JDBC

Devdatta Kulkarni

Databases and Java

- JDBC
  - Java Database Connectivity
  - MySQL Connectors
    - http://dev.mysql.com/downloads/connector/
- Example
  - JDBC

JDBC topics

- Maven dependency
  - commons-dbcp
  - mysql-connector-java
- Main Concepts/Classes
  - Data source
  - Connection
  - PreparedStatement
  - ResultSet
  - Transactions

Data Source

- A *data source* is where the data is stored by your application (or is accessed from)
  - E.g.: DBMS, a legacy file system
- A JDBC application connects to a target data source using one of the following classes
  - DriverManager
    - Automatically loads any JDBC 4.0 drivers available on the class path
  - DataSource
    - Abstracts the details of the underlying data source from your application

DriverManager

- Database connection is established using the DriverManager’s “getConnection” method
  - dbURL = "jdbc:mysql://localhost:3306/student_courses";
  - conn = DriverManager.*getConnection(dbURL,*"*devdatta*", ""*);*

DataSource

```
// Setup data source
BasicDataSource ds = new BasicDataSource();
ds.setUsername(this.dbUsername);
ds.setPassword(this.dbPassword);
ds.setUrl(this.dbURL);
ds.setDriverClassName("com.mysql.jdbc.Driver");

// Open connection
Connection conn = ds.getConnection();
```

Connection class

- Connection
  - Represents a database connection
    - Typical required parameters: Database URL, username, password
- Connection object is *not* thread safe

- How to support application’s multiple threads?
  - Option 1: Accessing Connection through synchronized methods
    - Not a good strategy
      - Different threads may be accessing different data items; such threads may end up blocking each other unnecessarily
  - Option 2: Use a separate connection per thread
    - Not a good strategy either
      - Each thread typically represents a user request; your application will have lots of users
      - It cannot open new Connections for all (it will run out of resources)

Connection Pool

- Connection Pooling
  - Keep a pool of open connections
  - Each thread will get its own connection
  - Threads will block if there are no connections available in the pool
  - A connection is given back to the pool once a thread is done using it
  - Apache DBCP
    - Library that implements connection pooling
    - http://commons.apache.org/proper/commons-dbcp/

PreparedStatement

- Represents a *parameterized SQL query*
- Parameter values are set based on data provided at runtime
- Use “executeQuery” to execute a statement that returns some value
- Use “executeUpdate” to execute a statement that does not return a value

PreparedStatement

```
String query = "select * from courses where course_id=?";

Connection conn = ds.getConnection();

PreparedStatement s = conn.prepareStatement(query);

s.setString(1, String.valueOf(courseId));

ResultSet r = s.executeQuery();
```

PreparedStatement

- “Parameter binding provides a means of separating executable code, such as SQL, from content, transparently handling content encoding and escaping.“ [1]

[1] http://martinfowler.com/articles/web-security-basics.html#BindParametersForDatabaseQueries

PreparedStatement – Getting keys back

```
String insert = "INSERT INTO courses(name,
course_num) VALUES(?, ?)";

PreparedStatement stmt =
conn.prepareStatement(insert,
Statement.RETURN_GENERATED_KEYS);

stmt.setString(1, c.getName());
stmt.setString(2, c.getCourseNum());

int affectedRows = stmt.executeUpdate();
```

ResultSet

- Represents a set of records from the table that satisfy the specified query criteria
- The records are accessed through a *cursor*
- The cursor is a pointer that points to one row of data in the ResultSet
- Initially the cursor is positioned before the first row
- The method ResultSet.next moves the cursor to the next row

ResultSet – Returning a single row

```
ResultSet r = s.executeQuery();

        if (!r.next()) {
            return null;
        }

NewCourse c = new NewCourse();
c.setCourseNum(r.getString("course_num"));
c.setName(r.getString("name"));
c.setCourseId(r.getInt("course_id"));
```

ResultSet – Returning multiple rows

```
ResultSet r = s.executeQuery();

List<NewCourse> courseList = new
ArrayList<NewCourse>();

while (!r.next()) {
      NewCourse c = new NewCourse();
      c.setCourseNum(r.getString("course_num"));
      c.setName(r.getString("name"));
      c.setCourseId(r.getInt("course_id"));
      courseList.add(c);
}
return courseList;
```

ResultSet Types

- Determines how does the *cursor* moves
  - TYPE_FORWARD_ONLY
    - Cursor only moves forward
  - TYPE_SCROLL_INSENSITIVE
    - The result can be scrolled; the result set is insensitive to changes made to the underlying data source while it is open
  - TYPE_SCROLL_SENSITIVE
    - The result can be scrolled; the result set reflects the changes made to the underlying data source while it is open

Transactions

- Problem:
  - How to ensure that more than one database statements are executed as a unit?
- Solution:
  - Transactions

Example where a transaction is needed

- Suppose we want to create a new project and add a meeting as part of project creation:

```
POST /myeavesdrop/projects/
<project>
   <name>NewSolum</name>
   <description>NewSolum</description>
   <meetings>
     <meeting>
        <name>M1</name>
        <year>2016</year>
        <month>March</month>
        <day>26</day>
     </meeting>
   </meetings>
</project>
```

