JAVA II

JDBC, JDBC templates, Spring Data JPA

The Java Database Connectivity (JDBC)

The Java Database Connectivity (JDBC) API is the industry standard for database-independent connectivity between the Java programming language and a wide range of databases .

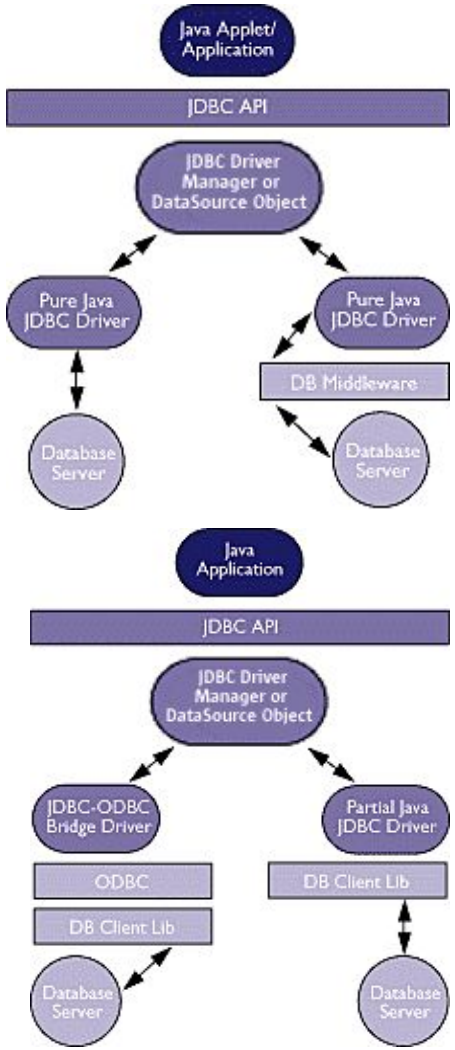
The JDBC API makes it possible to do three things:

- Establish a connection with a database or access any tabular data source;
- Send SQL statements;
- Process the results;

JDBC Architecture

The JDBC API contains two major sets of interfaces: the first is the JDBC API for application writers, and the second is the lower-level JDBC driver API for driver writers. JDBC technology drivers fit into one of four categories:

- Direct-to-Database Pure Java Driver;
- Pure Java Driver for Database Middleware;
- JDBC-ODBC Bridge plus ODBC Driver;
- A native API partly Java technology-enabled driver



JDBC functionality

JDBC support creating and executing statements. These may be update statements such as SQL's CREATE, INSERT, UPDATE and DELETE, or they may be query statements such as SELECT. Additionally, stored procedures may be invoked through a JDBC connection. JDBC represents statements using one of the following classes:

- Statement – the statement is sent to the database server each and every time.
- PreparedStatement – the statement is cached and then the execution path is pre-determined on the database server allowing it to be executed multiple times in an efficient manner.
- CallableStatement – used for executing stored procedures on the database.
- Update statements such as INSERT, UPDATE and DELETE return an update count that indicates how many rows were affected in the database. These statements do not return any other information.

Query statements return a JDBC row result set. The row result set is used to walk over the result set. Individual columns in a row are retrieved either by name or by column number. There may be any number of rows in the result set. The row result set has metadata that describes the names of the columns and their types.

JDBC connections are often managed via a connection pool rather than obtained directly from the driver.

Databases types to Java types

Host database types which Java can convert to with a function

Oracle Datatype	`setXXX()` Methods
CHAR	`setString()`
VARCHAR2	`setString()`
NUMBER	`setBigDecimal()`
	`setBoolean()`
	`setByte()`
	`setShort()`
	`setInt()`
	`setLong()`
	`setFloat()`
	`setDouble()`
INTEGER	`setInt()`
FLOAT	`setDouble()`
CLOB	`setClob()`
BLOB	`setBlob()`
RAW	`setBytes()`
LONGRAW	`setBytes()`
DATE	`setDate()`
	`setTime()`
	`setTimestamp()`

Establishing a Connection

- DriverManager: This fully implemented class connects an application to a data source, which is specified by a database URL. When this class first attempts to establish a connection, it automatically loads any JDBC 4.0 drivers found within the class path.
- DataSource: This interface is preferred over DriverManager because it allows details about the underlying data source to be transparent to your application. (Typically by using JNDI (Java Naming and Directory Interface))

Establishing a Connection (MySQL JDBC Url)

MySQL Connector/J Database URL

The following is the database connection URL syntax for MySQL Connector/J:

```
jdbc:mysql://[host][:port]/[database][?propertyName1][=propertyValue1][&propertyName2][=propertyValue2]...
```

- *host*:*port* is the host name and port number of the computer hosting your database. If not specified, the default values of *host* and *port* are 127.0.0.1 and 3306, respectively.
- *database* is the name of the database to connect to. If not specified, a connection is made with no default database.
- *propertyName*=*propertyValue* represents an optional, ampersand-separated list of properties. These attributes enable you to instruct MySQL Connector/J to perform various tasks.

