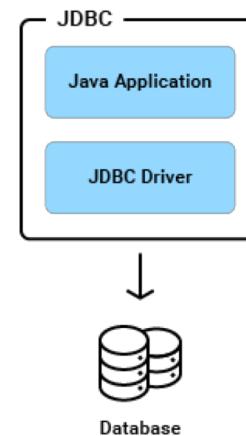


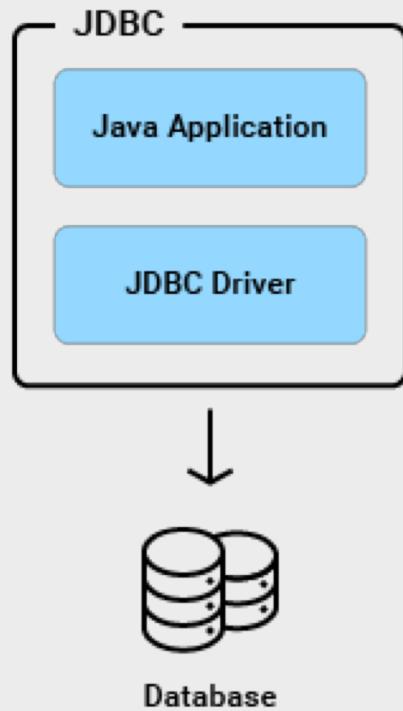
Web-based information systems 1

Database interaction
and
Java software design

Michiel Noback (NOMI)
Institute for Life Sciences and Technology
Hanze University of Applied Sciences



introduction



- Database interaction with Java applications: JDBC
- Java design issues that ensure flexible, reusable code that will easily deal with changing specs:
 - a different database provider
 - a different view
 - different analysis components
 - etc etc
- Just remember: the only constant in programming is change

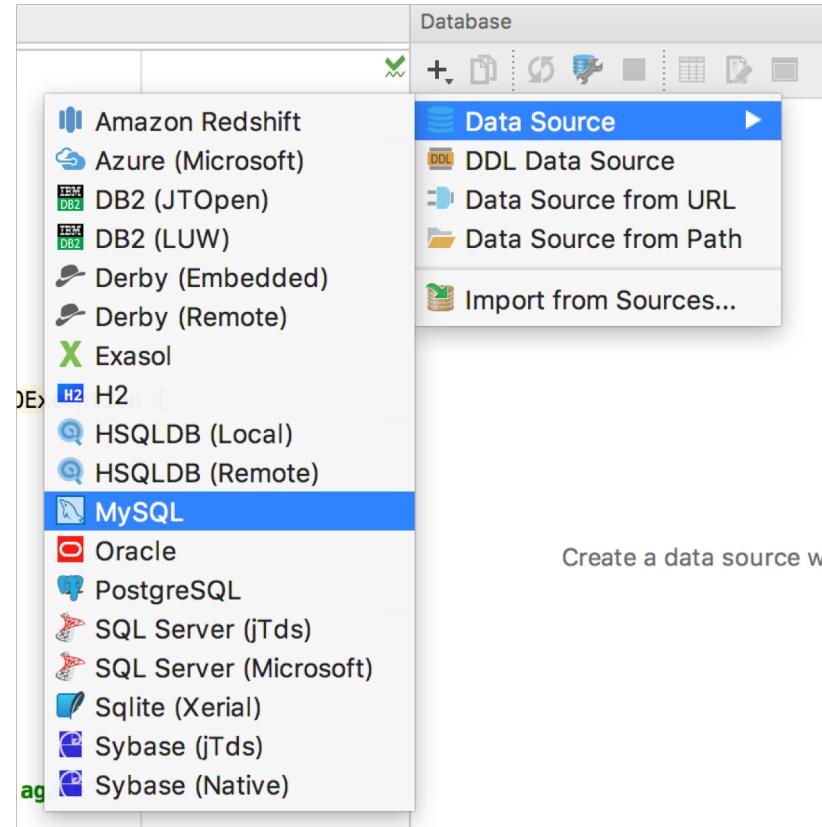
JDBC

- Let's start with the simple thing: how to communicate with a database using Java.
- Suppose we have a database called MyDB with this single, simple table Users

Field	Type
user_id	int (auto_inc)
user_name	varchar(30)
user_password	varchar(30)
user_email	varchar(30)
role	enum('GUEST', 'USER', 'ADMIN')

Database actions within IntelliJ

- To get the Database toolbox:
 - View -> Tool Windows -> Database
- Click “+” -> Data Source -> MySQL



Fill in the blanks

- Click “download missing driver files”
- Click “test connection”

The screenshot shows the 'General' tab of a MySQL Workbench connection configuration dialog. The 'Name' field contains 'useradmin@localhost'. The 'Comment' field is empty. The 'Host' field is set to 'localhost' and the 'Port' field is set to '3306'. The 'Database' field is set to 'useradmin'. The 'User' field is set to 'michiel'. The 'Password' field is obscured by black dots. A checked checkbox labeled 'Remember password' is present. The 'URL' field displays 'jdbc:mysql://localhost:3306/useradmin' with an 'Overrides settings above' note below it. A dropdown menu next to the URL field shows 'default'. At the bottom, there are 'Test Connection' and 'Successful Details' buttons, and a 'Driver' field set to 'MySQL'.

