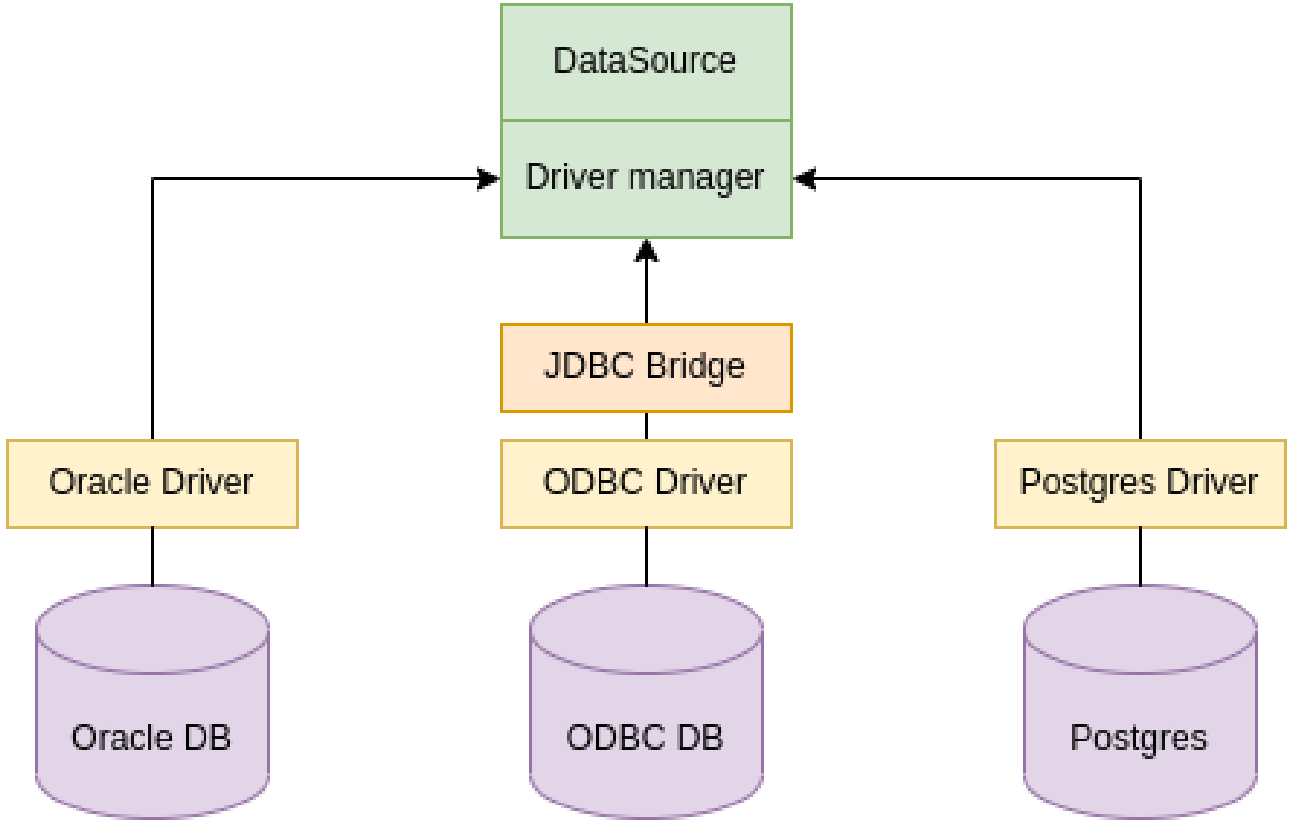
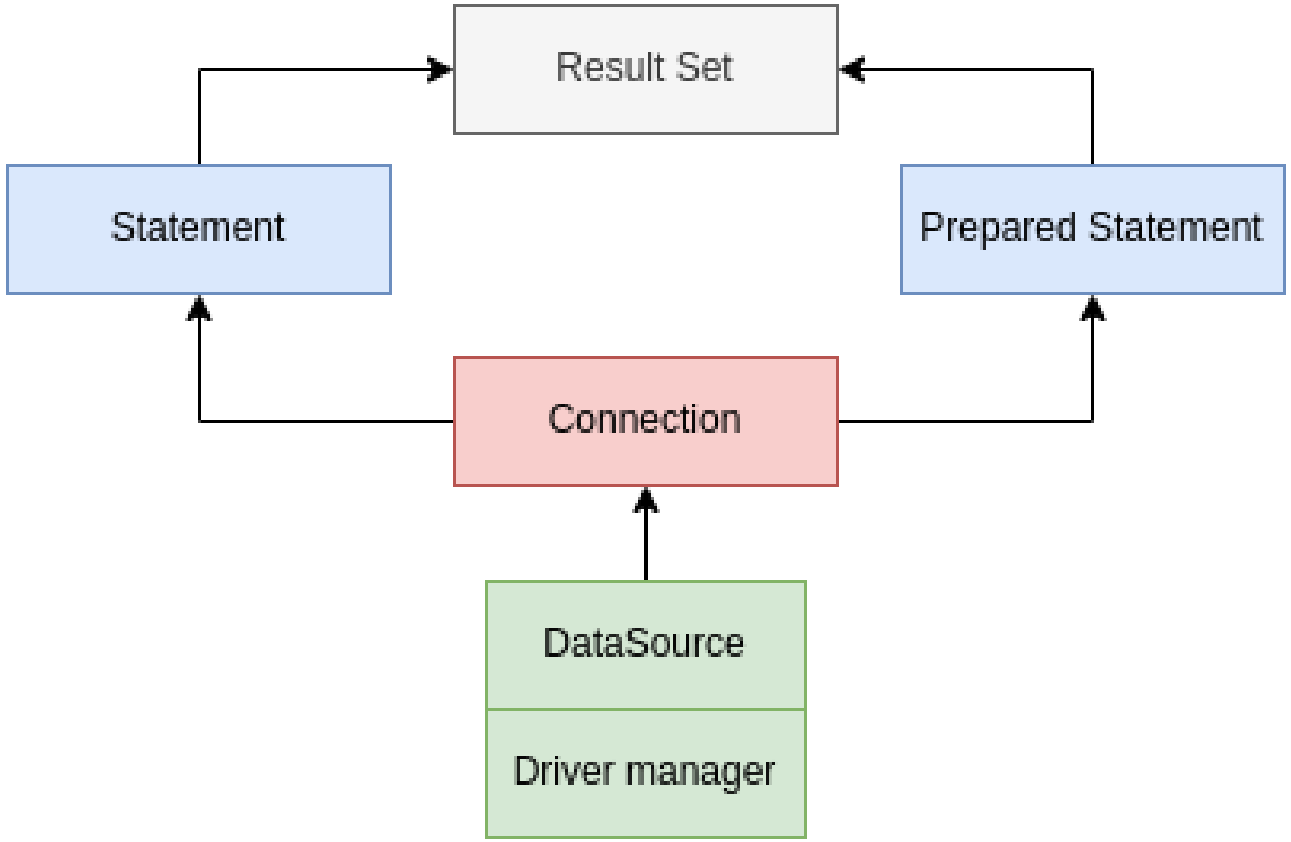
PV168

