
Workbook 5

Introduction

Last week you wrote your own implementation of a Java chat server. This week you will extend your implementation of the Java chat server and integrate a database. The database will be used to store some basic statistics on the use of the chat server together with a record of the conversations which take place. In addition, you will modify the server to replay the last ten messages sent between participants to any new user connecting to the server.

Important

An on-line version of this guide is available at:

<http://www.cl.cam.ac.uk/teaching/0910/FJava>

You should check this page regularly for announcements and errata. You might find it useful to refer to the on-line version of this guide in order to follow any provided web links or to cut 'n' paste example code.

Using a database in Java

All of the functionality described in this workbook could, in principle, be achieved by writing data into files in the filesystem. However databases provide several advantages over the filesystem interface in Java. In the context of this workbook, the database provides two distinct advantages: (1) a rich query language which permits the retrieval of precisely the data required from the database; and (2) support for concurrent transactions, permitting more than one Java thread to update data stored in the database without corruption.

There are a huge number of database systems to choose from. In this workbook you will use HSQLDB (<http://www.hsqldb.org/>), a database written in Java which supports Structured Query Language (SQL). Knowledge of SQL is not an examinable part of this course. You will cover this topic area in much greater detail in the Part 1B Databases course next term.

In order to use HSQLDB, you will need a database driver. Please download the following jar file from the course website now and save it in your home directory:

<http://www.cl.cam.ac.uk/teaching/0910/FJava/hsqldb.jar>

You will need to tell Eclipse to use this jar file when running your program. Later in the workbook, when you first run an application which accesses a database, you will need to choose **Run As...** and in the dialog which appears select the "Classpath" tab and add `hsqldb.jar` as an external jar.

HSQLDB is able to store a record of all the data in the database as a small set of files in the filesystem, and this is the method we will use today. A production version of the Java chat server would use a more sophisticated configuration in which the database was run as a separate operating system process, the details of which are beyond the scope of this course. To load the HSQLDB driver and create or load a database with a *path prefix* of `/home/crsid/chat-database` you need to perform the following steps in Java:

```
Class.forName("org.hsqldb.jdbcDriver");
Connection connection = DriverManager.getConnection("jdbc:hsqldb:file:"
+ "/home/crsid/chat-database", "SA", "");

Statement delayStmt = connection.createStatement();
try {delayStmt.execute("SET WRITE_DELAY FALSE");} //Always update data on disk
finally {delayStmt.close();}
```

This creates a small number of files whose names all start with `/home/crsid/chat-database` and ensures that any changes to the database are also made to the filesystem. In the above code, and in the remaining examples in the rest of this section, the classes used to talk to the database can be found in the package `java.sql`. For example, the fully qualified name for `Connection` is `java.sql.Connection`. Note that the above six lines are the only lines you will write which are specific to HSQLDB. All the remaining code presented in this Workbook will work with any SQL database.

Later on in this workbook you will modify the Java chat server you wrote last week to write to the database from within instances of the `ClientHandler` class. In this workbook you should manually control when a transaction is committed to the database and therefore you will need to do the following:

```
connection.setAutoCommit(false);
```

In an SQL database, data is stored in one or more *tables*. Each table has one or more columns and zero or more rows and each column has a type. For example, in order to store a record of all the `RelayMessage` objects sent to your Java chat server, you might create a table with three columns: a column to record the nickname of the individual who sent the message, a second column to record the message contents, and a third column to record the time. Each row in the table can then be used to record a specific message sent by an individual to the Java chat server at a specific time. The following piece of SQL creates such a table:

```
Statement sqlStmt = connection.createStatement();
try {
    sqlStmt.execute("CREATE TABLE messages(nick VARCHAR(255) NOT NULL, "+
                    "message VARCHAR(4096) NOT NULL,timeposted BIGINT NOT NULL)");
} catch (SQLException e) {
    System.out.println("Warning: Database table \"messages\" already exists.");
} finally {
    sqlStmt.close();
}
```

In the above snippet of code, the programmer has first got a handle on a new `Statement` object by using an instance of the `Connection` class you created earlier. This object is then used to execute an SQL query on the database. The query itself is written inside a Java `String`. The table is called `messages` and contains three columns. The first column is called `nick` and is of type `VARCHAR(255)`, which means it can hold a string of up to 255 characters in length; the phrase `NOT NULL` means that the database will not permit the storage of nothing, a string of some description must be provided. The column `message` is of type `VARCHAR(4096)` and is therefore able to store a string of up to 4096 characters. Finally, the column `timeposted` records the time at which the message was sent; the type `BIGINT` is a 64-bit integer value, equivalent to a Java `long`.