Name: useradmin@localhost Reset

Comment:

General SSH/SSL Schemas Options Advanced

Host: localhost Port: 3306

Database: useradmin

User: michiel

Password: [REDACTED] Remember password

URL: jdbc:mysql://localhost:3306/useradmin Overrides settings above default

Test Connection Successful Details

Driver: MySQL

MySQL within IntelliJ

- Open the console,
 - Enter a command
 - Press Ctrl + enter

The screenshot shows a database session titled 'useradmin@localhost' with the following details:

- Session Path: Database Consoles > useradmin@localhost > useradmin@localhost
- Tab Bar: Based1_2017, web.xml, LoginServlet.java, useradmin@localhost (active)
- Toolbar: Tx: Auto, Undo, Redo, Copy, Paste
- SQL Editor:

```
SELECT * FROM U
  UNION ALL
    SELECT 1 AS ID, 'useradmin' AS NAME, 'useradmin' AS PASSWD
    FROM DUAL
    FOR UPDATE
    GROUP BY ID, NAME, PASSWD
```
- Results Grid:

ID	NAME	PASSWD
1	useradmin	useradmin

Add some data

```
DROP TABLE IF EXISTS Users;

CREATE TABLE Users (
    user_id INT NOT NULL auto_increment,
    user_name VARCHAR(100) NOT NULL,
    user_password VARCHAR(100) NOT NULL,
    user_email VARCHAR(255) NOT NULL,
    user_role VARCHAR(100) NOT NULL,
    primary key(user_id)
);

INSERT INTO Users (user_name, user_password, user_email)
VALUES ('Henk', 'Henkie', 'Henk@example.com', 'ADMIN');

#SELECT * FROM Users;
```

IntelliJ Database tool window

The screenshot shows the IntelliJ Database tool window interface. The title bar says "Database". Below the title bar is a toolbar with several icons: a plus sign, a save icon, a circular arrow, a wrench, a gray square, a table icon, a pencil icon, and a refresh icon.

The main area displays a database structure:

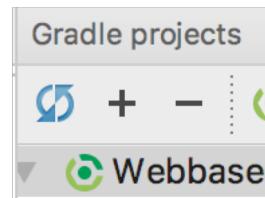
- A connection node for **useradmin@localhost** (1 of 5).
- Under "schemas":
 - schemas** (1 item)
 - useradmin** (1 item)
 - Users** (5 items)
 - user_id int(11) (auto increment)**
 - user_name varchar(255)**
 - user_password varchar(255)**
 - user_email varchar(255)**
 - PRIMARY (user_id)**
- collations** (222 items)

Add project dependency

- In build.gradle, add

```
// https://mvnrepository.com/artifact/mysql/mysql-connector-java  
compile group: 'mysql', name: 'mysql-connector-java', version: '5.1.6'
```

- You may need to refresh your Gradle project in the Gradle Tool window



Code against interfaces, not implementations

- This is especially relevant for database access!
- Change of database implementation...

```
public interface MyAppDao{  
    public void connect();  
    public User getUser(String uName, String uPass);  
    public void insertUser(String uName, String uPass, String eMail);  
    public void disconnect();  
}
```

(1) define the interface methods

```
class MyDbConnector implements MyAppDao{ ... }
```

```
public class MyApp {  
    private connectDB(){  
        MyAppDao dao = MyDbConnector.getInstance();  
        dao.connect();  
    }  
}
```

(2) implement the interface

(3) write code agianst the interface type

JDBC class loading

- Like all programming languages, you need drivers. You can load them like this in Java:

```
import java.sql.*;  
class MyDbConnector implements MyAppDao {  
  
    public void connect(){  
        Class.forName("com.mysql.jdbc.Driver");  
    }  
}
```

Of course you will need to import the correct classes/packages

This mechanism is called **dynamic class loading**

This is how you load the JDBC driver

JDBC creating a Connection

- After loading the driver class(es), you will need to establish a connection:

```
class MySqlConnector{  
    Connection connection;  
    String dbUrl = "jdbc:mysql://bioinf.nl/MyDB";  
    String dbUser = "Fred";  
    String dbPass = "Pass";  
  
    public MySqlConnector(){  
        connectDb();  
    }  
  
    public void connectDB(){  
        Class.forName("com.mysql.jdbc.Driver");  
        try{  
            connection = DriverManager.getConnection(dbUrl,dbUser,dbPass);  
        }catch(Exception e){e.printStackTrace();}  
    }  
}
```

