

JDBC and the DAO Layer

The story so far –

- we've defined the API via the REST endpoints

Now –

- we start at the bottom layer of the application's code
 - JDBC – how Java connects to MySQL and runs SQL
 - DAO (data access object) layer – wraps the SQL for each entity so the rest of the code works only with Java objects

JDBC

- your application and MySQL are separate programs
 - every query is a message over a connection
- JDBC combines two elements
 - a vendor-neutral Java API (java.sql)
 - a vendor-specific driver (MySQL Connector/J)
 - only need to swap the driver to connect to a different database product

Connecting – Preliminaries

- connection URL

```
jdbc:mysql://bridgeman-vm.hws.edu:3306/demo_library
```

└─ protocol: JDBC

└─ subprotocol: which driver

└─ host

└─ port (MySQL default 3306)

└─ database (schema) name

- also need to set up the driver
 - download Connector/J
 - add the JAR file to your project's build path
 - missing driver results in (only) runtime errors

Connecting – Code

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class ConnectionCheck {
    public static void main(String[] args) {
        String url = "jdbc:mysql://bridgeman-vm.hws.edu:3306/demo_library";

        try (Connection conn =
            DriverManager.getConnection(url, "demo_user",
                                       "demo_password")) {
            DatabaseMetaData meta = conn.getMetaData();
            System.out.println("Connected to " + meta.getDatabaseProductName()
                               + " " + meta.getDatabaseProductVersion());
            System.out.println("Using driver: " + meta.getDriverName());
        } catch (SQLException e) {
            System.out.println("Failed: " + e.getMessage());
        }
    }
}
```

Managing DB Credentials

- credentials should come from the environment
 - never hardcode or commit credentials!
 - in Eclipse, set on the Environment tab of the run configuration

```
public final class DB {
    private static final String URL = System.getenv("DB_URL");
    private static final String USER = System.getenv("DB_USER");
    private static final String PASSWORD = System.getenv("DB_PASSWORD");

    public static Connection getConnection() throws SQLException {
        return DriverManager.getConnection(URL, USER, PASSWORD);
    }
}

public class ConnectionCheck {
    public static void main(String[] args) {
        try (Connection conn = DB.getConnection()) {
            ...
        } catch (SQLException e) {
            System.out.println("Failed: " + e.getMessage());
        }
    }
}
```

Executing Statements

```
String sql = "SELECT book_id, title, author FROM BOOKS WHERE book_id = ?";
PreparedStatement ps = conn.prepareStatement(sql);
ps.setInt(1, 42);
ResultSet rs = ps.executeQuery();    // SELECT → rows
```

```
String sql = "UPDATE BOOKS SET available = FALSE WHERE book_id = ?";
PreparedStatement ps = conn.prepareStatement(sql);
ps.setInt(1, 42);
int changed = ps.executeUpdate();    // INSERT/UPDATE/DELETE → row count
```

- two execute methods, depending on whether or not the statement returns rows
 - `executeQuery()` returns a `ResultSet`
 - `executeUpdate()` returns the number of rows changed
 - a row count of 0 on an update or delete means no such row
- not yet handled – closing resources

Managing Connections

- establishing a connection is expensive
 - production code reuses connections from a pool

SQL Injection and Placeholders

- never concatenate unchecked user input into a query



```
String author = "Robert'); DROP TABLE Students; --";
String sql = "(SELECT * FROM BOOKS WHERE author = '" + author + "')";
```

sql now has the value
(SELECT * FROM BOOKS WHERE author = 'Robert'); DROP TABLE Students; --')

- `PreparedStatement`s provide placeholders for values
 - prevents *SQL injection* – SQL and values are kept separate, so a value can't be read as SQL

```
String sql = "SELECT * FROM BOOKS WHERE author = ?";
PreparedStatement ps = conn.prepareStatement(sql);
ps.setString(1, author);
```

Accessing the Results

- a `ResultSet` is a *cursor* – think of a finger starting before the first row
 - each `next()` moves the finger forward to the next row, returning `false` when there are no more rows
 - getters retrieve values from the current row
- one getter per type – `getInt`, `getString`, `getBoolean`, ...
 - column name specified as a parameter

```
String sql = "SELECT book_id, title, author, available FROM BOOKS ORDER BY title";
PreparedStatement ps = conn.prepareStatement(sql);
ResultSet rs = ps.executeQuery();
while (rs.next()) {
    int id = rs.getInt("book_id"); // move to the next row; false when done
    String title = rs.getString("title");
    String author = rs.getString("author");
    boolean available = rs.getBoolean("available");
    System.out.println(id + ": " + title + " by " + author
        + (available ? "" : " (checked out)");
}
```

Accessing the Results

- if at most one row is expected, can use `if` instead of `while`

```
if ( rs.next() ) {
    ...
} else {
    ...
}
```