

# **Java Data Base Connectivity**

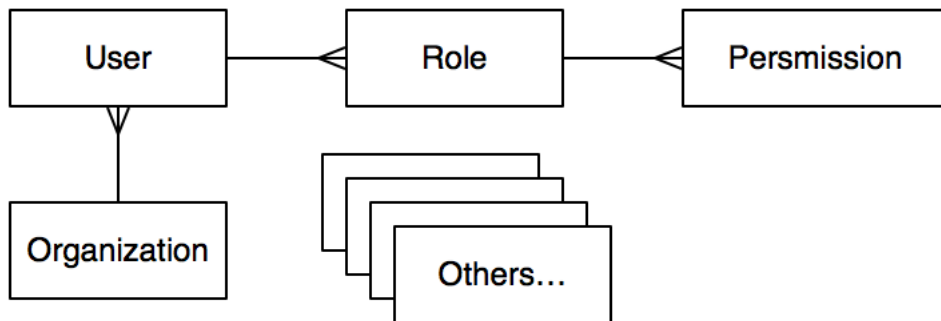
- Как подключиться к СУБД из Java
- Какие существуют драйверы работы с СУБД
- Как запрашивать данные из таблиц с помощью языка SQL
- Как вставлять, обновлять и удалять данные
- Как работать с исключениями JDBC
- Как управлять транзакциями

**JDBC API** – это программный интерфейс для предоставления доступа к табличным данным, в частности к данным хранимым в реляционных базах данных

С помощью JDBC API можно выполнить следующие основные операции

- Подключение к источнику данных таких как СУБД
- Отправлять в СУБД запросы на выборку и изменение
- Получать и обрабатывать результаты полученные от СУБД в ответ на запросы

Реляционная БД хранит связанные между собой объекты в структурах называемых таблицами – строки со столбцами. И обязанностью СУБД является предоставление доступа к этим объектам, их хранение и поддержание целостности данных\*.



User		Role	
ID		ID	
LOGIN		NAME	
PASSWORD_H	Organization	EM	
LAST_LOGIN_D			
...	ID		
	SHORT_NAME		
	INN		
	...		

- Уникальность колонок группы колонок в таблице – PRIMARY KEY
- Колонки таблиц содержат только данные из набора разрешенных – CHECK
- Колонки не содержат пустых данных – NOT NULL
- Колонка может содержать ссылку на строку в другой таблице – FOREIGN KEY
- ...

## Описание структуры данных – DDL

- **CREATE**
- **ALTER**
- **DROP**

```
CREATE TABLE USER (  
  id          NUMBER (18),  
  login       VARCHAR2(50 CHAR),  
  blocked     NUMBER (1),  
  CONSTRAINT pk_user PRIMARY KEY (id),  
  CONSTRAINT uq_login UNIQUE (login)  
)
```

