```java
package edu.asu.ASUHelloWorldJavaFXMaven;

/**
 * <p> JUnitTest of Database Helper </p>
 *
 * <p> Description: A Junit 5 test suite to test the main automated functions of
 *     Group 47's DatabaseHelper class. The database helper class executes most of
 *     the database functionality for any user stored in the help system. These tests
 *     cover most of the critical functions in the class.</p>
 *
 * <p> Copyright: Alan Lintemuth with Group 47 © 2024 </p>
 *
 * @author Alan Lintemuth
 *
 * @version 1.00
 */

import static org.mockito.Mockito.*;

public class JUnitTest {

 /**
  * Creating the proper objects to test a mock database. Extra objects and imports
  * are included to facilitate future testing.
  */

    private DatabaseHelper dbHelper;
    private Connection mockConnection;
    private PreparedStatement mockPreparedStatement;
    private Statement mockStatement;

 /**
  * Creating the connection to an in-memory H2 database for testing
  */
   @BeforeEach
    void setUp() throws SQLException {
        String jdbcUrl = "jdbc:h2:mem:testdb";
        mockConnection = DriverManager.getConnection(jdbcUrl, "sa", "");

        //Creating the initial table as a test and for future testing
    try (Statement stmt = mockConnection.createStatement()){

     String userTable = "CREATE TABLE IF NOT EXISTS cse360users ("
       + "id INT AUTO_INCREMENT PRIMARY KEY, "
       + "password VARCHAR(255), "
       + "role VARCHAR(20), "
       + "access BOOLEAN, "
       + "email VARCHAR(255), "
       + "first VARCHAR(255), "
       + "middle VARCHAR(255), "
       + "last VARCHAR(255), "
       + "preferred VARCHAR(255), "
       + "USERNAME VARCHAR(255), "
       + "temp VARCHAR(255), "
       + "date VARCHAR(255))";

     stmt.execute(userTable);
    }

    //Attempting to fill the table with two different users
        try (Statement stmt = mockConnection.createStatement()) {
```

```
71            stmt.executeUpdate("INSERT INTO cse360users (password, role, access, email, first,
  middle, last, preferred, USERNAME, temp, date) "
72                    + "VALUES ('pass123', 'admin', true, 'admin', 'First', 'Middle', 'Last',
73            stmt.executeUpdate("INSERT INTO cse360users (password, role, access, email, first,
74                    + "VALUES ('pass456', 'user', false, 'user', 'SFirst', 'SMiddle', 'SLast',
75        }
76
77        dbHelper = new DatabaseHelper(mockConnection);
78    }
79
80    //Closing the database after the connection is finished
81    @AfterEach
82    void closeDown() throws SQLException {
83        if (mockConnection != null) {
84        mockConnection.close();
85        }
86    }
87
88 /**
89  * A test for the Access method, called when a user logs in for the first time
90  */
91    @Test
92    void testAccess() throws SQLException {
93        String username = "testUser";
94
95        dbHelper.access(username); //Call the method to test with the username
96
97        // Then
98        String query = "SELECT access FROM cse360users WHERE username = ?";
99        try (PreparedStatement pstmt = mockConnection.prepareStatement(query)) {
100           pstmt.setString(1, username); //Set the variables in the query
101       }
102   }
103
104 /**
105  * A test for the Register method, called when a user creates their account
106  */
107   @Test
108   void testRegister() throws SQLException {
109       String username = "testUser";
110       String password = "testPassword";
111       String role = "student";
112
113       //Function call to test with the new variables
114       dbHelper.register(username, password, role); // Call the method to test
115
116       String query = "SELECT * FROM cse360users WHERE username = ? AND role = ?";
117       try (PreparedStatement pstmt = mockConnection.prepareStatement(query)) {
118           pstmt.setString(1, username);
119           pstmt.setString(2, role);     //Set the variables in the query
120
121           //Test what is stored in the database against the new values
122           try (ResultSet rs = pstmt.executeQuery()) {
123               assertTrue(rs.next(), "User should be inserted into the database");
124               assertEquals(username, rs.getString("username"));
125               assertEquals(role, rs.getString("role"));
126               String encryptedPasswordFromDb = rs.getString("password");
127               String expectedEncryptedPassword = Base64.getEncoder().encodeToString
128               assertEquals(expectedEncryptedPassword, encryptedPasswordFromDb, "Passwords
129           }
130       }
```

```
131     }
132
133 /**
134  * A test for the invitedata method, called when a user is invited to the app
135  */
136     @Test
137     void testInvitedata() throws SQLException {
138         String role = "role";
139         String temp = "temp123";
140         String date = "2024-11-20";
141
142         //Function call to test with the new variables
143         dbHelper.invitedata(role, temp, date);
144
145         //Testing the data stored in the database
146         String query = "SELECT * FROM cse360users WHERE role = ? AND temp = ? AND date = ?";
147         try (PreparedStatement pstmt = mockConnection.prepareStatement(query)) {
148             pstmt.setString(1, role);
149             pstmt.setString(2, temp);
150             pstmt.setString(3, date);
151
152             try (ResultSet rs = pstmt.executeQuery()) {
153                 assertTrue(rs.next(), "User should be inserted into the database");
154                 assertEquals(role, rs.getString("role"));
155                 assertEquals(temp, rs.getString("temp"));
156                 assertEquals(date, rs.getString("date"));
157                 }
158         }
159     }
160
161 /**
162  * A test for the update method, called when a user logs in for the 1st time
163  */
164     @Test
165     void testUpdate() throws SQLException {
166         String username = "testUser";
167         String initialEmail = "initialEmail@example.com";
168         String initialFirst = "InitialFirst";
169         String initialMiddle = "InitialMiddle";
170         String initialLast = "InitialLast";
171         String initialPreferred = "InitialPreferred";
172
173         //Insert initial data into the database for the given username
174         String insertUser = "INSERT INTO cse360users (username, email, first, middle, last,
175         try (PreparedStatement pstmt = mockConnection.prepareStatement(insertUser)) {
176             pstmt.setString(1, username);
177             pstmt.setString(2, initialEmail);
178             pstmt.setString(3, initialFirst);
179             pstmt.setString(4, initialMiddle);
180             pstmt.setString(5, initialLast);
181             pstmt.setString(6, initialPreferred);
182             pstmt.executeUpdate();
183         }
184
185         //New data to update
186         String newEmail = "newEmail@example.com";
187         String newFirst = "NewFirst";
188         String newMiddle = "NewMiddle";
189         String newLast = "NewLast";
190         String newPreferred = "NewPreferred";
191
```

```
192         //Function call to test with the new variables
193         dbHelper.update(newEmail, newFirst, newMiddle, newLast, newPreferred, username);
194 
195         //Check if the data is updated in the database
196         String query = "SELECT * FROM cse360users WHERE username = ?";
197         try (PreparedStatement pstmt = mockConnection.prepareStatement(query)) {
198             pstmt.setString(1, username);
199 
200             try (ResultSet rs = pstmt.executeQuery()) {
201                 assertTrue(rs.next(), "User should be updated in the database");
202                 assertEquals(newEmail, rs.getString("email"));
203                 assertEquals(newFirst, rs.getString("first"));
204                 assertEquals(newMiddle, rs.getString("middle"));
205                 assertEquals(newLast, rs.getString("last"));
206                 assertEquals(newPreferred, rs.getString("preferred"));
207                 assertTrue(rs.getBoolean("access"));
208             }
209         }
210     }
211 
212 /**
213  * A test for the resetuser method, called when a user's username and password
214  * is reset.