Rows can be added to the table using the SQL command `INSERT`. Here is an example which adds one row to the `messages` table defined above:

```
String stmt = "INSERT INTO MESSAGES(nick,message,timeposted) VALUES (?, ?, ?)";
PreparedStatement insertMessage = connection.prepareStatement(stmt);
try {
    insertMessage.setString(1, "Alastair"); //set value of first "?" to "Alastair"
    insertMessage.setString(2, "Hello, Andy");
    insertMessage.setLong(3, System.currentTimeMillis());
    insertMessage.executeUpdate();
} finally { //Notice use of finally clause here to finish statement
    insertMessage.close();
}
```

In the above example a different kind of SQL statement, a `PreparedStatement`, is used. This type of statement is useful when providing values from variables in Java. In the above, the three values to be added to the new row are substituted with question marks (?) in the statement. These question

marks are replaced with values drawn from Java variables inside the try block. For example, the first question mark (representing the value for the column `nick`) is updated with "Alastair" within the call to `setString`. This method of submitting data to the database looks laborious, but it is important to use this method. The alternative, preparing your own `String` object with the values held inside it directly, is likely to lead to error since many careful checks are needed (with string length being just one of them). It's good practice to use the `PreparedStatement` class to do this for you.

The database supports multiple simultaneous `Connection` objects. Each of these objects permit concurrent modifications (such as creating tables or adding rows to the database), and the results of any changes made to the database are *isolated* until the method `commit` is called on the `Connection` object. In other words:

```
connection.commit();
```

When `commit` is called, the thread of execution blocks until all the outstanding SQL statements which have been performed in isolation are written to the database for all other threads to see. Furthermore, all the statements are added in an *atomic* fashion and consequently all views of the database are consistent.

Data stored in tables can be retrieved by using the SQL `SELECT` statement:

```
stmt = "SELECT nick,message,timeposted FROM messages "+
        "ORDER BY timeposted DESC LIMIT 10";
PreparedStatement recentMessages = connection.prepareStatement(stmt);
try {
    ResultSet rs = recentMessages.executeQuery();
    try {
        while (rs.next())
            System.out.println(rs.getString(1)+" : "+rs.getString(2)+
                               " ["+rs.getLong(3)+"]");
    } finally {
        rs.close();
    }
} finally {
    recentMessages.close();
}
```

This query returns the top ten most recent posts made by users, latest first. The data returned contains the contents of the columns `nick`, `message` and `timeposted`. The contents of the top ten rows are returned encapsulated inside a `ResultSet` object. Notice how the object `rs` is used to interact with the database—each call to `rs.next` loads the next row of data into the `ResultSet` object `rs`, and calls to `rs.getString` or `rs.getLong` are used to retrieve the individual column elements of that row. Also, pay particular attention to the use of the `finally` clause—it's important to call `close` on any instance of `ResultSet` or `PreparedStatement` after data has been collected, both in the case where execution proceeds normally, and in the case where an `SQLException` object is thrown when executing the method `recentMessages.executeQuery` or `rs.next`; the `finally` clause does this neatly.

Whenever your Java program terminates, make sure you close all open database connections:

```
connection.close();
```

Important

Full documentation of HSQLDB are available on-line:

<http://hsqldb.org/doc/guide/index.html>

This workbook has so far only covered the creation of tables, the addition of rows, and the recall of data from a single database table; the `UPDATE` query will be described briefly in the next section. Knowl-

edge of this subset of features is sufficient to complete this Workbook, however you will probably find it helpful for your Group Project work next term, as well as in preparation for the 1B Database course and your general education, to consult the HSQLDB documentation over the holidays and read about `DROP TABLE` (i.e. delete a table and all its contents) and `DELETE` (remove zero or more rows). There are also many more advanced uses of the `SELECT` statement to retrieve and combine data stored in multiple tables.

