            ObjectOutputStream publicKeyFile = new ObjectOutputStream(
                                                   new FileOutputStream(
                                                       new File("./db/public.key")));
            publicKeyFile.writeObject(keys.getPublic());
            publicKeyFile.close();

            ObjectOutputStream privateKeyFile = new ObjectOutputStream(
                                                    new FileOutputStream(
                                                        new File("./db/private.key")));
            privateKeyFile.writeObject(keys.getPrivate());
            privateKeyFile.close();
        } catch (Exception e) {
            Logger.write("ERROR", "Crypto", "Could not generate keypair");
        }
    }

    //encrypt all files in db folder, rename to <filename>.aes
    public static boolean encryptDB(String password) {
        Logger.write("VERBOSE", "Crypto", "encryptDB(" + password + ")");
        try {
            String salt = Long.toString(System.currentTimeMillis());
            password += salt;
            FIO.writeFileBytes(salt.getBytes("UTF-8"), Database.path + "/salt");
            FIO.writeFileBytes(encryptBytes(FIO.readFileBytes(Database.path + "/turtlenet.db"), password+"db"), Database.path + "/turtlenet.db.aes");
            FIO.writeFileBytes(encryptBytes(FIO.readFileBytes(Database.path + "/public.key"), password+"pu"), Database.path + "/public.key.aes");
            FIO.writeFileBytes(encryptBytes(FIO.readFileBytes(Database.path + "/private.key"), password+"pr"), Database.path + "/private.key.aes");
            FIO.writeFileBytes(encryptBytes(FIO.readFileBytes(Database.path + "/lastread"), password+"lr"), Database.path + "/lastread.aes");
            new File(Database.path + "/turtlenet.db").delete();
            new File(Database.path + "/public.key").delete();
            new File(Database.path + "/private.key").delete();
            new File(Database.path + "/lastread").delete();
        } catch (Exception e) {
            Logger.write("FATAL", "Crypto", "Unable to encrypt files: " + e);
            return false;
        }
        return true;
    }

    //decrypt all files <filename>.aes in db folder, rename to <filename>
    public static boolean decryptDB(String password) {
        Logger.write("VERBOSE", "Crypto", "decryptDB(" + password + ")");
        try {
            password += new String(FIO.readFileBytes(Database.path + "/salt"));
            FIO.writeFileBytes(decryptBytes(FIO.readFileBytes(Database.path + "/turtlenet.db.aes"), password+"db"), Database.path + "/turtlenet.db");
            FIO.writeFileBytes(decryptBytes(FIO.readFileBytes(Database.path + "/public.key.aes"), password+"pu"),
Database.path + "/public.key");
            FIO.writeFileBytes(decryptBytes(FIO.readFileBytes(Database.path + "/private.key.aes"), password+"pr"),
Database.path + "/private.key");
            FIO.writeFileBytes(decryptBytes(FIO.readFileBytes(Database.path + "/lastread.aes"), password+"lr"), Database.path
+ "/lastread");
            new File(Database.path + "/turtlenet.db.aes").delete();
            new File(Database.path + "/public.key.aes").delete();
            new File(Database.path + "/private.key.aes").delete();
            new File(Database.path + "/lastread.aes").delete();
            new File(Database.path + "/salt").delete();
        } catch (Exception e) {
            Logger.write("FATAL", "Crypto", "Unable to decrypt files: " + e);
            return false;
        }
        return false;
    }

    public static KeyPair getTestKey() {
        Logger.write("INFO", "Crypto","Generating test keypair");
        try {
            KeyPairGenerator gen = KeyPairGenerator.getInstance("RSA");
            gen.initialize(1024, srand);
            return gen.generateKeyPair();
        } catch (Exception e) {
            Logger.write("ERROR", "Crypto", "Couldn't generate test keypair: " + e);
            return null;
        }
    }

    public static PublicKey getPublicKey() {
        try {
            ObjectInputStream file = new ObjectInputStream(
                                     new FileInputStream(
                                     new File("./db/public.key")));
            return (PublicKey) file.readObject();
        } catch (Exception e) {
            Logger.write("WARNING", "Crypto", "Could not read public key");
        }
        return null;
    }