215  */
216     @Test
217     void testResetUser() throws SQLException {
218         String username = "testUser";
219         String temp = "temp123"; //Temp value required by the function
220         String date = "2024-11-20";
221 
222         //Insert initial data into the database for the given username
223         String insertUser = "INSERT INTO cse360users (username, temp, password) VALUES (?, ?,
224         try (PreparedStatement pstmt = mockConnection.prepareStatement(insertUser)) {
225             pstmt.setString(1, username);
226             pstmt.setString(2, temp);
227             pstmt.setString(3, "oldPassword");  // Set an initial password
228             pstmt.executeUpdate();
229         }
230 
231         //Confirm the user is inserted
232         String checkInsertQuery = "SELECT username, temp, password FROM cse360users WHERE
233         try (PreparedStatement pstmt = mockConnection.prepareStatement(checkInsertQuery)) {
234             pstmt.setString(1, username);
235             try (ResultSet rs = pstmt.executeQuery()) {
236                 assertTrue(rs.next(), "User should be inserted into the database");
237                 assertEquals(temp, rs.getString("temp"));
238                 assertNotNull(rs.getString("password"), "Password should not be null
239             }
240         }
241 
242         //Function call to test with the new variables
243         dbHelper.resetuser(username, temp, date);
244 
245         //Check that the password is null, the temp is updated, and the date is set
246         String query = "SELECT password, temp, date FROM cse360users WHERE username = ?";
247         try (PreparedStatement pstmt = mockConnection.prepareStatement(query)) {
248             pstmt.setString(1, username);
249             try (ResultSet rs = pstmt.executeQuery()) {
250                 assertTrue(rs.next(), "User should be found in the database with the specified
251                 assertNull(rs.getString("password"), "Password should be set to null");
252                 assertEquals(temp, rs.getString("temp"), "Temp value should remain the same");
```

```
253                 assertEquals(date, rs.getString("date"), "Date should be updated");
254             }
255         }
256     }
257
258 /**
259  * A test for the setrole method, called when a user signs up and logs in the 1st time
260  * is reset.
261  */
262     @Test
263     void testSetRole() throws SQLException {
264         String username = "testuser";
265         String initialRole = "student";
266         dbHelper.register(username, "password", initialRole);
267
268         //Function call to test with the new variables
269         String newRole = "admin";
270         dbHelper.setrole(newRole, username);
271
272         //Verify the role is updated in the database
273         String query = "SELECT role FROM cse360users WHERE username = ?";
274         try (PreparedStatement pstmt = mockConnection.prepareStatement(query)) {
275             pstmt.setString(1, username);
276             try (ResultSet rs = pstmt.executeQuery()) {
277                 assertTrue(rs.next(), "User should be found in the database");
278                 assertEquals(newRole, rs.getString("role"), "Role should be updated to the new
279             }
280         }
281     }
282
283 /**
284  * A test for the login method, called when a user logs. Check the username and password match
285  */
286     @Test
287     void testlogin() throws SQLException {
288         String username = "testuser";
289         String password = "password123";  // This will be encrypted in the login method
290         dbHelper.register(username, password, "student");
291
292         //Function call to test with the new variables
293         boolean loginSuccessful = dbHelper.login(username, password);
294
295         //The login (Boolean) should be true
296         assertTrue(loginSuccessful, "Login should be successful with correct username and
297     }
298
299 /**
300  * A test for the helptemp method, checks if a user has a valid one time password
301  */
302     @Test
303     void testhelptemp() throws SQLException {
304         String temp = "temp123";
305         String username = "testuser";
306
307         dbHelper.register(username, "password123", "student");
308
309         String insertTemp = "INSERT INTO cse360users (username, temp) VALUES (?, ?)";
310         try (PreparedStatement pstmt = mockConnection.prepareStatement(insertTemp)) {
311             pstmt.setString(1, username);
312             pstmt.setString(2, temp);
313             pstmt.executeUpdate();
```

```
314         }
315
316         //Function call to test with the new variables
317         boolean result = dbHelper.helptemp(temp);
318
319         //The result should be true if the temp exists in the database
320         assertTrue(result, "Temp should exist in the database");
321     }
322
323 /**
324  * A test for the doesUserExist method, checks if a user exists in the database
325  */
326     @Test
327     void testDoesUserExistWithExistingUser() throws SQLException {
328         //A user "testuser" exists in the database
329         String username = "testuser";
330
331         try (Statement stmt = mockConnection.createStatement()) {
332             stmt.executeUpdate("INSERT INTO cse360users (USERNAME) VALUES ('testuser')");
333             mockConnection.commit();
334         }
335
336         dbHelper = new DatabaseHelper(mockConnection);
337
338         //Function call to test with the new variables
339         boolean result = dbHelper.doesUserExist(username);
340
341         //The result should be true if the user exists
342         assertTrue(result, "User should exist in the database");
343     }
344
345 /**
346  * A test for the doesUserExist method, checks if a user does not exist
347  * in the database
348  */
349     @Test
350     void testDoesUserExistWithNonExistingUser() throws SQLException {
351         //A user "nonuser" does not exist in the database
352         String username = "nonuser"; // Non-existing user
353
354         //Function call to test with the new variables
355         boolean result = dbHelper.doesUserExist(username);
356
357         //The result should be false since the user does not exist
358         assertFalse(result, "User should not exist in the database");
359     }
360
361 /**
362  * A test for the prefname method, checks if a user's preferred name is stored
363  * in the database
364  */
365     @Test
366     void testPrefname() throws SQLException {
367         String username = "testUser";
368         String prefname = "Pref";
369
370         try (Statement stmt = mockConnection.createStatement()) {
371             stmt.executeUpdate("INSERT INTO cse360users (USERNAME, preferred) "
372                     + "VALUES ('testUser', 'Pref')");
373             mockConnection.commit();
374         }
```

```
375
376         //Verify insertion
377         try (Statement stmt = mockConnection.createStatement()) {
378             ResultSet rs = stmt.executeQuery("SELECT * FROM cse360users WHERE USERNAME =