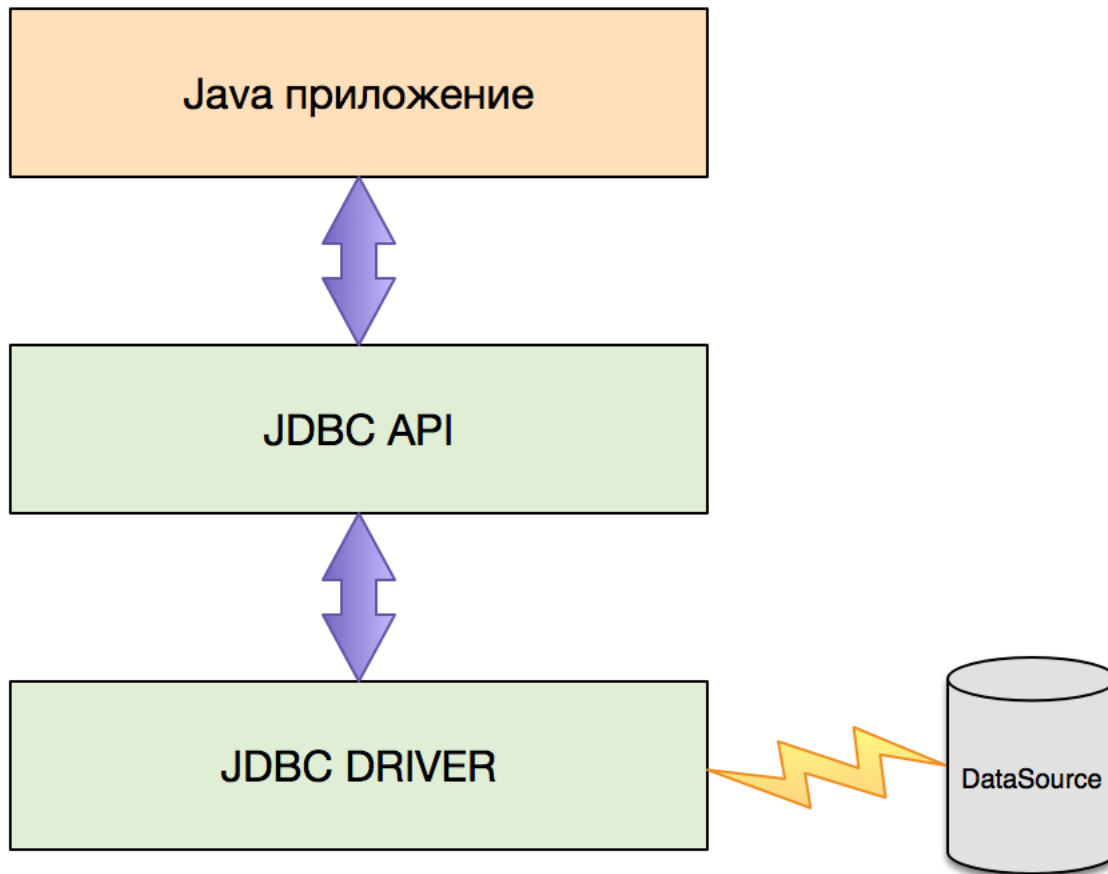
## Язык манипуляции данными – DML

- **SELECT**
- **INSERT**
- **UPDATE**
- **DELETE**

```
SELECT login  
FROM USER  
WHERE id = 12;
```

```
DELETE FROM  
USER  
WHERE id = 1313;
```

```
UPDATE USER  
SET blocked = 1  
WHERE id = 13;
```



JDBC драйверы бывают 4 типов

**Тип 1:** JDBC драйвер работает через JDBC-ODBC адаптер

**Тип 2:** JDBC драйвер вместо ODBC драйвера вызывает «нативный» код

**Тип 3:** JDBC драйвер полностью написан на Java но соединяется не напрямую с СУБД а с промежуточным сервером, который уже передает запросы в СУБД

**Тип 4:** JDBC драйвер полностью написан на Java и соединение происходит с сервером СУБД без каких-либо посредников



Использовать будем свободно распространяемую с открытым кодом СУБД **H2**

<http://www.h2database.com>

```
<!-- https://mvnrepository.com/artifact/com.h2database/h2 -->  
<dependency>  
    <groupId>com.h2database</groupId>  
    <artifactId>h2</artifactId>  
    <version>1.4.192</version>  
</dependency>
```

Чтобы отправить запрос в СУБД с помощью JDBC API нужно выполнить следующее

1. Зарегистрировать драйвер работы с СУБД (см. документацию драйвера)
2. Установить соединение с БД – получить активный Connection
3. Создать объект Statement для отправки SQL запроса
4. Выполнить Statement::executeXXX
5. Обработать ответ от СУБД
6. Закрыть Statement для освобождения ресурсов
7. Закрыть соединение с СУБД

Соединение предоставляется через интерфейс *java.sql.Connection*

Способы установления соединения с источником данных

- *java.sql.DriverManager.getConnection(String url)*
- *javax.sql.DataSource.getConnection()*

Адрес к источнику данных передается с помощью строки “**url**”, формат которой зависит от конкретного драйвера.