Databases II

JDBC: Architecture



JDBC API


Result Set
Statement
Prepared Statement
Connection
DataSource
Driver manager

JDBC Connection

- Transaction per statement execution
- Can be disabled by `con.setAutoCommit(false)`
- Without auto commit `con.commit()` / `con.rollback()` is required
- Rollback on error
- Must be closed

JDBC Connection Pooling

- Creating connection is expensive
- You don't always need a new physical connection
- Reuse physical connections across multiple logical uses
- Ability to setup the size of the connection pool
- H2 `JdbcConnectionPool` (simple implementation) implements `DataSource` interface (like most connection pool implementations)

Code Sample

Connection pool example

```
String jdbcUri = "jdbc:h2:/db/path;DATABASE_TO_UPPER=false";
String userName = "databaseUsername"
String password = "databasePassword"
DataSource dataSource = JdbcConnectionPool.create(jdbcUri, username, password);

// After the {Connection#close()} is called,
// it is returned to the connection pool
try(Connection connection = dataSource.getConnection()) {
   // Perform the operations
}
```

JDBC (Prepared) Statement

- Don't use `Statement` !!!
- `PreparedStatement` protects against SQL Injection
- Closed when connection object is closed
- Must be closed explicitely anyway, why?

JDBC ResultSet

- Represents result of database
- Iterator-like object over rows in result relation
- Should be closed (closed with `PreparedStatement`)

