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

The topics of this presentation are

- 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

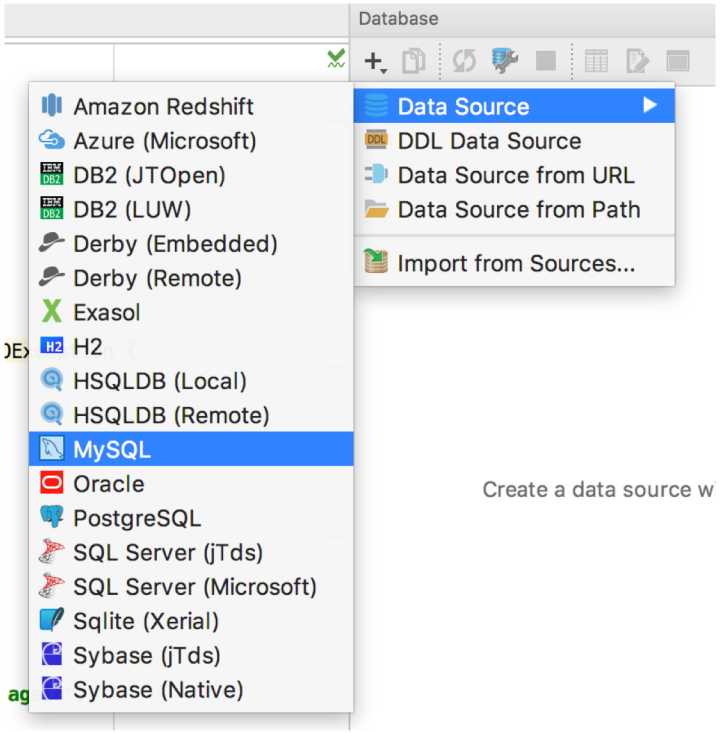
JDBC

- 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)