Пример URL:

- H2: *jdbc:mysql:~/testdb*
- MySQL: *jdbc:mysql://localhost:3306/*
- Java DB: *jdbc:derby:testdb;create=true*

```
interface Connection {  
    Statement createStatement() throws SQLException;  
    PreparedStatement prepareStatement(String sql) throws SQLException;  
    CallableStatement prepareCall(String sql) throws SQLException  
    void setAutoCommit(boolean autoCommit) throws SQLException;  
    void setTransactionIsolation(int level) throws SQLException;  
    void commit() throws SQLException;  
    void rollback() throws SQLException;  
    void close() throws SQLException;  
    ...  
}
```

Для отправки SQL запроса используется Statement объект

```
Statement statement = connection.createStatement();
```

Для параметризованного запроса используется PreparedStatement объект

```
PreparedStatement stmt = connection.prepareStatement("select * from user where login = ? ")  
stmt.setString(1, "root");
```

Объект посылает запрос в СУБД одним из предоставленных методов

- `boolean result = statement.execute("CREATE TABLE NAMES(NAME VARCHAR(100))")`
- `int result = statement.executeUpdate()`
- `ResultSet resultSet = statement.executeQuery();`

Запросы к реляционным БД всегда возвращают результат в виде табличных структур. JDBC скрывает работу с такими таблицами интерфейсом ResultSet

```
interface ResultSet {  
    long getLong(int columnIndex) throws SQLException;  
    String getString(int columnIndex) throws SQLException;  
    boolean next() throws SQLException;  
    void close() throws SQLException;  
    // ...  
}
```

При создании ResultSet можно установить дополнительные атрибуты

1. Тип итератора и чувствительность к изменениям данных ResultSet на стороне БД
2. Поддерживаемый уровень обновления данных ResultSet
3. Доступность данных после коммита текущей транзакции

```
Statement statement = connection.createStatement(  
    ResultSet.TYPE_FORWARD_ONLY,  
    ResultSet.CONCUR_READ_ONLY,  
    ResultSet.CLOSE_CURSORS_AT_COMMIT  
);
```

```
Statement statement = connection.createStatement(  
    ResultSet.TYPE_SCROLL_INSENSITIVE,  
    ResultSet.CONCUR_UPDATABLE,  
    ResultSet.HOLD_CURSORS_OVER_COMMIT  
);
```

Тип итератора и чувствительность к изменениям данных ResultSet на стороне БД

TYPE_FORWARD_ONLY	перемещение курсора только вперед
TYPE_SCROLL_INSENSITIVE	перемещение в обе стороны + данные не меняются после загрузки с БД
TYPE_SCROLL_SENSITIVE	перемещение в обе стороны + данные обновляются пока ResultSet открыт

\* Чтобы определить поддерживает ли СУБД данный тип вызываем  
`connection.getMetaData().supportsResultSetType(<TYPE>)`



Поддерживаемый уровень обновления данных ResultSet

**CONCUR\_READ\_ONLY** изменение строк ResultSet не поддерживается

**CONCUR\_UPDATABLE** разрешено обновление строк ResultSet

```
Statement stmt = connection.createStatement(TYPE_SCROLL_SENSITIVE, CONCUR_UPDATABLE);
ResultSet uprs = stmt.executeQuery("SELECT * FROM ACCOUNT");
while (uprs.next()) {
    BigDecimal b = uprs.getBigDecimal("BALANCE");
    uprs.updateBigDecimal("BALANCE", b.multiply(BigDecimal.valueOf(1.2d)));
    uprs.updateRow();
}
```

\*Чтобы определить поддерживает ли СУБД данный тип вызываем  
connection.getMetaData().supportsResultSetConcurrency(<CONCUR>)

Вызов метода `Connection.commit` может закрыть открытый `ResultSet`, но иногда может понадобиться оставить его открытым.

Можно указать атрибут доступности данных после коммита константами

`HOLD_CURSORS_OVER_COMMIT` оставить `ResultSet` открытым после коммита

`CLOSE_CURSORS_AT_COMMIT` закрыть `ResultSet` после коммита транзакции

\* Чтобы определить поддерживает ли СУБД данный тип вызываем  
`connection.getMetaData().supportsResultSetHoldability(<CURSORS>)`

ResultSet предоставляет

- механизм итерирования по строкам полученной таблицы
- методы получения типизированных значений колонок
  - по индексу колонки начиная с 1
  - по имени колонки

Метод	Описание
next	перевести курсор на одну строку вперед
previous	перевести курсор на одну строку назад
first	курсor на первую строку
last	курсor на последнюю строку
beforeFirst	курсor в начало ResultSet
afterLast	курсor в самый конец ResultSet
relative(int rows)	переместить курсор на заданное количество строк от текущей
absolute(int row)	переместить курсор на заданную строку

Для каждого типа Java + java.sql.\* есть геттеры по индексу колонки, начиная с 1!