Code Sample

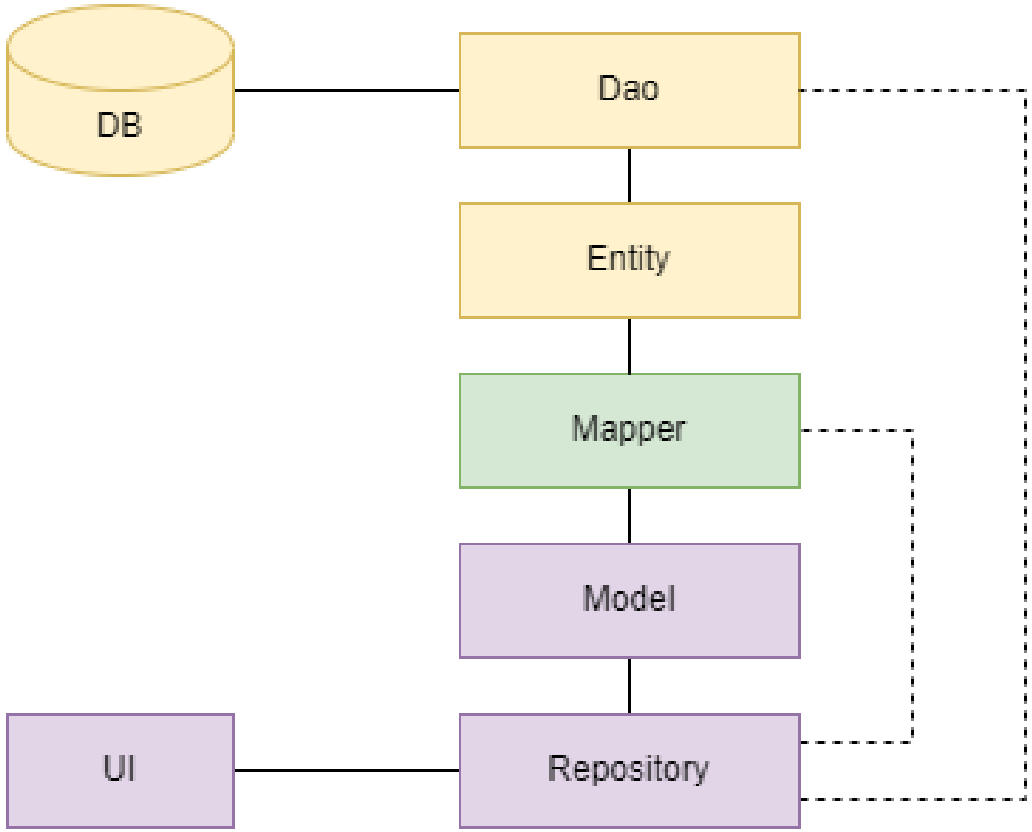
```
// connection and statement should be both closed after we are done
try(
    Connection connection = dataSource.getConnection();
    var statement = connection.prepareStatement("SELECT id, number FROM Department WHERE id = ?")
) {
    ps.setLong(1, departmentID); // set the
    try (var rs = statement.executeQuery()) { // we should close the result set
        while (rs.next()) {
            long departmentId = rs.getLong("id");
            String number = rs.getString("number");
        }
    }
}
```

Code Sample: Transaction

```
try(Connection connection = dataSource.getConnection()) {
  connection.setAutoCommit(false);
  try(var st = connection.prepareStatement("INSERT INTO ...")) {
      st.executeUpdate();
  }
  ...
  if(isSomeError)
    connection.rollback(); // maunal rollback
  ...
  try(var st = connection.prepareStatement("UPDATE ...")) {
    st.executeUpdate();
  }
   connection.commit(); // manual commit
}
```

Employee Records Architecture

Layers Diagram



Data Layer

- Responsible for storing data in Database
- Data represented by Entity object
- Entities managed via Data Access Objects
- In our project, DAOs receive a supplier for `ConnectionHandler`
  - connection handlers provides a connection instance
  - this will be useful when we will be working with transactions

Business / Model Layer

- Representation of data according to business reqirements
- Business object can separate / aggregate entity data
- Data represented by Model Objects
- Model objects managed via Repositories

Connecting Data and Business

- In ideal case - only repositories can delegate to DAO
- Mapping between Entities and Model Objects
- Done by mappers

Why such separation?

- Data Presentation vs Data Storage

Validation

- In our project the mapper is validating the data
  - Validation is always done when mapping *Model* to *Entity*

Database specific validation:

- Better to validate before storing data to the database
- `VARCHAR` and other database types have fixed size
- Example: `Department.number` max length is 10 characters

Seminar Reflection

- What was problematic?
- Possible solution...

Enum Representation

- Database dependent
- Possible representations:
  - Enums:
    - `gender ENUM('male', 'female')`
  - `VARCHAR` and check if database supports it
  - Enum values as a separate table

Application State Initialization

Task 3: Departments are missing in the application

- Multiple enviroments: Production, Development, Test
- Each environment should have different prepared data

Possible solutions (not all):

- Data migrations - data directly to the database (run `data_dev.sql`)
- Data initialization using code `DataInit` `~>` `ProductionDataInit`
- Import - data can be imported (next week)

Problems with Data Initialization

- When to run the initialization?
- How to initialize the data?
  - Separate command/action (explicit)
  - At start, when database is empty (implicit)
  - Special property/environment variable (explicit)