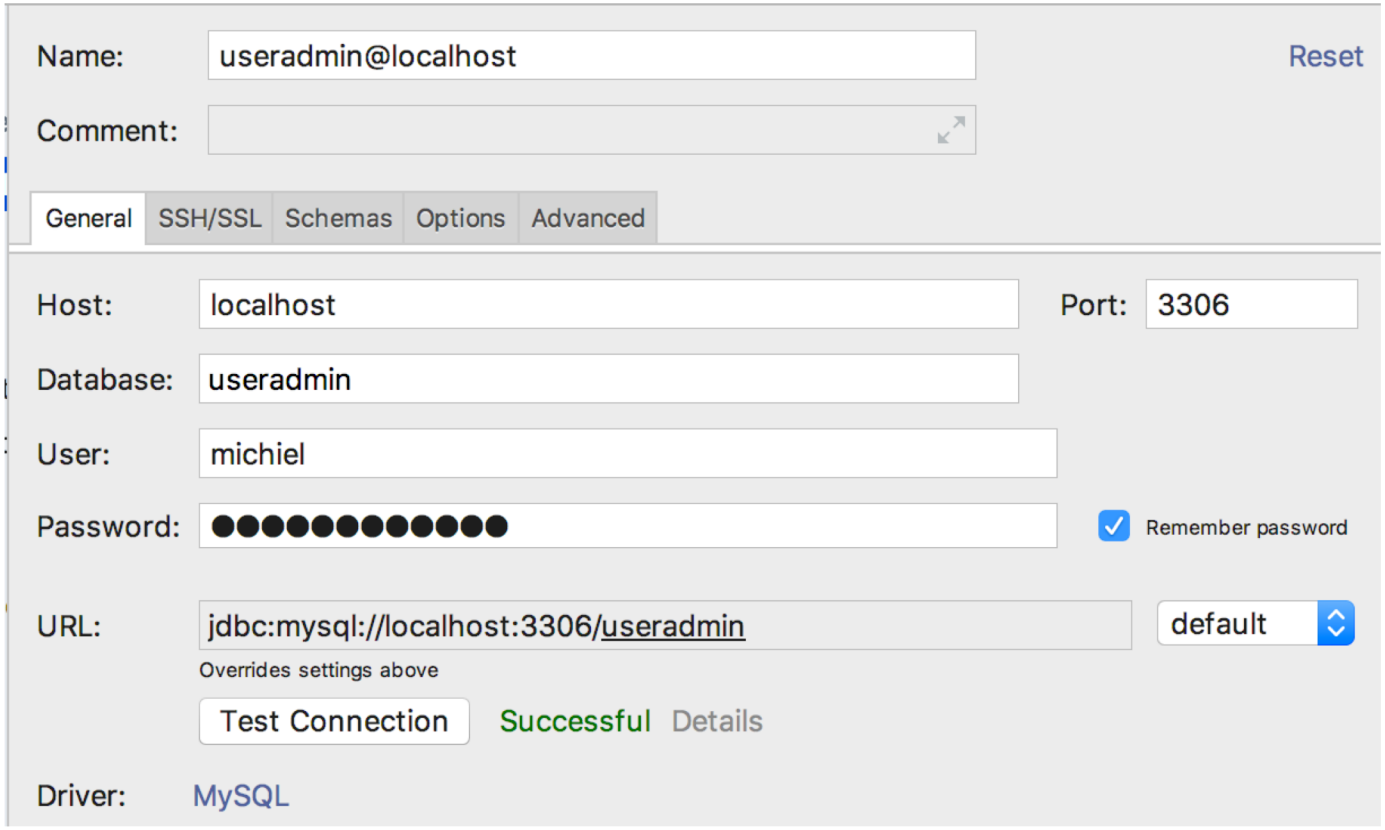
Database actions within IntelliJ

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



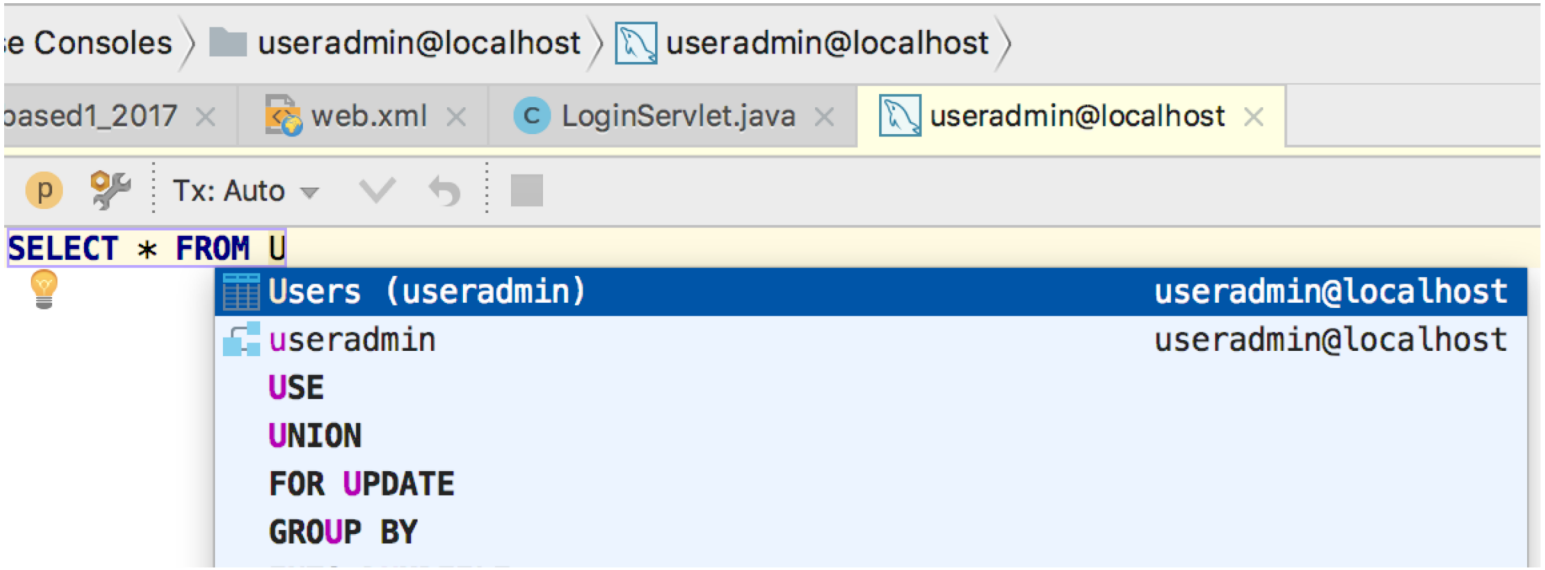
Fill in the blanks

- Click "download missing driver files
- Click "test connection"