379             while (rs.next()) {
380                 System.out.println("Inserted User: " + rs.getString("USERNAME") + ", Preferred:
381             }
382         }
383
384         //Call the prefname method to retrieve the preferred name
385         String result = dbHelper.prefname(username);
386     }
387
388 /**
389  * A test for the displayUsersByAdmin method, displays all users to an admin
390  */
391     @Test
392     void testDisplayUsersByAdmin() throws SQLException {
393         String insertUser1 = "INSERT INTO cse360users (username, password, email, role) VALUES
394         int user1Id = 0;
395         try (PreparedStatement pstmt = mockConnection.prepareStatement(insertUser1,
396             pstmt.setString(1, "user1");
397             pstmt.setString(2, "password1"); // This will be stored as plain text for testing
398             pstmt.setString(3, "test1@example.com");
399             pstmt.setString(4, "admin");
400             pstmt.executeUpdate();
401
402             // Retrieve the generated ID for the first user
403             try (ResultSet rs = pstmt.getGeneratedKeys()) {
404                 if (rs.next()) {
405                     user1Id = rs.getInt(1);
406                 }
407             }
408         }
409
410         String insertUser2 = "INSERT INTO cse360users (username, password, email, role) VALUES
411         int user2Id = 0;
412         try (PreparedStatement pstmt = mockConnection.prepareStatement(insertUser2,
413             pstmt.setString(1, "user2");
414             pstmt.setString(2, "password2");
415             pstmt.setString(3, "test2@example.com");
416             pstmt.setString(4, "user");
417             pstmt.executeUpdate();
418
419             // Retrieve the generated ID for the second user
420             try (ResultSet rs = pstmt.getGeneratedKeys()) {
421                 if (rs.next()) {
422                     user2Id = rs.getInt(1);
423                 }
424             }
425         }
426
427         //Function call to test with the new variables
428         dbHelper.displayUsersByAdmin();
429
430         //Verify that the data for these users exists in the database using the generated IDs
431         String query = "SELECT * FROM cse360users WHERE id = ?";
432
433         //Check for first user using the generated ID
434         try (PreparedStatement pstmt = mockConnection.prepareStatement(query)) {
435             pstmt.setInt(1, user1Id);
```

```
436             try (ResultSet rs = pstmt.executeQuery()) {
437                 assertTrue(rs.next(), "First user should be present in the database");
438                 assertEquals("test1@example.com", rs.getString("email"));
439                 assertEquals("password1", rs.getString("password"));
440                 assertEquals("admin", rs.getString("role"));
441             }
442         }
443 
444         //Check for second user using the generated ID
445         try (PreparedStatement pstmt = mockConnection.prepareStatement(query)) {
446             pstmt.setInt(1, user2Id);
447             try (ResultSet rs = pstmt.executeQuery()) {
448                 assertTrue(rs.next(), "Second user should be present in the database");
449                 assertEquals("test2@example.com", rs.getString("email"));
450                 assertEquals("password2", rs.getString("password"));
451                 assertEquals("user", rs.getString("role"));
452             }
453         }
454     }
455 }
```

```java
package edu.asu.ASUHelloWorldJavaFXMaven;


/**
 * <p> JUnitTest of ArticleDatabaseHelper </p>
 *
 * <p>  The ArticleDatabaseHelper class handles most of our article database
 *  functionality. This testing insures those functions are performing correctly.
 *  Mockito was imported in case any team members preferred to test with that.</p>
 *
 * <p> Copyright: Alan Lintemuth with Group 47 © 2024 </p>
 *
 * @author Alan Lintemuth
 *
 * @version 1.00
 */

import static org.mockito.Mockito.*;  //

public class JUnitArticleTest {

/*****
 * Our functionality creates a virtual H2 database in memory to test
 * the functions in ArticleDatabaseHelper
 */
    private ArticleDatabaseHelper dbArtHelper;
    private Connection connection;

    @BeforeEach
    void setUp() throws SQLException {
        //Use an in-memory database for testing
        String jdbcUrlArt = "jdbc:h2:mem:testdb";
        connection = DriverManager.getConnection(jdbcUrlArt, "sa", "");

        //Set up and test the database, matches ArticleDatabaseHelper

    try (Statement stmt = connection.createStatement()){

            String articleTable = "CREATE TABLE IF NOT EXISTS articles ("
                    + "id BIGINT PRIMARY KEY, "
                    + "title VARCHAR(255), "
                    + "authors VARCHAR(255), "
                    + "abstract TEXT, "
                    + "keywords VARCHAR(255), "
                    + "body TEXT, "
                    + "references TEXT, "
                    + "level VARCHAR(255), "
                    + "groups TEXT)";
            stmt.execute(articleTable);
    }

    try (Statement stmt = connection.createStatement()){

            String thirdTables = "CREATE TABLE IF NOT EXISTS groups ("
                        + "groupID BIGINT PRIMARY KEY, "
                        + "groupTitle TEXT, "
                        + "groupArticles TEXT, "
                        + "specialAccess BOOLEAN, "
                        + "groupAdmin TEXT, "
                        + "groupAccess TEXT) ";
            stmt.execute(thirdTables);
```

```
69              }
70
71      try (Statement stmt = connection.createStatement()){
72              String messageTable = "CREATE TABLE IF NOT EXISTS message ("
73                      + "id INT AUTO_INCREMENT PRIMARY KEY, "
74                      + "type VARCHAR(255), "
75                      + "body TEXT)";
76              stmt.execute(messageTable);
77          }
78
79          try (Statement stmt = connection.createStatement()) {
80              stmt.executeUpdate("DELETE FROM articles");  //Remove existing articles
81              stmt.executeUpdate("DELETE FROM groups");    //Remove existing groups
82          }
83
84          try (Statement stmt = connection.createStatement()) {
85              //Insert test data
86              stmt.executeUpdate("INSERT INTO articles (id, title, authors, abstract, keywords,
87                      + "VALUES (1, 'title', 'authors', 'abstract', 'keys', 'body', 'references',
88              stmt.executeUpdate("INSERT INTO groups (groupID, groupTitle, groupArticles,
89                      + "VALUES (7, 'Gtitle', 'Garticles', true, 'Gadmin', 'Gaccess') ");
90          }
91
92          dbArtHelper = new ArticleDatabaseHelper(connection);
93      }
94
95      @AfterEach
96      void closeDown() throws SQLException {
97          if (connection != null) {
98              connection.close();  //Close the database connection after each test
99          }
100     }
101
102 /**
103  * Some basic connection testing
104  */
105     @Test