Establishing a Connection (DriverManager && DataSource)

```
public class JDBCHelper {
        …
        public Connection getConnectionDriverManager() throws
        SQLException {
          Connection connection = null;
          Properties connectionProps = new Properties();
          connectionProps.put("user", this.userName);
          connectionProps.put("password", this.password);

          if (this.dbms.equals("mysql")) {
             connection = DriverManager.getConnection(
                   String.format("jdbc:%s://%s:%s/", this.dbms,
        this.serverName, this.portNumber),
                   connectionProps
             );
             connection.setCatalog(this.dbName);
          }

          return connection;
        }
        …
}
```

```
public class JDBCHelper {
        …
        public Connection getConnectionDataSource() throws
        SQLException {
          MysqlDataSource dataSource = new
        MysqlDataSource();
          dataSource.setUser(this.userName);
          dataSource.setPassword(this.password);
          dataSource.setDatabaseName(this.dbName);

          return dataSource.getConnection();
        }
        …
}
```

Creating table, inserting, updating, selecting data from table.

```
try(
    val connection = jdbcHelper.getConnectionDriverManager();
    val stmt = connection.createStatement()
) {
    final String CREATE_UPLOAD_FILE_TABLE =
        "CREATE TABLE UPLOAD_FILE (" +
            "ID INT NOT NULL PRIMARY KEY AUTO_INCREMENT, " +
            "FILENAME TINYTEXT NOT NULL, " +
            "CONTENT_TYPE VARCHAR(50) NOT NULL DEFAULT 'IMAGE/JPG', "
            "DOWNLOAD_URL TEXT NOT NULL, " +
            "DESCRIPTION TEXT" +
        ")";
    final String INSERT_UPLOAD_FILE_TABLE =
        "INSERT INTO UPLOAD_FILE " +
        "(FILENAME, DOWNLOAD_URL, DESCRIPTION) " +
        "VALUES ('test', 'some/url', 'some description')";

//      final String UPDATE_UPLOAD_FILE_TABLE = "...";
//      final String DELETE_UPLOAD_FILE_TABLE = "...";
//      final String DROP_UPLOAD_FILE_TABLE = "...";

    stmt.executeUpdate(CREATE_UPLOAD_FILE_TABLE);
    stmt.executeUpdate(INSERT_UPLOAD_FILE_TABLE);
} catch (SQLException e) {
    e.printStackTrace();
}
```

```
final String SELECT_UPLOAD_FILE_TABLE = "SELECT * FROM UPLOAD_FILE";

val resultSet = stmt.executeQuery(SELECT_UPLOAD_FILE_TABLE);

while (resultSet.next()) {
    System.out.println(
        String.format(
            "id: %s, filename: %s",
            resultSet.getInt( columnLabel: "ID"),
            resultSet.getString( columnLabel: "FILENAME")
        )
    );
}
```

Spring JDBC template

The **JdbcTemplate** class is the central class in the **JDBC** core package. It handles the creation and release of resources, which helps you avoid common errors such as forgetting to close the connection. It performs the basic tasks of the core **JDBC** workflow such as statement creation and execution, leaving application code to provide **SQL** and extract results. The JdbcTemplate class executes **SQL** queries, update statements and stored procedure calls, performs iteration over ResultSets and extraction of returned parameter values. It also catches **JDBC** exceptions and translates them to the generic, more informative, exception hierarchy defined in the **org.springframework.dao package.**

Creating table, inserting, updating, selecting data from table.