1. Create a new package called `uk.ac.cam.crsid.fjava.tick5` and create a new class called `Database` inside this package.
2. Create a special `main` method inside the class `Database` and cut 'n' paste all the example code shown in this section of the workbook. (In other words, the code to connect to the database, turn on transaction support, create a new table called `messages`, add a row to the table, commit the transaction, query the database and print out the answer, and close all database connections.)
3. Add appropriate import statements to the top of your `Database` class.
4. Modify your implementation of `Database` so that it takes the filesystem path prefix to the database as the only command-line argument. If the main method is executed without any arguments, your program should print out the following error message to `System.err` and terminate:

```
Usage: java uk.ac.cam.crsid.fjava.tick5.Database <database name>
```

5. Run your program several times. What happens?

Database integration

The last section introduced a small subset of SQL and the associated Java language bindings. In this section you will modify your implementation of the Java chat server you wrote last week to make use of the database. Your database should store data in two tables: (1) the details of every message sent through the server should be recorded in a table called `messages` with exactly the same column definitions provided in the last section; and (2) a table called `statistics` which should have the following SQL definition:

```
CREATE TABLE statistics(key VARCHAR(255),value INT)
```

The `statistics` table should only *ever* have two rows, which must be initialised *only when the table is first created*. The initialisation is given in the following two lines of SQL:

```
INSERT INTO statistics(key,value) VALUES ('Total messages',0)
INSERT INTO statistics(key,value) VALUES ('Total logins',0)
```

Whenever a user logs in to the server, you should increment the count associated with the row recording the total number of logins as follows:

```
UPDATE statistics SET value = value+1 WHERE key='Total logins'
```

You should increment the count associated with the row recording the total number of messages whenever a new message is sent in similar fashion.

Rather than scatter the details of the database across multiple locations in your implementation of the Java chat server, you should enhance the definition of your `Database` class you wrote in the last section to provide a suitable abstraction. In particular, you should define the following fields and methods inside the class `Database`:

```

public class Database {
    private Connection connection;
    public Database(String databasePath) throws SQLException { ... }
    public void close() throws SQLException { ... }
    public void incrementLogins() throws SQLException { ... }
    public void addMessage(RelayMessage m) throws SQLException { ... }
    public List<RelayMessage> getRecent() throws SQLException { ... }
    public static void main(String []args) { /* leave as-is */ }
}

```

Please do not modify the contents of the main method—leave it exactly as specified in the previous section. The implementation details of the remaining methods and field are as follows:

- The class `Database` has a single constructor which takes a string describing the filesystem path prefix to the database on disk. The constructor should load the HSQLDB driver and initialise the field `connection` with a connection to the database; you should also create the database tables if they don't already exist.
- The method `close` should do (almost) the inverse of the constructor, namely call the `close` method on `connection`.
- The `incrementLogins` method should use the reference held in the field `connection` to update the appropriate value stored in the `statistics` table. Don't forget to call `commit`!
- The `addMessage` method should add the contents of the `RelayMessage` object `m` to the `messages` table *and* increment the appropriate value stored in the `statistics` table. Make sure you do both these updates as part of *one* transaction so that concurrent execution of this method is thread-safe. (Thread-safety is essential so that later on, when this method is invoked by instances of `ClientHandler`, data in the `statistics` table are correctly recorded.)
- The method `getRecent` should retrieve the top ten most recent messages from the `messages` table, and copy them into a class which implements the `java.util.List` interface.

6. Complete your implementation of `Database` as specified above.

7. Copy across the following interfaces and classes from your Ticklet 4 submission into the same package as `Database`: `ChatServer`, `ClientHandler`, `MessageQueue`, `MultiQueue` and `SafeMessageQueue`.