MySQL within IntelliJ

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



Add some data

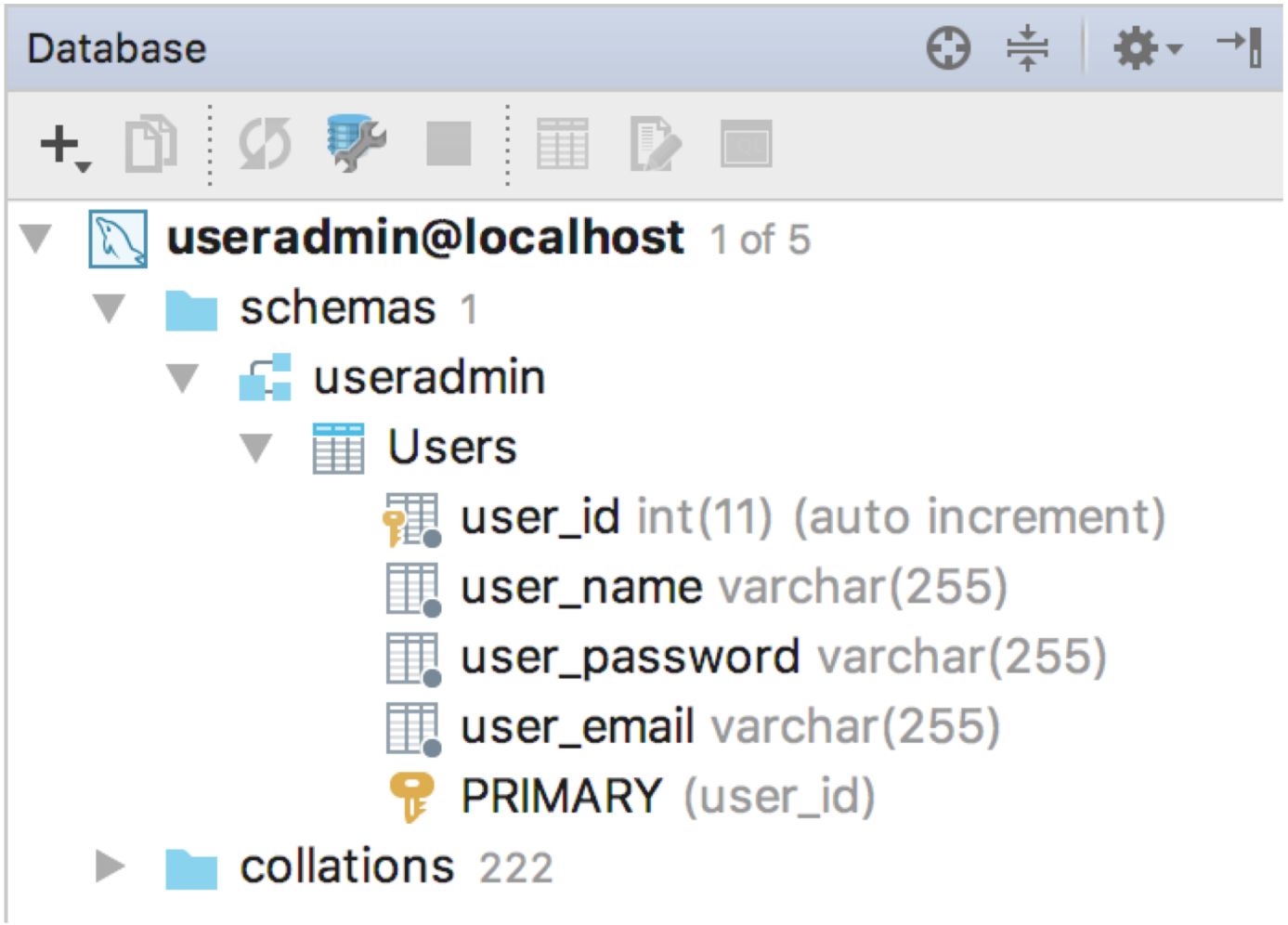
```
DROP TABLE IF EXISTS Users;

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

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

#SELECT * FROM Users;
```

IntelliJ Database tool window

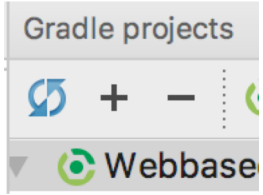


Add project dependency

- 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



Accessing through an interface

- You should always code against interfaces as much as possible
- This is especially relevant for database access; what if you decide to change your 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

(2) implement the interface

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

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

(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:

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

```
import java.sql.*;
class MyDbConnector implements MyAppDao {

  public void connect(){
    Class.forName("com.mysql.jdbc.Driver");
  }
}
```