- Initialise jdbcTemplate object:

```
val jdbcHelper = new JDBCHelper(
      "root",
      "",
      "mysql",
      "localhost",
      "33060",
      "lesson5"
);
val dataSource = jdbcHelper.getMysqlDataSource();
val jdbcTemplate = new JdbcTemplate(dataSource);
```

- Create table if not exist:

```
final String CREATE_UPLOAD_FILE_TABLE =
      "CREATE TABLE IF NOT EXISTS UPLOAD_FILE (" +
            "ID INT NOT NULL PRIMARY KEY AUTO_INCREMENT, " +
            "FILENAME TINYTEXT NOT NULL, " +
            "CONTENT_TYPE VARCHAR(50) NOT NULL DEFAULT 'IMAGE/JPG', "
+
            "DOWNLOAD_URL TEXT NOT NULL, " +
            "DESCRIPTION TEXT" +
            ")";
jdbcTemplate.execute(CREATE_UPLOAD_FILE_TABLE);
```

- Insert, update, delete table:

```
final String INSERT_UPLOAD_FILE_TABLE =
      "INSERT INTO UPLOAD_FILE " +
            "(FILENAME, DOWNLOAD_URL, DESCRIPTION) " +
            "VALUES ('test3', 'some/url', 'some description')";

val insertionCount=jdbcTemplate.update(INSERT_UPLOAD_FILE_TABLE);
```

- Select records:

```
val SELECT_UPLOADED_FILES = "SELECT FILENAME, CONTENT_TYPE,
DOWNLOAD_URL FROM UPLOAD_FILE";
val uploadedFiles = jdbcTemplate.query(
  SELECT_UPLOADED_FILES,
  new RowMapper<UploadedFile>() {
     public UploadedFile mapRow(ResultSet rs, int rowNum) throws
SQLException {
        UploadedFile uploadedFile = new UploadedFile();
        uploadedFile.setFileName(rs.getString("FILENAME"));
        uploadedFile.setContentType(rs.getString("CONTENT_TYPE"));
        uploadedFile.setDownloadUrl(rs.getString("DOWNLOAD_URL"));
        return uploadedFile;
     }
  }
);
uploadedFiles.forEach(
  uploadedFile -> System.out.println(uploadedFile.fileName)
);
```

5 min JDBCtemplate QUIZ

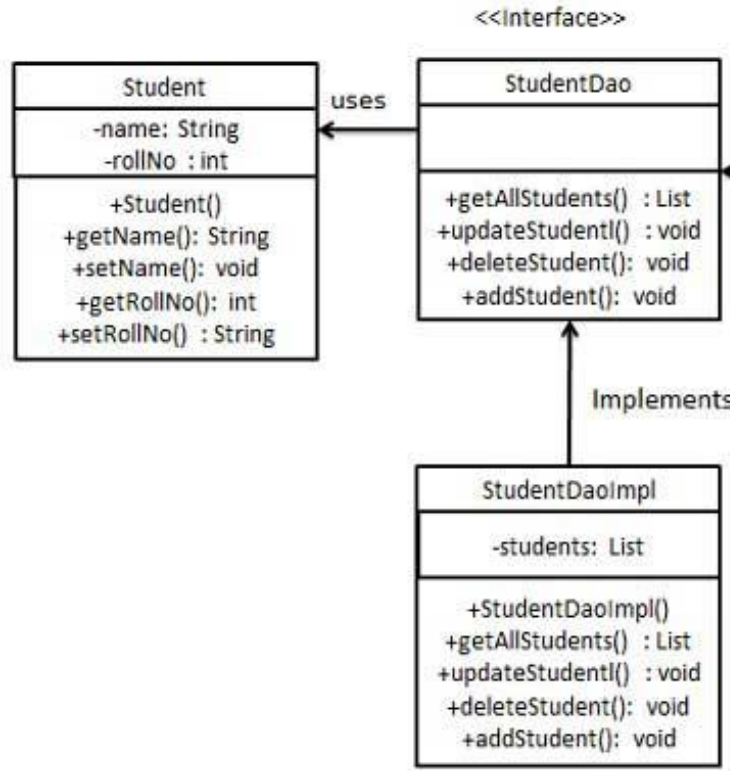
By using example that was provided before with JDBCTemplate:

- Create table **CAR** with fields that you have used in your previous exercises;
- Insert some data into newly created table;
- Select data and map to your model **Car** Class
- Update some data in new inserted records;
- Delete some data (rows);
- Drop table;

DAO as Data Access Object design pattern

Data Access Object Pattern or DAO pattern is used to separate low level data accessing API or operations from high level business services. Following are the participants in Data Access Object Pattern.

- Data Access Object Interface - This interface defines the standard operations to be performed on a model object(s).
- Data Access Object concrete class - This class implements above interface. This class is responsible to get data from a data source which can be database / xml or any other storage mechanism.
- Model Object or Value Object - This object is simple POJO containing get/set methods to store data retrieved using DAO class.