Your final task this week is to integrate your implementation of `Database` so that it is used by your implementation of the Java chat server. To do so, you will need to do the following:

- Create a new field called `database` of type `Database` inside the `ClientHandler` class. Modify the constructor to the `ClientHandler` class to accept a reference to a `Database` object as the third argument and update `database` in the constructor to reference it.
- Modify the `main` method in `ChatServer` to accept two arguments on the command line: the port number for the service, and the filesystem path prefix to the database. Your implementation of the `main` method of `ChatServer` should then create an instance of `Database` and pass a reference to this into the constructor of `ClientHandler`.
- Modify your implementation of `ClientHandler` so that when a new client connects, it receives up to ten objects of type `RelayMessage` immediately which represent the ten most recent messages stored in the `messages` table in the database. (Hint: call the method `getRecent` on the field `database`.)
- Whenever a new user connects to the server, a suitable part of the `ClientHandler` class should call the method `incrementLogins` on the field `database`.

- Whenever a user sends a serialised instance of `ChatMessage` to the server, modify your implementation of `ClientHandler` to add the message to the database by calling `addMessage` on the field database.

8. Complete the necessary modifications to your Java chat server to ensure that messages are recorded and appropriate statistics are stored in the database. When new clients connect, they should receive the ten most recent messages sent to the server.

Ticklet 5

You have now completed all the necessary code to gain your fifth ticklet. Please generate a jar file which contains all the code you have written for package `uk.ac.cam.crsid.fjava.tick5` together with the code you downloaded and imported in package `uk.ac.cam.cl.fjava.messages`. Please use Eclipse to export both the class files *and the source files* into a jar file called `crsid-tick5.jar`. Once you have generated your jar file, check that it contains at least the following classes:

```
crsid@machine~:> jar tf crsid-tick5.jar
META-INF/MANIFEST.MF
uk/ac/cam/crsid/fjava/tick5/ChatServer.java
uk/ac/cam/crsid/fjava/tick5/ChatServer.class
uk/ac/cam/crsid/fjava/tick5/ClientHandler.java
uk/ac/cam/crsid/fjava/tick5/ClientHandler.class
uk/ac/cam/crsid/fjava/tick5/Database.java
uk/ac/cam/crsid/fjava/tick5/Database.class
uk/ac/cam/crsid/fjava/tick5/MessageQueue.java
uk/ac/cam/crsid/fjava/tick5/MessageQueue.class
uk/ac/cam/crsid/fjava/tick5/MultiQueue.java
uk/ac/cam/crsid/fjava/tick5/MultiQueue.class
uk/ac/cam/crsid/fjava/tick5/SafeMessageQueue.java
uk/ac/cam/crsid/fjava/tick5/SafeMessageQueue.class
uk/ac/cam/cl/fjava/messages/ChangeNickMessage.class
uk/ac/cam/cl/fjava/messages/ChangeNickMessage.java
uk/ac/cam/cl/fjava/messages/ChatMessage.class
uk/ac/cam/cl/fjava/messages/ChatMessage.java
uk/ac/cam/cl/fjava/messages/NewMessageType.class
uk/ac/cam/cl/fjava/messages/NewMessageType.java
uk/ac/cam/cl/fjava/messages/Message.class
uk/ac/cam/cl/fjava/messages/Message.java
uk/ac/cam/cl/fjava/messages/RelayMessage.class
uk/ac/cam/cl/fjava/messages/RelayMessage.java
uk/ac/cam/cl/fjava/messages/StatusMessage.class
uk/ac/cam/cl/fjava/messages/StatusMessage.java
crsid@machine~:>
```

When you are satisfied you have built the jar correctly, you should submit your jar file as an email attachment to `ticks1b-java@cl.cam.ac.uk`.

You should receive an email in response to your submission. The contents of the email will contain the output from a program (written in Java!) which checks whether your jar file contains all the relevant files, and whether your program has run successfully or not. If your jar file does not pass the automated checks, then the response email will tell you what has gone wrong; in this case you should correct any errors in your work and resubmit your jar file. You can resubmit as many times as you like and there is no penalty for re-submission. If, after waiting one hour, you have not received any response you should notify `ticks1b-admin@cl.cam.ac.uk` of the problem. You should submit a jar file which successfully passes the automated checks by the deadline, so don't leave it to the last minute!