    public static PrivateKey getPrivateKey() {
        try {
            ObjectInputStream file = new ObjectInputStream(
                                     new FileInputStream(
                                     new File("./db/private.key")));
            return (PrivateKey) file.readObject();
        } catch (Exception e) {
            Logger.write("WARNING", "Crypto", "Could not read private key");
        }
        return null;
    }

    public static String sign (Message msg) {
        Logger.write("INFO", "Crypto","sign()");
        return sign(msg, Crypto.getPrivateKey());
    }

    public static String sign (Message msg, PrivateKey k) {
        Logger.write("INFO", "Crypto","sign()");
        try {
            Signature signer = Signature.getInstance("SHA1withRSA");
            signer.initSign(k);
            signer.update((Long.toString(msg.timestamp) + msg.content).getBytes("UTF-8"));
            byte[] sig = signer.sign();
            return Crypto.Base64Encode(sig);
        } catch (Exception e) {
            Logger.write("ERROR", "Crypto", "Could not sign message");
        }
        return "";
    }

    public static String hash (String data) {
        try {
            MessageDigest hasher = MessageDigest.getInstance("SHA-256");
            return DatatypeConverter.printHexBinary(hasher.digest(data.getBytes("UTF-8")));
        } catch (Exception e) {
            Logger.write("FATAL", "DB","SHA-256 not supported by your JRE");
        }
        return "not_a_hash";
    }

    public static boolean verifySig (Message msg, PublicKey author) {
        Logger.write("INFO", "Crypto","verifySig()");
        try {
            Signature sigChecker = Signature.getInstance("SHA1withRSA");
            sigChecker.initVerify(author);
            sigChecker.update((Long.toString(msg.getTimestamp())+msg.getContent()).getBytes("UTF-8"));
            boolean valid = sigChecker.verify(Crypto.Base64Decode(msg.getSig()));
            if (valid) {
                Logger.write("INFO", "Crypto","verifySig() - TRUE");
            } else {
                Logger.write("INFO", "Crypto","verifySig() - FALSE");
            }
            return valid;
        } catch (Exception e) {
            Logger.write("ERROR", "Crypto", "Could not verify signature");
        }
        return false;
    }

    //Time differentials can, and have, been used to corrolate otherwise
    //  anonymous messages; therefore server time is used. This is not to
    //  protect against malicious server operators, but operators ordered after
    //  the fact to provide the data they've collected.
    //The NetworkConnection is used to get the servers time.
    public static String encrypt(Message msg, PublicKey recipient, NetworkConnection connection) {
        try {
            Logger.write("INFO", "Crypto","encrypt()");
            //encrypt with random AES key
            byte[]     iv = new byte[16];
            byte[] aeskey = new byte[16];
            srand.nextBytes(iv); //fills the array with random data
            srand.nextBytes(aeskey);

            SecretKeySpec aesKeySpec = new SecretKeySpec(aeskey, "AES");
            IvParameterSpec IVSpec   = new IvParameterSpec(iv);

            Cipher aes = Cipher.getInstance("AES/CBC/PKCS5Padding");
            aes.init(Cipher.ENCRYPT_MODE, aesKeySpec, IVSpec);
            byte[] aesCipherText = aes.doFinal(msg.toString().getBytes("UTF-8"));

            //encrypt AES key with RSA
            Cipher rsa = Cipher.getInstance("RSA");
            rsa.init(Cipher.ENCRYPT_MODE, recipient);
            byte[] encryptedAESKey = rsa.doFinal(aeskey);

            //"iv\RSA encrypted AES key\ciper text"
            return Crypto.Base64Encode(iv) + "\\" + Crypto.Base64Encode(encryptedAESKey) + "\\" +
                   Crypto.Base64Encode(aesCipherText);
        } catch (Exception e) {
            Logger.write("WARNING", "Crypto", "Unable to encrypt message: " + e);
        }
        return "";
    }

    public static Message decrypt(String msg) {
        Logger.write("INFO", "Crypto","decrypt()");
        try {
            //claim messages are the only plaintext in the system, still need decoding
            if (msg.substring(0,2).equals("c ")) {
                String decoding = new String(Crypto.Base64Decode(msg.substring(2)));
                return Message.parse(decoding);
            }