DAO example by using JDBCTemplate

```java
@Configuration
public class DaoConfig {
    @Bean
    DaoHelper daoHelper(DataSource dataSource) {
        return new DaoHelper(dataSource, dropTablesAndAddShutdownHook: true);
    }

    @Bean
    UserDao userDao(DaoHelper daoHelper) {
        UserDao userDao = new UserDao();
        userDao.setDaoHelper(daoHelper);
        return userDao;
    }
}
```

```java
@RestController
public class UserDaoController {
    @Autowired
    UserDao userDao;

    @GetMapping("/users")
    List<User> getUsers() { return userDao.getAllUsers(); }
}
```

```java
public class DaoHelper {
    Logger logger = LoggerFactory.getLogger(DaoHelper.class);

    private Schema[] schemas = {
        new MySQLSchema1()
    };

    private static boolean shutdownHookAdded;
    private DataSource dataSource;

    public DaoHelper(){

    }

    public DaoHelper(DataSource dataSource) {
        this.dataSource = dataSource;
    }

    public DaoHelper(DataSource dataSource, boolean dropTablesAndAddShutdownHook) {
        this.dataSource = dataSource;
        if (dropTablesAndAddShutdownHook) {
            dropTables();
            addShutdownHook();
```

JPA (Java Persistence API)

The Java Persistence API (JPA) is a Java application programming interface specification that describes the management of relational data in applications using Java Platform, Standard Edition and Java Platform, Enterprise Edition.

- Persistence in this context covers three areas:
- the API itself, defined in the javax.persistence package
- the Java Persistence Query Language (JPQL)
- object/relational metadata

The reference implementation for JPA is EclipseLink.

Related technologies:

- Enterprise JavaBeans
- Java Data Objects API
- Hibernate
- Spring Data JPA

JPA (Java Persistence API) in spring

Spring data JPA - An implementation of the repository abstraction that's a key building block of Domain-Driven Design based on the Java application framework Spring. Transparently supports all available JPA implementations and supports CRUD operations as well as the convenient execution of database queries.

Repository design pattern

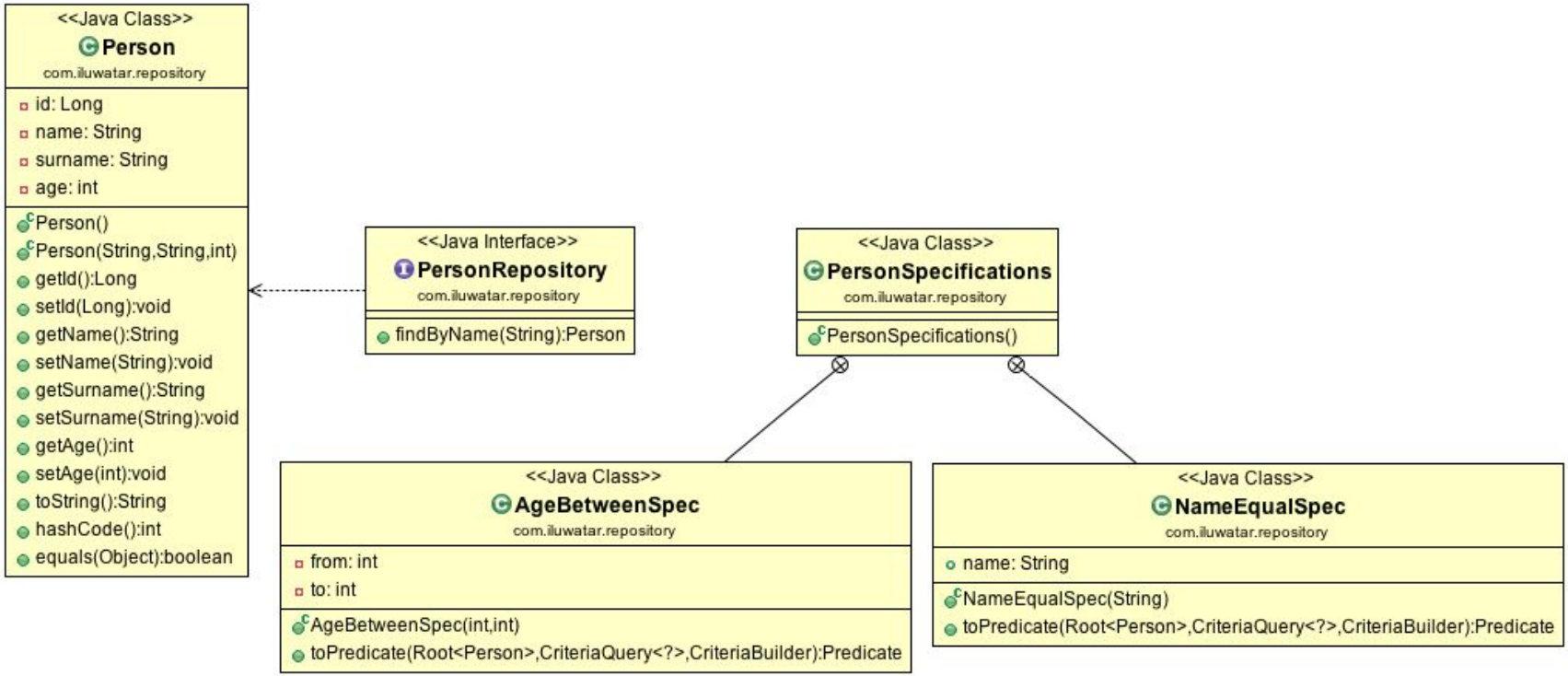
A **Repository** mediates between the domain and data mapping layers, acting like an in-memory domain object collection. Client objects construct query specifications declaratively and submit them to **Repository** for satisfaction. Objects can be added to and removed from the **Repository**, as they can from a simple collection of objects, and the mapping code encapsulated by the **Repository** will carry out the appropriate operations behind the scenes.

