Database Tutorial 5: Transactions and JDBC

December 8, 2017

1 Repetition

1.1 Permissions

- GRANT Grants permissions (e.g. SELECT, INSERT, ...) on tables and columns and (e.g. CREATE, CONNECT, ...) on databases, including permissions to grant permissions to others

```
GRANT { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER }
[,...] | ALL [ PRIVILEGES ] }
ON { [ TABLE ] table_name [, ...]
TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { { SELECT | INSERT | UPDATE | REFERENCES } ( column [, ...] )
[,...] | ALL [ PRIVILEGES ] ( column [, ...] ) }
ON [ TABLE ] table_name [, ...]
TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]
```

- REVOKE Revokes permissions granted with GRANT

1.2 JDBC

- Connect

```
import java.sql.*; // JDBC stuff.
import java.util.Properties;

Class.forName("org.postgresql.Driver");
String url = "jdbc:postgresql://ate.ita.chalmers.se/";
Properties props = new Properties();
props.setProperty("user",USERNAME);
props.setProperty("password",PASSWORD);
Connection conn = DriverManager.getConnection(url, props);
```

- Create Queries
  - Statement

    ```
    String query = "SELECT * FROM Users;";
    Statement pst = con.createStatement();
    ResultSet rs = stmt.executeQuery(query);
    ```

    Affected by SQL injection, should be avoided or only used with fixed queries.
  - PreparedStatement

    ```
    String query = "SELECT * FROM Users WHERE Uid = ?;";
    PreparedStatement pst = con.prepareStatement(query);
    pst.setString(1,"1");
    ResultSet rs = pst.executeQuery()
    ```

    Avoids SQL injection by properly escaping input, always use setter functions to set values

- Results
  - ```
    String query = ... // any valid SQL query
    PreparedStatement pst = con.prepareStatement(query);
    Bool r = pst.execute()
    ```

    Indicates for of first result: ResultSet → true, otherwise → false
  - ```
    String query = ... // any valid SQL query
    PreparedStatement pst = con.prepareStatement(query);
    ResultSet rs = pst.executeQuery()
    // the pointer initially points before the first row
    while (rs.next())
    {
      int uid = rs.getInt(1); // Get value by index
      String name = rs.getString("username"); // Get value by column name
    }
    ```

    Returns a ResultSet object that contains the data produced by the query
  - ```
    String query = ... // Only INSERT, UPDATE or DELETE
    PreparedStatement pst = con.prepareStatement(query);
    int r = pst.executeUpdate()
    ```

    Either count of affected rows or 0 for statements that return nothing Throws:

1.3 Transactions

1.4 Interference

- Dirty Reads: a transaction reads data from a partial transaction before the second one is rolled back

  Time →

	A = 1	A = 2	A = 1
T1		R(A),A=2	
T2	A=2,W(A)	Rollback	

- Non-repeatable reads: a transcation reads data that gets changed or deleted by another transaction before the first transaction is finished

  Time →

	A = 1	A = 2		
T1	R(A), A = 1		R(A), A = 2	Commit
T2		A = 2,W(A)	Commit	

- Phantoms: new data matching a search condition, that is queried in a transaction, is added before the transaction is finished

  Time →

T1	SELECT * FROM Users WHERE Age > 30 ;		...	Commit
T2		INSERT INTO Users (id,age) VALUES (23,42) ;	Commit	

1.5 Isolation levels

- READ UNCOMMITTED: transaction allows other transactions to modify the database while running, everything *changed* affects the reads of this transaction
- READ COMMITTED: the transaction allows other transactions to modify the database while running, everything *committed* affects the reads of this transaction
    ```

- REPEATABLE READ: like REPEATABLE READ but for every repeated read we get *at least* the same tuple again
- SERIALIZABLE: no other transaction may interfere with it in any way

	Dirty Reads	Non-repeatable reads	Phantoms
READ UNCOMMITTED	Yes	Yes	Yes
READ COMMITTED	No	Yes	Yes
REPEATABLE READ	No	No	Yes
SERIALIZABLE	No	No	No

2 Exercises

1. (10 points) Consider an existing database with the following database definition in a PostgreSQL DBMS:

```
CREATE TABLE Users (id INTEGER PRIMARY KEY, name TEXT, password TEXT);

CREATE TABLE UserStatus (id INTEGER PRIMARY KEY REFERENCES Users, loggedin BOOLEAN NOT NULL);

CREATE TABLE Logbook (id INTEGER REFERENCES Users, timestamp INTEGER, name TEXT,
  PRIMARY KEY (id, timestamp)
);
```

   (a) (4 points) A database user "Alice" is granted the following permissions:

```
GRANT SELECT(id, name, password) ON Users TO Alice;
GRANT SELECT(id, loggedin) ON UserStatus TO Alice;
GRANT SELECT(id, timestamp, name) ON LogBook TO Alice;
GRANT INSERT(id, timestamp, name) ON LogBook TO Alice;
```

   Alice now executes the following SQL statement:

```
INSERT INTO LogBook
SELECT u.id, 201701101400, u.name
FROM (UserStatus us JOIN Users u ON us.id = u.id)
WHERE us.loggedin = True ;
```

   We want Alice to only have exactly the privileges that are necessary to complete this SQL statement. Does Alice have too few, exactly enough, or too many privileges? What minimal set of permissions should she be granted instead, if not the same as listed above?

   Solution:
   Alice has too many privileges, since she does not need to read the password in the Users table, nor the LogBook entries. The minimally required set of permissions is:

```
GRANT SELECT (id, name) ON Users TO Alice ;
GRANT SELECT (id, loggedin) ON UserStatus TO Alice ;
GRANT INSERT (id, timestamp, name) ON LogBook TO Alice ;
```

   (b) (2 points) Users of a web application are allowed to query this database for a certain user id. This functionality is implemented in JDBC using the following code fragment:

```
...
String query =
  "SELECT * FROM UserStatus WHERE id = ' " + userinput + " ' " ;
PreparedStatement stmt = conn.prepareStatement(query) ;
ResultSetrs = stmt.executeQuery();
...
```

   Does this code contain an SQL injection vulnerability? If it does not, why not? If it does, how would you correct the code?

Solution:
Yes this code contains an SQL injection vulnerability. The vulnerability can be removed by either correctly sanitizing or escaping the data in the userinput variable. A better solution is to use a PreparedStatement with placeholder:

```
...
String query = "SELECT * FROM UserStatus WHERE id = ? ";
PreparedStatement stmt = conn.prepareStatement(query);
stmt.setString (1, userinput);
ResultSet rs = stmt.executeQuery();
...
```

(c) (4 points) The following transaction calculates the total number of entries in UserStatus as the sum of the number of logged-in and not logged-in users.

```
BEGIN TRANSACTION ISOLATION LEVEL READ COMMITTED;
SELECT
(SELECT COUNT( * ) FROM UserStatus WHERE loggedIn = True)
+
(SELECT COUNT( * ) FROM UserStatus WHERE loggedIn = False ) ;
COMMIT;
```

The used transaction isolation level is not sufficient to ensure an accurate count of entries in UserStatus. Why not? Give all isolation levels that are sufficient so that the query works as expected.

Solution:
The transaction is vulnerable to "non-repeatable read" and "phantom read" interferences, because the READ COMMITTED transaction isolation level does not protect against them. The stronger REPEATABLE READ isolation level is not sufficient because it still allows phantom reads. Only the SERIALIZABLE isolation level is sufficient, since it protects against dirty read, non-repeatable read and phantom read.

2. (12 points) Consider the same database as above.

(a) (4 points) A database user "Alice" is granted the following permissions:

```
GRANT SELECT(id, name , password ) ON Users TO Alice;
GRANT SELECT(id, loggedin) ON UserStatus TO Alice;
GRANT INSERT(id, loggedin) ON UserStatus TO Alice;
GRANT SELECT(id, timestamp , name ) ON LogBook TO Alice;
```

Alice now executes the following SQL statement:

```
INSERT INTO LogBook
SELECT u.id , 201706071400, u.name
FROM (UserStatus us JOIN Users u ON us.id = u.id)
WHERE us.loggedin = True;
```

We want Alice to only have exactly the privileges that are necessary to complete this SQL statement. Does Alice have the correct privileges? What minimal set of permissions should she be granted instead, if not the same as listed above?

Solution:
No, Alice does not have the correct privileges. She does not need to read the password in the Users table, she does not need INSERT privileges on UserStatus, and she does not need SELECT privileges on LogBook. In addition, Alice lacks INSERT privileges on LogBook.
The minimally required set of permissions is:

```
GRANT SELECT (id, name) ON Users TO Alice ;
GRANT SELECT (id, loggedin) ON UserStatus TO Alice ;
GRANT INSERT (id, timestamp, name) ON LogBook TO Alice ;
```

(b) (4 points) Users of a web application are allowed to query this database for a certain user id. This functionality is implemented in JDBC using the following code fragment:

```
...
String query = "SELECT * FROM UserStatus WHERE id = ' " + userinput + " ' ";
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery(query);
...
```

The userinput variable is controlled directly by an attacker. What can an attacker input into userinput so that the SQL query returns all data in UserStatus? What is this specific security problem called? How should the above code be corrected to prevent this problem?

Solution: The attacker can input something like *' or '1'='1*, so that the query turns into

```
SELECT FROM UserStatus WHERE id = '' or '1'= '1';
```

(mind the closing quote).
This is an example of an SQL injection vulnerability or attack. The vulnerability can be removed by either correctly sanitizing or escaping the data in the userinput variable. A better solution is to use a PreparedStatement with placeholder:

```
...
String query = "SELECT * FROM UserStatus WHERE id = ? ";
PreparedStatement stmt = conn.prepareStatement(query);
stmt.setString (1, userinput);
ResultSet rs = stmt.executeQuery();
...
```

(c) (Bonus points)