106     void testArticleDatabaseHelperConstructor() throws SQLException {
107         // Test that the constructor initializes dbArtHelper and other components
108         assertNotNull(dbArtHelper, "ArticleDatabaseHelper should be initialized.");
109
110         // Check if connection was properly set up
111         assertTrue(connection.isValid(2), "The database connection should be valid.");
112     }
113
114     @Test
115     void testDatabaseTableCreation() throws SQLException {
116         // Ensure that data inserted into the test database can be queried successfully
117         try (Statement stmt = connection.createStatement()) {
118             ResultSet rs = stmt.executeQuery("SELECT COUNT(*) FROM articles");
119             if (rs.next()) {
120                 int count = rs.getInt(1);
121                 assertTrue(count > 0, "There should be at least one article in the database.");
122             }
123         }
124     }
125
126 /**
127  * A test for the createGroup method, called when a new group is saved in the
128  * groups table
129  */
```

```java
    @Test
    void testCreateGroup() throws SQLException {
        //Set up test data for creating a group
        String title = "Gtitle";
        String articles = "Garticles";
        Boolean specialAccess = true;
        String groupAdmin = "Gadmin";
        String groupAccess = "Gaccess";

        //Create a new group
        dbArtHelper.createGroup(title, articles, specialAccess, groupAdmin, groupAccess);

        //Verify that the group was created
        assertTrue(dbArtHelper.groupExists(title), "Group should exist in the database after
    }

/**
 * A test for the groupExists method, called to check if a group is saved in the
 * groups table. This test is if a group does exist (calls by groupTitle)
 */
    @Test
    void testGroupExists() throws SQLException {
        //Set up a group for testing
        String title = "Gtitle";
        String articles = "Garticles";
        Boolean specialAccess = true;
        String groupAdmin = "Gadmin";
        String groupAccess = "Gaccess";

        //Function call to test with the new variables
        dbArtHelper.createGroup(title, articles, specialAccess, groupAdmin, groupAccess);

        //Check if the group exists
        boolean exists = dbArtHelper.groupExists(title);

        //The group should exist
        assertTrue(exists, "Group exists");
    }

/**
 * A test for the groupExists method, called to check if a group is saved in the
 * groups table. This test is if a group does not exist (calls by groupTitle)
 */
    @Test
    void testGroupNotExists() throws SQLException {
        //A group that has not been created
        String title = "NonGroup";

        //Check if the non-existent group exists
        boolean exists = dbArtHelper.groupExists(title);

        //The group should not exist
        assertFalse(exists, "Group should not exist");
    }

/**
 * A test for the viewallmessage method, called to show all the messages left
 * by students. This method also tests the savemessage function.
 */
    @Test
    void testViewAllMessages() throws Exception {
```

```
191         //Insert some test messages, testing the savemessage function
192     dbArtHelper.savemessage(1, "Generic message content");
193     dbArtHelper.savemessage(2, "Specific message content");
194 
195         //Retrieve all messages
196         String messages = dbArtHelper.viewallmessage();
197 
198         //Verify that the messages are correctly retrieved
199         assertTrue(messages.contains("Type: Generic message"), "should contain Generic
200         assertTrue(messages.contains("Type: Specific message"), "should contain Specific
201 
202         //Check if the actual message bodies are present
203         assertTrue(messages.contains("Body of message: Generic message content"), "should
204         assertTrue(messages.contains("Body of message: Specific message content"), "should
205     }
206 
207 /**
208  * A test for the isSpecialGroup method, called to check if a group has special status
209  */
210     @Test
211     void testIsSpecialGroup() throws SQLException {
212         //Set up a group
213         String title = "Special Group";
214         String articles = "Article1, Article2";
215         Boolean specialAccess = true;
216         String groupAdmin = "adminUser";
217         String groupAccess = "read";
218 
219         //Function call to test with the new variables
220         dbArtHelper.createGroup(title, articles, specialAccess, groupAdmin, groupAccess);
221 
222         //Check if the group is marked as special
223         boolean isSpecial = dbArtHelper.isSpecialGroup(title);
224 
225         //The group should have special access
226         assertTrue(isSpecial, "The group is special");
227     }
228 
229 /**
230  * A test for the isSpecialGroup method, called to check if a group has special status
231  */
232     @Test
233     void testAddArticleToGroup() throws SQLException {
234         //Set up a group and an article
235         String title = "Group1";
236         String articles = "Article1";
237         Boolean specialAccess = true;
238         String groupAdmin = "adminUser";
239         String groupAccess = "read";
240 
241         dbArtHelper.createGroup(title, articles, specialAccess, groupAdmin, groupAccess);
242 
243         String article = "Article3";
244 
245         //Add an article to the group
246         dbArtHelper.addArticleToGroup(title, article);
247 
248         //Verify the article was added to the group
249         String sql = "SELECT groupArticles FROM groups WHERE groupTitle = ?";
250         try (PreparedStatement pstmt = connection.prepareStatement(sql)) {
251             pstmt.setString(1, title);
```

```
252             ResultSet rs = pstmt.executeQuery();
253             if (rs.next()) {
254                 String updatedArticles = rs.getString("groupArticles");
255                 assertTrue(updatedArticles.contains(article), "The article should be added");
256             }
257         }
258     }
259
260 /**
261  * A test for the checkInArticles method, called to check if an article is in a group
262  */
263     @Test
264     void testCheckInArticles() throws SQLException {
265         //Set up a group with a list of articles
266         String group = "Gtitle";
267         String newArticle = "Article1";
268         String groupArticles = "Article1, Article2, Article3";
269
270         //Manually generate a unique GROUPID
271         String getGroupId = "SELECT MAX(groupID) FROM groups";
272         long groupId = 1; // Default to 1 for the first insert
273
274         try (PreparedStatement maxIdStmt = connection.prepareStatement(getGroupId);
275                 ResultSet maxIdRs = maxIdStmt.executeQuery()) {
276                if (maxIdRs.next()) {
277                    groupId = maxIdRs.getLong(1) + 1;
278                }
279            }
280
281         //Insert the group into the database with the GROUPID
282         String insertGroupSql = "INSERT INTO groups (groupID, groupTitle, groupAccess) VALUES