Repository layer is added between the domain and data mapping layers to isolate domain objects from details of the database access code and to minimize scattering and duplication of query code. The **Repository** pattern is especially useful in systems where number of domain classes is large or heavy querying is utilized.

Use the Repository pattern when

- the number of domain objects is large
- you want to avoid duplication of query code
- you want to keep the database querying code in single place
- you have multiple data sources

Repository implementation by using Spring data JPA (persistence tier)



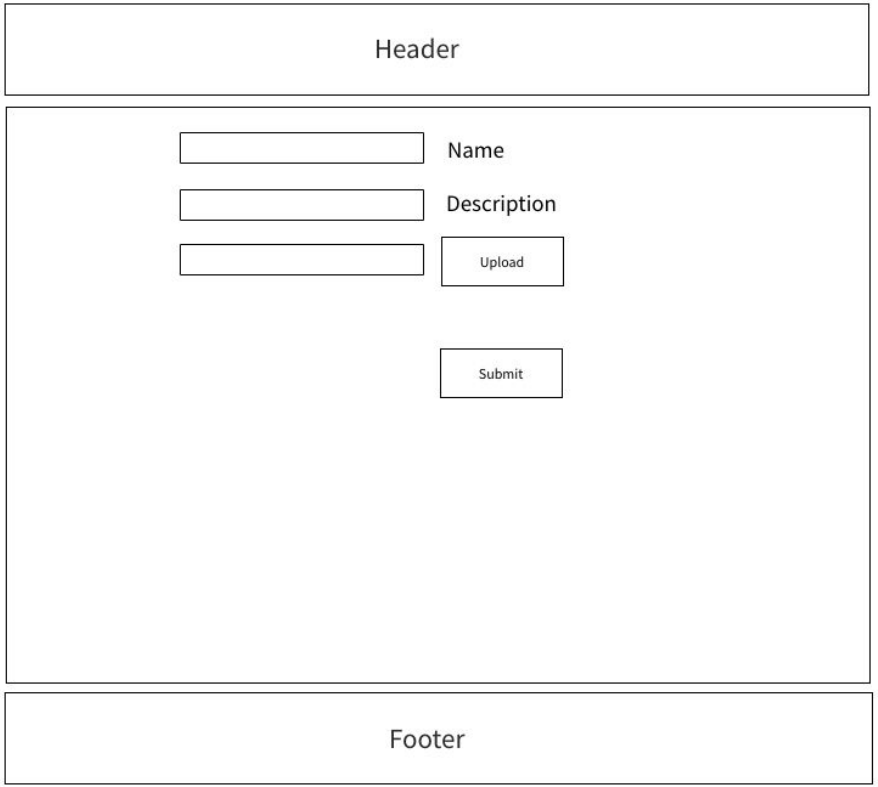
Home work :)

- Read about Spring Data JPA: https://docs.spring.io/spring-data/jpa/docs/current/reference/html/
- Watch video: https://www.youtube.com/watch?v=WZLTwbeENGs
- Set up and run project from https://spring.io/guides/gs/accessing-data-jpa/ (also read theory)
- Set up and run project from https://spring.io/guides/gs/accessing-data-mysql/ (also read theory)
- Create a web application that will let (look at wireframes in next slide as well):

1. Upload images (use file upload service from previous examples);
2. Enter metadata about images in database by using form;
3. Display all uploaded images;

No images uploaded

Upload image form



Display images