Transactions

- We need to define a transaction that includes the following two statements:
  - Insert the new project's name and description into the "projects" table
  - Insert the project's meeting information into the "meetings" table

  - We need to execute both these within a single transaction
    - Either both are committed successfully or both are not
      - The execution of the two statements needs to be *'atomic'*

Transactions

- A transaction is a set of statements that are executed as a *unit (also called the ‘atomicity’ property of a transaction)*
  - Either all statements in a transaction are completed successfully or none are
  - The effect of executing a statement is reflected within a database only after it is *committed*

- A *Connection* is in auto-commit mode by default
  - Each individual statement is treated as a transaction
  - If we want to group two or more statements into a transaction then we have to *disable* the auto-commit mode
    - conn.setAutoCommit(false);
  - Commit/rollback of the transaction is our responsibility
    - conn.commit();
    - conn.rollback();

- https://docs.oracle.com/javase/tutorial/jdbc/basics/transactions.html

Transactions

- A simple approach for the database system to guarantee atomicity (*all or nothing property*) is to perform only one transaction at a time

- But this approach would limit performance severely

- So we want transactions to be executed concurrently to gain performance benefits

- However, then we need to solve for *isolation*

Isolation

- What is isolation?
  - At high-level it means preserving data integrity by ensuring that concurrently executing transactions do not interfere with one another

  - What does this mean?
    - Prevent transactions from interfering with one another
    - For example, if two transactions are modifying the same row then the *isolation* property guarantees that none of the transactions would see an *intermediate* state of the row

How to provide isolation?

- We know how to prevent concurrent updates to shared state
  - Use locks
- At a high-level that is exactly what the database does
  - It *locks* the rows that are part of the transaction
  - After a lock is set, it remains in effect till either the transaction is committed or rolled back
  - No other transaction can access the rows that are locked

Transaction Isolation

- Instead of all-or-nothing isolation, can we do better to improve performance?

- Define different types of locks (isolation levels) that provide different concurrency guarantees
  - Read_uncommitted
  - Read_committed
  - Repeatable_read
  - Seriazable
    - Guarantees that there will no dirty reads, no non-repeatable reads, and no phantom reads

Concurrent Transactions: Issues

- Read uncommited level leads to *Dirty reads*
  - A transaction reads a row from a database table containing uncommitted changes from another transaction.
- Read commited leads to *Non-repeatable reads*
  - A transaction reads a row from a database table, a second transaction changes the same row and the first transaction re-reads the row and gets a different value.
- Repeatable reads leads to *Phantom reads*
  - A transaction re-executes a query, returning a set of rows that satisfies a search condition and finds that another committed transaction has inserted additional rows that satisfy the condition.
- http://www.onjava.com/pub/a/onjava/2001/05/23/j2ee.html?page=2

Transaction Isolation Levels

- Read Uncommitted
  - Read data that is not yet committed (dirty reads)
- Read Committed
  - No dirty reads; but non-repeatable reads
- Repeatable Read
  - No dirty reads
  - No non-repeatable reads
- Serializable
  - No dirty reads
  - No non-repeatable reads
  - No phantom reads

  - Example:
    - findTransactionIsolationLevel
    - testRepeatableRead
    - testSerializableRead

Transaction Isolation Levels

- Serializable
  - http://sqlperformance.com/2014/04/t-sql-queries/the-serializable-isolation-level
- Repeatable read
  - http://sqlperformance.com/2014/04/t-sql-queries/the-repeatable-read-isolation-level
- Why use read-uncommitted?
  - http://stackoverflow.com/questions/2471055/why-use-a-read-uncommitted-isolation-level

JDBC Reference

- https://github.com/devdattakulkarni/ModernWebApps/tree/master/JDBC
- http://stackoverflow.com/questions/4246646/mysql-java-get-id-of-the-last-inserted-value-jdbc
- http://stackoverflow.com/questions/42648/best-way-to-get-identity-of-inserted-row
- http://stackoverflow.com/questions/8146793/no-suitable-driver-found-for-jdbcmysql-localhost3306-mysql

JDBC Reference

- http://stackoverflow.com/questions/9428573/is-it-safe-to-use-a-static-java-sql-connection-instance-in-a-multithreaded-syste
- http://stackoverflow.com/questions/7592056/am-i-using-jdbc-connection-pooling
- Apache DBCP2:
  - http://svn.apache.org/viewvc/commons/proper/dbcp/trunk/doc/BasicDataSourceExample.java?view=markup

JDBC References

- Auto increment id
  - http://stackoverflow.com/questions/1915166/how-to-get-the-insert-id-in-jdbc
- Foreign Key
  - http://stackoverflow.com/questions/25920251/how-to-automatically-insert-foreign-key-references-in-tables-in-mysql-or-jdbc
- Triggers
  - http://dev.mysql.com/doc/refman/5.7/en/trigger-syntax.html