283         try (PreparedStatement stmt = connection.prepareStatement(insertGroupSql)) {
284             stmt.setLong(1, groupId);
285             stmt.setString(2, group);
286             stmt.setString(3, groupArticles);
287             stmt.executeUpdate();
288         }
289     }
290
291 /**
292  * A test for the removeArticleFromGroup method, called to an article from a group
293  * This method check an article that exists
294  */
295     @Test
296     void testRemoveArticleFromGroup_ArticleExists() throws SQLException {
297         //Set up a group with some articles
298         String group = "Gtitle";
299         String article = "Article1";
300         String initialArticles = "Article1, Article2, Article3";
301
302         //Same manual generation
303         String getGroupId = "SELECT MAX(groupID) FROM groups";
304         long groupId = 1;
305
306         try (PreparedStatement maxIdStmt = connection.prepareStatement(getGroupId);
307                 ResultSet maxId = maxIdStmt.executeQuery()) {
308                if (maxId.next()) {
309                    groupId = maxId.getLong(1) + 1;  // Increment the maximum GROUPID
310                }
311            }
312
```

```
313         //Insert the group into the database with the GROUPID
314         String insertGroup = "INSERT INTO groups (groupID, groupTitle, groupAccess) VALUES (?,
315         try (PreparedStatement stmt = connection.prepareStatement(insertGroup)) {
316             stmt.setLong(1, groupId);
317             stmt.setString(2, group);
318             stmt.setString(3, initialArticles);
319             stmt.executeUpdate();
320         }
321
322         //Remove the article from the group
323         dbArtHelper.removeArticleFromGroup(group, article);
324
325         //Verify that the article was removed from the group
326         String updatedGroup = null;
327         try (PreparedStatement stmt = connection.prepareStatement(
328                 "SELECT groupArticles FROM groups WHERE groupTitle = ?")) {
329             stmt.setString(1, group);
330             ResultSet rs = stmt.executeQuery();
331             if (rs.next()) {
332             updatedGroup = rs.getString("groupArticles");
333             }
334         }
335
336         //Verify that the updated groupArticles no longer contains the article
337         assertFalse(updatedGroup.contains(article), "Article should be removed from the
338     }
339
340 /**
341  * A test for the addAccessToGroup method, called to give an admin or instructor access to a
342  */
343     @Test
344     void testAddAccessToGroup() throws SQLException {
345     //Set up a group with some initial access
346     String group = "Gtitle";
347     String initialAccess = "user1";
348     String newAccess = "user3";
349
350     String getGroupId = "SELECT MAX(groupID) FROM groups";
351     long groupId = 1;
352
353     try (PreparedStatement maxIdStmt = connection.prepareStatement(getGroupId);
354       ResultSet maxIdRs = maxIdStmt.executeQuery()) {
355       if (maxIdRs.next()) {
356        groupId = maxIdRs.getLong(1) + 1;  // Increment the maximum GROUPID
357       }
358     }
359
360    String insertGroup = "INSERT INTO groups (groupID, groupTitle, groupAccess) VALUES (?, ?,
361    try (PreparedStatement stmt = connection.prepareStatement(insertGroup)) {
362     stmt.setLong(1, groupId);
363     stmt.setString(2, group);
364     stmt.setString(3, initialAccess);
365     stmt.executeUpdate();
366    }
367
368    //Verify group exists before proceeding
369    String checkGroup = "SELECT 1 FROM groups WHERE groupTitle = ?";
370    try (PreparedStatement checkStmt = connection.prepareStatement(checkGroup)) {
371     checkStmt.setString(1, group);
372     try (ResultSet checkRs = checkStmt.executeQuery()) {
373      assertTrue(checkRs.next(), "Group should exist in the database before adding access.");
```

```
374     }
375     }
376
377     //Add new access to the group
378     dbArtHelper.addAccessToGroup(group, initialAccess);
379     dbArtHelper.addAccessToGroup(group, newAccess);
380
381     //Verify that the new access has been added to the groupAccess list
382     String sql = "SELECT groupAccess FROM groups WHERE groupTitle = ?";
383     try (PreparedStatement stmt = connection.prepareStatement(sql)) {
384      stmt.setString(1, group);
385      try (ResultSet rs = stmt.executeQuery()) {
386             assertTrue(rs.next(), "Group should exist in the database");
387
388             //Check the groupAccess field
389             String updatedAccess = rs.getString("groupAccess");
390             assertTrue(updatedAccess.contains(newAccess), "New access should be added to
391             assertTrue(updatedAccess.contains(initialAccess), "Initial access should remain in
392     }
393     }
394      }
395
396 /**
397  * A test for the removeAccessFromGroup method, called to remove access from an admin or
398  * This test removes a user with valid access
399  */
400     @Test
401     void testRemoveAccessFromGroup_ValidAccess() throws SQLException {
402         String group = "groupA";
403         String user = "user1";
404
405         String getGroupId = "SELECT MAX(groupID) FROM groups";
406         long groupId = 1;
407
408         try (PreparedStatement maxIdStmt = connection.prepareStatement(getGroupId);
409              ResultSet maxIdRs = maxIdStmt.executeQuery()) {
410             if (maxIdRs.next()) {
411                 groupId = maxIdRs.getLong(1) + 1;
412             }
413         }
414
415         String insertGroup = "INSERT INTO groups (groupID, groupTitle, groupAdmin, groupAccess)
416         try (PreparedStatement stmt = connection.prepareStatement(insertGroup)) {
417             stmt.setLong(1, groupId);
418             stmt.setString(2, group);
419             stmt.setString(3, user);
420             stmt.setString(4, "");
421             stmt.executeUpdate();
422         }
423
424         //Call the method to remove the access
425         dbArtHelper.removeAccessFromGroup(group, user);
426
427         //Get the updated groupAccess
428         String currentAccess = "";
429         try (PreparedStatement ps = connection.prepareStatement("SELECT groupAccess FROM groups
430             ps.setString(1, group);
431             ResultSet rs = ps.executeQuery();
432             if (rs.next()) {
433                 currentAccess = rs.getString("groupAccess");
434             }
```

```
435         }
436
437         //Assert that the access is removed from the groupAccess
438         assertFalse(currentAccess.contains(user), "The access should be removed from the
439     }
440
441 /**
442  * A test for the removeAccessFromGroup method, called to remove access from an admin or
443  * This test tries to remove an invalid user
444  */
445     @Test
446     void testRemoveAccessFromGroup_AccessNotFound() throws SQLException {
447         String group = "groupA";
448         String userValid = "user1";
449         String userInvalid = "user2";
450
451         String getGroupId = "SELECT MAX(groupID) FROM groups";
452         long groupId = 1;
453