These are the required objects. The actual connection is of course a Connection object

Here you establish the connection. Talking to a DB is risky business, so yo'have to put it in a try/catch block

JDBC filling the database

- Now you are ready to put some data in the database:

```
public void insertUser(String name, String pass, String eMail) {  
    try {  
        String insertQuery =  
            "INSERT INTO Users (user_name, user_password, user_email) "  
            + " VALUES (?, ?, ?)";  
        PreparedStatement ps = connection.prepareStatement(insertQuery);  
        ps.setString(1, name);  
        ps.setString(2, pass);  
        ps.setString(3, eMail);  
        ps.executeUpdate();  
        ps.close();  
    } catch (SQLException e) {  
        e.printStackTrace();  
    }  
}
```

JDBC getting data out of the database

- Now the other way around: get data out of the database:

```
public void getUser(String name, String pass ){  
    try {  
        String fetchQuery =  
            "SELECT * FROM Users WHERE user_name = ? AND user_password = ?";  
        PreparedStatement ps = connection.prepareStatement(fetchQuery);  
        ps.setString(1, name);  
        ps.setString(2, pass);  
        ResultSet rs = ps.executeQuery();  
        while (rs.next()) {  
            String userMail = rs.getString("user_email");  
            String userIdStr = rs.getString("user_id");  
            System.out.println("userMail = " + userMail);  
            System.out.println("userIdStr = " + userIdStr);  
        }  
        ps.close();  
    } catch (SQLException e) ... }
```

JDBC resource efficiency

- `PreparedStatement`, `Connections` and `ResultSets` are expensive objects in terms of computer resources; you should instantiate them sparingly.
- Here follow some code constructs to do this:
- First: the **Singleton Pattern**. This is an object-oriented design pattern that is used to ensure there is always only one object of a particular class alive in an application.

Singleton Pattern

```
class MySqlConnector{  
    private static MySqlConnector uniqueInstance;  
  
    private MySqlConnector(){}
  
  
    public static getInstance(){  
        if (uniqueInstance == null ){
            uniqueInstance = new MySqlConnector();
        }
        return uniqueInstance;
    }
}
```

These are the key ingredients to implementing the singleton pattern:

- (1) have a static variable of the one instance
- (2) make the constructor private
- (3) provide a single access point to the instance and create it on first request

Cashing PreparedStatements

- PreparedStatements should be reused as much as possible:

```
@Override
public void connect() throws DatabaseException {
    //connection code omitted
    prepareStatements();
}
/**Prepares statements and stores them for reuse*/
private void prepareStatements() throws SQLException {
    String fetchQuery = "SELECT * FROM Users WHERE user_name = ?
                           AND user_password = ?";
    PreparedStatement ps = connection.prepareStatement(fetchQuery);
    this.preparedStatements.put("GET_USER", ps);

    String insertQuery = "INSERT INTO Users (user_name, user_password,
                           user_email, user_role) VALUES (?, ?, ?, ?)";
    ps = connection.prepareStatement(insertQuery);
    this.preparedStatements.put("INSERT_USER", ps);
}
```

Using PreparedStatements

```
private HashMap<String, PreparedStatement> pStmts;
void getUser(String name, String pass ){
    try{
        PreparedStatement ps = pStmts.get("fetch_user");
        preparedStatement.setString(1, "JohnDoe");
        preparedStatement.setString(2, "mypass");
        ResultSet rs = preparedStatement.executeQuery();
        while (resultSet.next()) {
            String userMail = resultSet.getString("user_email");
            String userIdStr = resultSet.getString("user_id");
        }
    }catch( Exception e ){ ... }
}
```

Cleaning up PreparedStatements

- After you are finished, you should clean up your resources
- In a web app, this happens at shutdown time (there are hooks for this!)

```
private HashMap<String, PreparedStatement> pStmts;
void disconnect() {
    try {
        for (String key : pStmts.keySet()) {
            pStmts.get(key).close();
        }
    } catch(Exception e) {
        e.printStackTrace();
    } finally {
        connection.close();
    }
}
```

Data access summary

- ALWAYS code data access logic against an interface
- Restrict use of database-specific code to dedicated classes
- Implement the singleton pattern for data access objects
- Keep use of data access resources to a minimum
- Do not forget to close prepared statements and connections

The final picture

The model can now communicate safely with a data source that may change in the future

The view (Thymeleaf) takes the prepared data (beans, lists and maps preferably) and displays them