Например:

- `getBytes(int column); getBytes(String column)`
- `getLong(int column); getLong(String column)`
- `getDouble(int column); getDouble(String column)`
- `getString(int column); getString(String column)`
- `getTimestamp(int column); getTimestamp(String column)`
- `getClob(int column); getClob(String column)`
- ...





































```
public void connectToAndQueryDatabase(String username, String password) throws
SQLException {
    try (Connection connection = DriverManager.getConnection(
        "jdbc:h2:./mydb", username, password);
        Statement stmt = connection.createStatement())
    {
        ResultSet resultSet= stmt.executeQuery("SELECT a, b, c FROM Table1");
        while (resultSet.next()) {
            int a = resultSet.getInt("a");
            String b = resultSet.getString("b");
            float c = resultSet.getFloat("c");
        }
    }
}
```

Все Statement объекты могут иметь список команд, которые возвращают количество измененных строк – UPDATE, DELETE, INSERT , а также команды DDL – CREATE TABLE, DROP TABLE...

```
private static void insertTelephones(Connection connection) throws SQLException {  
    try(Statement statement = connection.createStatement()) {  
        statement.addBatch("INSERT INTO telephone values ('ivan', '1231231')");  
        statement.addBatch("INSERT INTO telephone values ('stepan', '4231231')");  
        statement.addBatch("INSERT INTO telephone values ('kostya', '5231231')");  
        int[] executeBatch = statement.executeBatch();  
    }  
}
```

```
private static void insertTelephones(Connection connection) throws SQLException {  
    try(PreparedStatement statement =  
        connection.prepareStatement("INSERT INTO telephone VALUES (?, ?)")) {  
  
        statement.setString(1, "feodor");  
        statement.setString(2, "9949433");  
        statement.addBatch();  
  
        statement.setString(1, "anastasiya");  
        statement.setString(2, "8345783458");  
        statement.addBatch();  
  
        int[] executeBatch = statement.executeBatch();  
    }  
}
```



- ▼   Exception (java.lang)
  - ▼   SQLException (java.sql)
    - ▼   SQLNonTransientException (java.sql)
      -   SQLFeatureNotSupportedException (java.sql)
      -   SQLSyntaxErrorException (java.sql)
      -   SQLInvalidAuthorizationSpecException (java.sql)
      -   SQLDataException (java.sql)
      -   SQLNonTransientConnectionException (java.sql)
      -   SQLIntegrityConstraintViolationException (java.sql)
    -   SQLClientInfoException (java.sql)
    -   SQLRecoverableException (java.sql)
    -   BatchUpdateException (java.sql)
  - ▼   SQLTransientException (java.sql)
    -   SQLTimeoutException (java.sql)
    -   SQLTransactionRollbackException (java.sql)
    -   SQLTransientConnectionException (java.sql)
  - ▼   SQLWarning (java.sql)
    -   DataTruncation (java.sql)

Метод	Описание
getMessage	описание ошибки JDBC драйвера
getSQLState	XOPEN SQLState – возвращает стандартный код состояния запроса, <a href="https://docs.oracle.com/database/121/ZZMOD/appd.htm#ZZMOD338">https://docs.oracle.com/database/121/ZZMOD/appd.htm#ZZMOD338</a>
getErrorCode	возвращает номер ошибки, специфичный для вашей реализации БД.
getNextException	возвращает следующий SQLException в цепочке, если при исполнении запроса было сгенерировано несколько SQLException

SQLState: 42Y55

Error Code: 30000

Message: 'DROP TABLE' cannot be performed on 'USERS' because it does not exist.

Исключение	Описание
SQLNonTransientException	ошибка, которая не может быть исправлена кроме как исправлением запроса или его данных
SQLTransientException	ошибка, которая может быть исправлена при повторе операции позднее.
SQLRecoverableException	ошибка, которая может быть исправлена методом «попробуйте выключить и включить»: переустановить соединение, переповторить транзакцию итд.
BatchUpdateException	выбрасывается при ошибках выполнения executeBatch()

Иногда требуется чтобы результат выполнения одного Statement не было применено до тех пор пока другой Statement не выполнится успешно.

Например, перевод средств со счета на счет.

Для этих целей используется понятие транзакции.