454         try (PreparedStatement maxIdStmt = connection.prepareStatement(getGroupId);
455              ResultSet maxIdRs = maxIdStmt.executeQuery()) {
456             if (maxIdRs.next()) {
457                 groupId = maxIdRs.getLong(1) + 1;
458             }
459         }
460
461         String insertGroup = "INSERT INTO groups (groupID, groupTitle, groupAdmin, groupAccess)
462         try (PreparedStatement stmt = connection.prepareStatement(insertGroup)) {
463             stmt.setLong(1, groupId);
464             stmt.setString(2, group);
465             stmt.setString(3, userValid);
466             stmt.setString(4, "");
467             stmt.executeUpdate();
468         }
469
470         //Get the original groupAccess before calling the method
471         String originalAccess = "";
472         try (PreparedStatement ps = connection.prepareStatement("SELECT groupAccess FROM groups
473             ps.setString(1, group);
474             ResultSet rs = ps.executeQuery();
475             if (rs.next()) {
476                 originalAccess = rs.getString("groupAccess");
477             }
478         }
479
480         //Call removeAccessFromGroup with an invalid user (userInvalid)
481         dbArtHelper.removeAccessFromGroup(group, userInvalid);
482
483         //Verify that the groupAccess has not changed
484         String currentAccess = "";
485         try (PreparedStatement ps = connection.prepareStatement("SELECT groupAccess FROM groups
486             ps.setString(1, group);
487             ResultSet rs = ps.executeQuery();
488             if (rs.next()) {
489                 currentAccess = rs.getString("groupAccess");
490             }
491         }
492
493         //Assert that the groupAccess remains unchanged
494         assertEquals(originalAccess, currentAccess, "The access list shouldn't change.");
495     }
```

```
496
497 /**
498  * A test for the addAdminToGroup method, called to add access for an admin or instructor
499  */
500    @Test
501    void testAddAdminToGroup() throws SQLException {
502    //A group exists and does not have 'admin1' as a member.
503    String group = "groupA";
504    String admin = "admin1";
505
506    String getGroupId = "SELECT MAX(groupID) FROM groups";
507    long groupId = 1;
508
509    try (PreparedStatement maxIdStmt = connection.prepareStatement(getGroupId);
510     ResultSet maxIdRs = maxIdStmt.executeQuery()) {
511     if (maxIdRs.next()) {
512      groupId = maxIdRs.getLong(1) + 1;
513     }
514    }
515
516    String insertGroup = "INSERT INTO groups (groupID, groupTitle, groupAccess) VALUES (?, ?,
517    try (PreparedStatement stmt = connection.prepareStatement(insertGroup)) {
518     stmt.setLong(1, groupId);
519     stmt.setString(2, group);
520     stmt.setString(3, "");
521     stmt.executeUpdate();
522    }
523
524    //Add an admin to the group
525    dbArtHelper.addAdminToGroup(group, admin);
526
527    //Verify the admin was added to the group
528    String sql = "SELECT groupAdmin FROM groups WHERE groupTitle = ?";
529    try (PreparedStatement pstmt = connection.prepareStatement(sql)) {
530     pstmt.setString(1, group);
531     ResultSet rs = pstmt.executeQuery();
532     if (rs.next()) {
533      String admins = rs.getString("groupAdmin");
534      assertTrue(admins.contains(admin), "Admin should be added to the group.");
535     }
536    }
537    }
538
539 /**
540  * A test for the removeAdminFromGroup method, called to remove access for an admin or
541  * This test is for an admin that is already in the group
542  */
543    @Test
544    void testRemoveAdminFromGroup() throws SQLException {
545        //A group exists and has 'admin1' as a member.
546        String group = "groupA";
547        String admin = "admin1";
548
549        String getGroupId = "SELECT MAX(groupID) FROM groups";
550        long groupId = 1;
551
552        try (PreparedStatement maxIdStmt = connection.prepareStatement(getGroupId);
553             ResultSet maxIdRs = maxIdStmt.executeQuery()) {
554            if (maxIdRs.next()) {
555                groupId = maxIdRs.getLong(1) + 1;
556            }
```

```
557         }
558
559         String insertGroup = "INSERT INTO groups (groupID, groupTitle, groupAdmin, groupAccess)
560         try (PreparedStatement stmt = connection.prepareStatement(insertGroup)) {
561             stmt.setLong(1, groupId);
562             stmt.setString(2, group);
563             stmt.setString(3, admin);
564             stmt.setString(4, "");
565             stmt.executeUpdate();
566         }
567
568         //Call the method to remove the admin from the group
569         dbArtHelper.removeAdminFromGroup(group, admin);
570
571         //Verify the admin was removed from the group
572         String sql = "SELECT groupAdmin FROM groups WHERE groupTitle = ?";
573         try (PreparedStatement pstmt = connection.prepareStatement(sql)) {
574             pstmt.setString(1, group);
575             ResultSet rs = pstmt.executeQuery();
576             if (rs.next()) {
577                 String admins = rs.getString("groupAdmin");
578                 assertFalse(admins.contains(admin), "Admin should be removed from the group.");
579             }
580         }
581     }
582
583 /**
584  * A test for the removeAdminFromGroup method, called to remove access for an admin or
585  * This test is for an admin that is not in the group
586  */
587     @Test
588     void testRemoveAdminThatDoesNotExist() throws SQLException {
589         //A group exists and does not have 'admin1' as a member.
590         String group = "groupB";
591         String admin = "admin1";
592
593
594         String getGroupId = "SELECT MAX(groupID) FROM groups";
595         long groupId = 1;
596
597         try (PreparedStatement maxIdStmt = connection.prepareStatement(getGroupId);
598              ResultSet maxIdRs = maxIdStmt.executeQuery()) {
599             if (maxIdRs.next()) {
600                 groupId = maxIdRs.getLong(1) + 1;
601             }
602         }
603
604         String insertGroup = "INSERT INTO groups (groupID, groupTitle, groupAdmin, groupAccess)
605         try (PreparedStatement stmt = connection.prepareStatement(insertGroup)) {
606             stmt.setLong(1, groupId);
607             stmt.setString(2, group);
608             stmt.setString(3, "admin2");
609             stmt.setString(4, "");
610             stmt.executeUpdate();
611         }
612
613         //Try to remove an admin ('admin1') who is not part of the group
614         dbArtHelper.removeAdminFromGroup(group, admin);
615
616         //Verify the admin was not removed because they were not part of the group
617         String sql = "SELECT groupAdmin FROM groups WHERE groupTitle = ?";
```

```
618         try (PreparedStatement pstmt = connection.prepareStatement(sql)) {
619             pstmt.setString(1, group);
620             ResultSet rs = pstmt.executeQuery();
621             if (rs.next()) {
622                 String admins = rs.getString("groupAdmin");
623                 //Check that 'admin2' is still in the list of admins.