This is how you load the JDBC driver

This mechanism is called **dynamic class loading**

JDBC creating a Connection

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

```
class MyDbConnector{
  Connection connection;
  String dbUrl = "jdbc:mysql://bioinf.nl/MyDB";
  String dbUser = "Fred";
  String dbPass = "Pass";

  public MyDbConnector(){
    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

- PreparedStatements, Connections and ResultSets are expensive objects in terms of computer resources; you should instantiate them sparingly.
- This also ensures the most efficient program execution
- Here are 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.
- You should always implement your database access object in this manner

Singleton Pattern

- PreparedStatements, Connections and ResultSets are expensive objects in terms of computer resources; you should instantiate them sparingly

```
class MyDbConnector{
  private static MyDbConnector uniqueInstance;

  private MyDbConnector(){}

  public static getInstance(){
    if (uniqueInstance == null ){
      uniqueInstance = new MyDbConnector();
    }
    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

PreparedStatements (1)

- PreparedStatements should be reused as much as possible: this is efficient in resources and program run time:

```
  private HashMap<String, PreparedStatements> pStmts;
  private MyDbConnector(){
    initStatements();
  }

  private initStatements(){
    String iQuery = "INSERT INTO Users
        (user_name, user_password, user_email) VALUES (?, ?, ?)";
    PreparedStatement ps = connection.prepareStatement( iQuery );
    pStmts.put("insert_user", ps);
    String fQuery = "SELECT * FROM Users WHERE user_name=?
                     AND user_password=?";
    ps = connection.prepareStatement( fQuery );
    pStmts.put("fetch_user", ps);
  }
}
```

PreparedStatements (2)

```
private HashMap<String, PreparedStatements> pStmts;
void getUser(String name, String pass ){
  try{
    PreparedStatement ps = pStmts.get("fetch_user");
    assert ps != null;
    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 ){ ... }
} }
```

PreparedStatements (3)

- After you are finished, you should really clean up your resources:

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

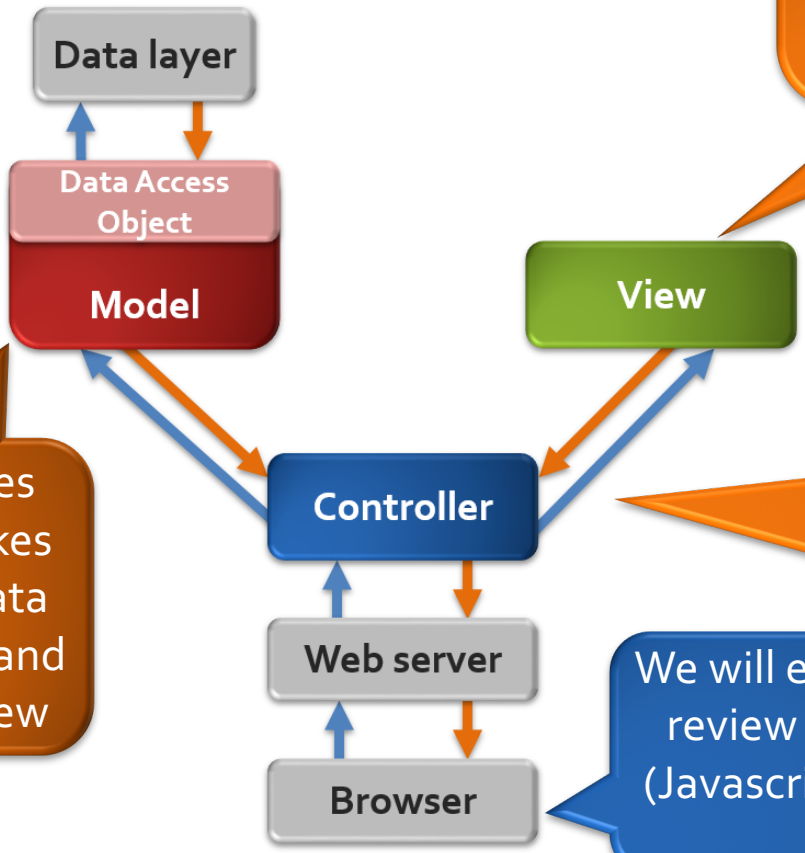
Data access summary

- ALWAYS code data access logic against an interface
- Restrict use of database-specific code to one class
- 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

Putting it all together

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

The view (JSP) takes the prepared data (beans, lists and maps preferably) and displays them without using code



The model processes the request and makes sure the returned data are readily available and accessible for the view

Now the Servlet is responsible for handling requests and invoking correct parts of the model

We will end this course with a short review of client-side technology (Javascript, Ajax, css) for your web application