**Транзакция** – неделимая с точки зрения воздействия на БД последовательность операторов манипулирования данными (чтения, удаления, вставки, модификации), такая, что:

- либо результаты всех операторов, входящих в транзакцию, отображаются в БД
- либо воздействие всех операторов полностью отсутствует

Чтобы включить ручное управление транзакцией необходимо отключить авто-коммит:

`Connection::setAutoCommit(false);`

Чтобы закоммитить (применить изменения) транзакцию вызвать метод:

`Connection::commit();`

Чтобы откатить (отменить все изменения) транзакцию вызвать метод:

`Connection::rollback();`

```
void transfer100$(Connection connection) throws SQLException {  
    connection.setAutoCommit(false);  
    try (Statement statement = connection.createStatement()) {  
        statement.execute("UPDATE account SET balance = balance + 100 WHERE id = 1");  
        statement.execute("UPDATE account SET balance = balance - 100 WHERE id = 2");  
        connection.commit();  
    } catch (SQLException e) {  
        connection.rollback();  
    }  
}
```

Начиная с JDBC 3 появилась возможность более гранулированного управления транзакциями – **Savepoints**. Позволяют организовать подтранзакции.

Метод Connection::	Описание
setSavepoint(String name)	установить точку сохранения
releaseSavepoint(Savepoint s)	удалить точку сохранения
rollback(Savepoint s)	откатить все изменения БД до указанной точки сохранения

```
void tryTransfer100$(Connection connection) throws SQLException {  
    connection.setAutoCommit(false);  
    Savepoint savepoint = connection.setSavepoint();  
    try (Statement statement = connection.createStatement()) {  
        statement.execute("UPDATE account SET balance = balance + 100 WHERE id = 1");  
        statement.execute("UPDATE account SET balance = balance - 100 WHERE id = 2");  
        connection.commit();  
    } catch (SQLException e) {  
        connection.rollback(savepoint);  
    }  
}
```

В общем случае транзакции могут выполняться параллельно. Проблемы целостности данных могут возникнуть когда несколько транзакций работают с одними и теми же объектами. В СУБД эти проблемы решаются с помощью блокировок.

Блокировки могут устанавливаться как на чтение данных до их коммита так и на изменение данных которые в данный момент изменяются.

Поведение блокировок называется уровнем изоляции транзакций:

1. TRANSACTION\_NONE – транзакций нет
2. TRANSACTION\_READ\_COMMITTED – чтение только закоммиченных данных
3. TRANSACTION\_READ\_UNCOMMITTED – чтение «грязных» данных
4. TRANSACTION\_REPEATABLE\_READ – повторное чтение вернет тот же результат
5. TRANSACTION\_SERIALIZABLE – все транзакции выполняются одна за другой



Проблема	Описание
потерянное обновление	при одновременном изменении одного блока данных разными транзакциями одно из изменений теряется
«грязное» чтение	чтение данных, добавленных или изменённых транзакцией, которая впоследствии не подтвердится (откатится)
неповторяющееся чтение	при повторном чтении в рамках одной транзакции ранее прочитанные данные оказываются изменёнными
фантомное чтение	транзакция А несколько раз выбирает множество строк по критериям X. Транзакция Б в интервалах между этими выборками успешно добавляет или удаляет строки или изменяет столбцы некоторых строк, используемых в критериях X. В результате, что одни и те же выборки в транзакции А дают разные множества строк

# ПОБОЧНЫЕ ЭФФЕКТЫ ПРИ РАЗНЫХ УРОВНЯХ ИЗОЛЯЦИИ

Уровень изоляции	Фантомное чтение	Неповторяющееся чтение	«Грязное» чтение	Потерянное обновление
SERIALIZABLE	+	+	+	+
REPEATABLE READ	-	+	+	+
READ COMMITTED	-	-	+	+
READ UNCOMMITTED	-	-	-	+
NO TRANSACTION	-	-	-	-

- <http://tutorials.jenkov.com/jdbc/index.html>
- [https://ru.wikipedia.org/wiki/Уровень\\_изолированности\\_транзакций](https://ru.wikipedia.org/wiki/Уровень_изолированности_транзакций)

Разработать продвинутый кэш, который помнит о кэшированных данных после перезапуска приложения.

```
@interface Cachable {  
    boolean persistent() default false;  
}
```

```
class Calculator {  
    @Cachable(persistent = true)  
    public int fibonacci(int n) {  
        // algorithm  
    }  
}
```