624                 assertTrue(admins.contains("admin2"), "Admin 'admin2' should still be in the
625             }
626         }
627     }
628
629 /**
630  * A test for the checkInAdmin method, called to check if the user is an admin in the group
631  * This test is for a group that does not exist
632  */
633     @Test
634     void testCheckInAdmin_GroupNotFound() throws SQLException {
635         String group = "NonExistentGroup";
636         String newAdmin = "newAdmin";
637
638         //Call the method to check if the user is an admin
639         boolean result = dbArtHelper.checkInAdmin(group, newAdmin);
640
641         // erify that the result is false since the group does not exist
642         assertFalse(result, "The group does not exist, so the admin can't be checked.");
643     }
644
645 /**
646  * A test for the checkInAdmin method, called to check if the user is an admin in the group
647  * This test is for a group that does exist, but that user is not the admin
648  */
649     @Test
650     void testCheckInAdmin_NewAdminNotInGroup() throws SQLException {
651         String group = "GroupWithAdmins";
652         String existingAdmin = "existingAdmin";
653         String newAdmin = "newAdmin";
654
655         //Generate groupID dynamically based on existing groups
656         String getGroupId = "SELECT MAX(groupID) FROM groups";
657         long groupId = 1;
658
659         try (PreparedStatement maxIdStmt = connection.prepareStatement(getGroupId);
660              ResultSet maxIdRs = maxIdStmt.executeQuery()) {
661             if (maxIdRs.next()) {
662                 groupId = maxIdRs.getLong(1) + 1;
663             }
664         }
665
666         //Insert a group with an existing admin
667         String insertGroup = "INSERT INTO groups (groupID, groupTitle, groupAdmin, groupAccess)
668         try (PreparedStatement stmt = connection.prepareStatement(insertGroup)) {
669             stmt.setLong(1, groupId);
670             stmt.setString(2, group);
671             stmt.setString(3, existingAdmin);  // Only one admin
672             stmt.setString(4, "");  // No access
673             stmt.executeUpdate();
674         }
675
676         //Call the method to check if the new admin is in the group
677         boolean result = dbArtHelper.checkInAdmin(group, newAdmin);
678
```

```
679         //Assert that the result is false since the new admin is not in the list
680         assertFalse(result, "The new admin is not in the group, so the result should be
681     }
682
683 /**
684  * A test for the checkInAdmin method, called to check if the user is an admin in the group
685  * This test is for a group that does exist, and that user is the admin
686  */
687     @Test
688     void testCheckInAdmin_NewAdminInGroup() throws SQLException {
689         String group = "GroupWithAdmins";
690         String existingAdmin = "existingAdmin";
691         String newAdmin = "existingAdmin";  // newAdmin is the same as existingAdmin
692
693         //Generate groupID dynamically based on existing groups
694         String getGroupId = "SELECT MAX(groupID) FROM groups";
695         long groupId = 1;
696
697         try (PreparedStatement maxIdStmt = connection.prepareStatement(getGroupId);
698              ResultSet maxIdRs = maxIdStmt.executeQuery()) {
699             if (maxIdRs.next()) {
700                 groupId = maxIdRs.getLong(1) + 1;
701             }
702         }
703
704         //Insert a group with the existing admin
705         String insertGroup = "INSERT INTO groups (groupID, groupTitle, groupAdmin, groupAccess)
706         try (PreparedStatement stmt = connection.prepareStatement(insertGroup)) {
707             stmt.setLong(1, groupId);
708             stmt.setString(2, group);
709             stmt.setString(3, existingAdmin);  // The group already has this admin
710             stmt.setString(4, "");  // No access
711             stmt.executeUpdate();
712         }
713
714         //Call the method to check if the new admin (which is the same as existingAdmin) is in
715         boolean result = dbArtHelper.checkInAdmin(group, newAdmin);
716
717         //Assert that the result is true since the new admin is already in the list
718         assertTrue(result, "The new admin is already an admin in the group, so the result should
719     }
720
721 /**
722  * A test for the addArticle method, called to add an article to the database
723  */
724     @Test
725     void testAddArticle() throws Exception {
726         //Define article details
727         String title = "TestTitle";
728         String authors = "TestAuthor";
729         String abstractText = "TestAbstract.";
730         String keywords = "TestKey";
731         String body = "TestBody, should be encrypted.";
732         String references = "TestReference";
733         String level = "Beginner";
734         String groups = "TestGroup";
735
736         //Add the article to the database
737         dbArtHelper.addArticle(title, authors, abstractText, keywords, body, references, level,
738
739         //Verify the article was inserted into the database
```

```
740         String selectArticle = "SELECT * FROM articles WHERE title = ?";
741         try (PreparedStatement pstmt = connection.prepareStatement(selectArticle)) {
742             pstmt.setString(1, title);
743             ResultSet rs = pstmt.executeQuery();
744 
745             //Ensure that the article was inserted and retrieved
746             if (rs.next()) {
747                 //Check the article
748                 long id = rs.getLong("id");
749                 String retrievedTitle = rs.getString("title");
750                 String retrievedAuthors = rs.getString("authors");
751                 String retrievedAbstract = rs.getString("abstract");
752                 String retrievedKeywords = rs.getString("keywords");
753                 String encryptedBody = rs.getString("body");
754                 String retrievedReferences = rs.getString("references");
755                 String retrievedLevel = rs.getString("level");
756                 String retrievedGroups = rs.getString("groups");
757 
758                 //Check that the data matches the inserted values
759                 assertEquals(title, retrievedTitle);
760                 assertEquals(authors, retrievedAuthors);
761                 assertEquals(abstractText, retrievedAbstract);
762                 assertEquals(keywords, retrievedKeywords);
763                 assertEquals(references, retrievedReferences);
764                 assertEquals(level, retrievedLevel);
765                 assertEquals(groups, retrievedGroups);
766 
767                 //Check that the body is properly encrypted (Base64 encoded)
768                 String decodedBody = new String(Base64.getDecoder().decode(encryptedBody));
769                 assertEquals(body, decodedBody);  //Verify that the decoded body matches the
770 
771                 //Verify that the unique ID is greater than 0 (indicating it's a valid ID)
772                 assertTrue(id > 0, "Unique ID should be greater than 0.");
773             }
774         }
775     }
776 
777 /**
778  * A test for the addRestoredArticle method, called to add a restored article to the database
779  */
780     @Test
781     void testAddRestoredArticle() throws Exception {