            String[] tokens = new String[3];
            StringTokenizer tokenizer = new StringTokenizer(msg, "\\", false);
            tokens[0] = tokenizer.nextToken();
            tokens[1] = tokenizer.nextToken();
            tokens[2] = tokenizer.nextToken();

            byte[] iv            = Crypto.Base64Decode(tokens[0]);
            byte[] cipheredKey   = Crypto.Base64Decode(tokens[1]);
            byte[] cipherText    = Crypto.Base64Decode(tokens[2]);

            //decrypt AES key
            Cipher rsa = Cipher.getInstance("RSA");
            rsa.init(Cipher.DECRYPT_MODE, getPrivateKey());
            byte[] aesKey = rsa.doFinal(cipheredKey);

            //decrypt AES Ciphertext
            SecretKeySpec aesKeySpec = new SecretKeySpec(aesKey, "AES");
            IvParameterSpec IVSpec = new IvParameterSpec(iv);
            Cipher aes = Cipher.getInstance("AES/CBC/PKCS5Padding");
            aes.init(Cipher.DECRYPT_MODE, aesKeySpec, IVSpec);
            byte[] messagePlaintext = aes.doFinal(cipherText);

            return Message.parse(new String(messagePlaintext));
        } catch (Exception e) {
            //This is to be expected for messages not addressed to you
            //Logger.write("WARNING", "Crypto", "Unable to decrypt message: " + e);
        }
        return new Message("NULL", "", 0, "");
    }

    public static String encodeKey (PublicKey key) {
        if (key != null) {
            return Base64Encode(key.getEncoded());
        } else {
            Logger.write("ERROR", "Crypto","encodeKey passed null key");
            return "--INVALID KEYSTRING--";
        }
    }

    public static PublicKey decodeKey (String codedKey) {
        if (codedKey != null) {
            try {
                return KeyFactory.getInstance("RSA").generatePublic(
                                    new X509EncodedKeySpec(Base64Decode(codedKey)));
            } catch (Exception e) {
                Logger.write("ERROR", "Crypto", "decodeKey(" + codedKey + ") passed invalid keystring");
                return null;
            }
        }
        Logger.write("WARNING", "Crypto", "decodeKey(...) returning null - passed invalid keystring");
        return null;
    }

    public static String Base64Encode (byte[] data) {
        return DatatypeConverter.printBase64Binary(data);
    }

    public static byte[] Base64Decode (String data) {
        return DatatypeConverter.parseBase64Binary(data);
    }

    public static int rand (int min, int max) {
        int range = max - min;
        return (int)(Math.random() * (range + 1)) + min;
    }

    public static byte[] encryptBytes (byte[] data, String key) {
        try {
            SecretKeySpec spec = new SecretKeySpec(getAESKey(key), "AES");
            Cipher cipher = Cipher.getInstance("AES");
            cipher.init(Cipher.ENCRYPT_MODE, spec);
            return cipher.doFinal(data);
        } catch (Exception e) {
            Logger.write("FATAL", "Crypto", "Could not encrypt bytes: " + e);
            return null;
        }
    }

    public static byte[] decryptBytes (byte[] data, String key) {
        try {
            SecretKeySpec spec = new SecretKeySpec(getAESKey(key), "AES");
            Cipher cipher = Cipher.getInstance("AES");
            cipher.init(Cipher.DECRYPT_MODE, spec);
            return cipher.doFinal(data);
        } catch (Exception e) {
            Logger.write("FATAL", "Crypto", "Could not decrypt bytes: " + e);
            return null;
        }
    }

    private static byte[] getAESKey(String password) {
        try {
            byte[] pwBytes = password.getBytes("UTF-8");
            KeyGenerator gen = KeyGenerator.getInstance("AES");
            SecureRandom srandAES = SecureRandom.getInstance("SHA1PRNG");
            srandAES.setSeed(pwBytes);
            gen.init(128, srandAES);
            SecretKey key = gen.generateKey();
            return key.getEncoded();
        } catch (Exception e) {
            Logger.write("FATAL", "Crypto", "Could not get AES key: " + e);
            return null;
        }
    }
}
```