782         //Article details to be inserted
783         Long id = 100L;
784         String title = "Sample Article Title";
785         String authors = "Author One, Author Two";
786         String abstractText = "This is an abstract of the article.";
787         String keywords = "keyword1, keyword2";
788         String body = "This is the body of the article, already encrypted.";
789         String references = "Reference 1, Reference 2";
790         String level = "Level 1";
791         String groups = "Group 1, Group 2";
792 
793         //Call the addRestoredArticle method to insert the article
794         dbArtHelper.addRestoredArticle(id, title, authors, abstractText, keywords, body,
795 
796         //Verify that the article was successfully added
797         String query = "SELECT * FROM articles WHERE id = ?";
798         try (PreparedStatement ps = connection.prepareStatement(query)) {
799             ps.setLong(1, id);
800             ResultSet rs = ps.executeQuery();
```

```
801
802             //Assert that the article is inserted
803             assertTrue(rs.next(), "The article should be inserted into the database.");
804
805             //Verify the fields in the article
806             assertEquals(id, rs.getLong("id"), "The article ID should match.");
807             assertEquals(title, rs.getString("title"), "The article title should match.");
808             assertEquals(authors, rs.getString("authors"), "The article authors should match.");
809             assertEquals(abstractText, rs.getString("abstract"), "The article abstract should
810             assertEquals(keywords, rs.getString("keywords"), "The article keywords should
811             assertEquals(body, rs.getString("body"), "The article body should match.");
812             assertEquals(references, rs.getString("references"), "The article references should
813             assertEquals(level, rs.getString("level"), "The article level should match.");
814             assertEquals(groups, rs.getString("groups"), "The article groups should match.");
815         }
816     }
817
818 /**
819  * A test for the deleteArticle method, called to delete an article from the database
820  * This test is for a valid article
821  */
822     @Test
823     void testDeleteArticle() throws SQLException {
824         //Define article details
825         String title = "TempTitle";
826         String authors = "TempAuthor";
827         String abstractText = "TempAbstract.";
828         String keywords = "TempKeys";
829         String body = "TempBody";
830         String references = "TempReference";
831         String level = "Beginner";
832         String groups = "TempGroup";
833
834         //Insert an article into the database, the catch should fail as there is no exception
835         try {
836   dbArtHelper.addArticle(title, authors, abstractText, keywords, body, references, level,
837  } catch (Exception e) {
838   e.printStackTrace();
839  }
840
841         //Retrieve the ID of the inserted article
842         String selectArticleSql = "SELECT id FROM articles WHERE title = ?";
843         long articleId = 0;
844         try (PreparedStatement pstmt = connection.prepareStatement(selectArticleSql)) {
845             pstmt.setString(1, title);
846             ResultSet rs = pstmt.executeQuery();
847             if (rs.next()) {
848                 articleId = rs.getLong("id");
849             }
850         }
851
852         //Delete the article using its ID
853         dbArtHelper.deleteArticle(articleId);
854
855         //Verify that the article was deleted, console also prints the trace from the method
856         String checkDeletedSql = "SELECT * FROM articles WHERE id = ?";
857         try (PreparedStatement pstmt = connection.prepareStatement(checkDeletedSql)) {
858             pstmt.setLong(1, articleId);
859             ResultSet rs = pstmt.executeQuery();
860             if (!rs.next()) {
861                 System.out.println("Test article successfully deleted.");
```

```
862             }
863         }
864     }
865
866 /**
867  * A test for the deleteArticle method, called to delete an article from the database
868  * This test is for an article that does not exist
869  */
870     @Test
871     void testDeleteNonExistingArticle() throws SQLException {
872         //Define a non-existing article ID
873         long nonExistingArticleId = 9999;  //This ID should not exist in the database
874
875         //Try to delete a non-existing article
876         dbArtHelper.deleteArticle(nonExistingArticleId);
877
878         //Ensure that no exception is thrown and the article was not found
879         String checkNonExistingSql = "SELECT * FROM articles WHERE id = ?";
880         try (PreparedStatement pstmt = connection.prepareStatement(checkNonExistingSql)) {
881             pstmt.setLong(1, nonExistingArticleId);
882             ResultSet rs = pstmt.executeQuery();
883             if (!rs.next()) {
884                 System.out.println("No article found with ID: " + nonExistingArticleId);
885             }
886         }
887     }
888
889 /**
890  * A test for the filterbybeginner method, called to filter articles with difficulty
891  * set at 'beginner'
892  */
893     @Test
894     void testFilterByBeginner() throws Exception {
895         //Define article details
896         String title1 = "BeginnerArticle";
897         String authors1 = "Author1";
898         String abstractText1 = "BeginnerAbs";
899         String keywords1 = "BeginnerKey";
900         String body1 = "Beginnerbody";
901         String references1 = "BeginnerRef";
902         String level1 = "Beginner";
903         String groups1 = "BeginnerGroup";
904
905         String title2 = "IntermediateArticle";
906         String authors2 = "Author2";
907         String abstractText2 = "IntermediateAbs";
908         String keywords2 = "IntermediateKey";
909         String body2 = "Intermediatebody";
910         String references2 = "IntermediateRef";
911         String level2 = "Intermediate";
912         String groups2 = "IntermediateGroup";
913
914         //Add the article to the database
915         dbArtHelper.addArticle(title1, authors1, abstractText1, keywords1, body1, references1,
916         dbArtHelper.addArticle(title2, authors2, abstractText2, keywords2, body2, references2,
917
918         //Call the filterbybeginner method to test the 2 articles we added
919         String result = dbArtHelper.filterbybeginner("Beginner");
920
921         //Check if the result contains the expected "Beginner Article"
922         assertTrue(result.contains("Title: " + title1), "Result should contain the Beginner
    Article title");
```

```
923         assertTrue(result.contains("Authors: " + authors1), "Result should contain the Beginner
    Article author");
924         assertTrue(result.contains("Abstract: " + abstractText1), "Result should contain the
925
926         // Ensure that the "Intermediate Article" is not included in the result
927         assertFalse(result.contains("Title: " + title2), "Result should not contain the
928         assertFalse(result.contains("Authors: " + authors2), "Result should not contain the
929         assertFalse(result.contains("Abstract: " + abstractText2), "Result should not contain
930     }
931 }
```