Занятие № 15.

Базы данных, JDBC

SQL

БД MySQL

Введение в JDBC – statement, preparedstatement

SQL

SQL (англ. structured query language — «язык структурированных запросов») — формальный непроцедурный язык программирования, применяемый для создания, модификации и управления данными в произвольной реляционной базе данных, управляемой соответствующей системой управления базами данных (СУБД).

Выборка в базе данных производится с помощью оператора SELECT.
В общем виде он выглядит так:

SELECT названия_полей FROM названия_таблиц WHERE условие [ORDER BY, LIMIT]

Где ORDER BY и LIMIT - дополнительные опции.

Попробуем применить его. Выберем все значения поля username из таблицы table1.

```
SELECT username FROM table1;
```

и отсортируем их

```
SELECT username FROM table1 ORDER BY username;
```

Как видно, **ORDER BY** используется для сортировки по одному из полей, указанных после оператора **SELECT**. По умолчанию делается возрастающая сортировка, если хотим использовать сортировку в обратном порядке, то после поля необходимо добавить **DESC**:

```
SELECT username FROM table1 ORDER BY username DESC;
```

Так как нам нужны все значения, то оператор WHERE можно не использовать. Ещё один пример: выбираем значения полей phone_id и user_id из таблицы table2, где phone_number равен '200'.

```
SELECT phone_id, user_id FROM table2 WHERE phone_number=200;
SELECT phone_id, user_id FROM table2 WHERE phone_number=200 LIMIT 1, 3;
```

LIMIT выводит строки в указанном диапазоне (нижняя граница не включается). Если первый аргумент не указан, то он считается равным 0.

```
SELECT phone_id, user_id FROM table2 WHERE phone_number=200 LIMIT 3;
```

Виды оператора JOIN

Рассмотрим следующие таблицы:

City (Города)

Id	Name
1	Москва
2	Санкт-Петербург
3	Казань

Person (Люди)

Name	CityId
Андрей	1
Леонид	2
Сергей	1
Григорий	4

INNER JOIN

Оператор внутреннего соединения INNER JOIN соединяет две таблицы. Порядок таблиц для оператора неважен, поскольку оператор является симметричным.
Заголовок таблицы-результата является объединением (конкатенацией) заголовков соединяемых таблиц.
Тело результата логически формируется следующим образом. Каждая строка одной таблицы сопоставляется с каждой строкой второй таблицы, после чего для полученной «соединённой» строки проверяется условие соединения (вычисляется предикат соединения). Если условие истинно, в таблицу-результат добавляется соответствующая «соединённая» строка.
Описанный алгоритм действий является строго логическим, то есть он лишь объясняет результат, который должен получиться при выполнении операции, но не предписывает, чтобы конкретная СУБД выполняла соединение именно указанным образом. Существует множество способов реализации операции соединения, например, соединение вложенными циклами (англ. inner loops join), соединение хэшированием (англ. hash join), соединение слиянием (англ. merge join). Единственное требование состоит в том, чтобы любая реализация логически давала такой же результат, как при применении описанного алгоритма.

```
SELECT *
FROM
  Person
  INNER JOIN
  City
    ON Person.CityId = City.Id
```

Результат:

Person.Name	Person.CityId	City.Id	City.Name
Андрей	1	1	Москва
Леонид	2	2	Санкт-Петербург
Сергей	1	1	Москва

OUTER JOIN

Соединение двух таблиц, в результат которого в обязательном порядке входят все строки либо одной, либо обеих таблиц.

LEFT OUTER JOIN

Оператор левого внешнего соединения LEFT OUTER JOIN соединяет две таблицы. Порядок таблиц для оператора важен, поскольку оператор не является симметричным.

Заголовок таблицы-результата является объединением (конкатенацией) заголовков соединяемых таблиц.

Тело результата логически формируется следующим образом. Пусть выполняется соединение левой и правой таблиц по предикату (условию) p.

1. В результат включается внутреннее соединение (*INNER JOIN*) левой и правой таблиц по предикату *p*.

2. Затем в результат добавляются те записи левой таблицы, которые не вошли во внутреннее соединение на шаге 1. Для таких записей поля, соответствующие правой таблице, заполняются значениями *NULL*.

```sql
SELECT *
FROM
  Person -- Левая таблица
  LEFT OUTER JOIN
  City   -- Правая таблица
    ON Person.CityId = City.Id
```

Результат:

Person.Name	Person.CityId	City.Id	City.Name
Андрей	1	1	Москва
Леонид	2	2	Санкт-Петербург
Сергей	1	1	Москва
Григорий	4	NULL	NULL

RIGHT OUTER JOIN

Оператор правого внешнего соединения **RIGHT OUTER JOIN** соединяет две таблицы. Порядок таблиц для оператора важен, поскольку оператор не является симметричным.

Заголовок таблицы-результата является объединением (конкатенацией) заголовков соединяемых таблиц.

Тело результата логически формируется следующим образом. Пусть выполняется соединение левой и правой таблиц по предикату (условию) p.

1. В результат включается внутреннее соединение (*INNER JOIN*) левой и правой таблиц по предикату *p*.
2. Затем в результат добавляются те записи правой таблицы, которые не вошли во внутреннее соединение на шаге 1. Для таких записей поля, соответствующие левой таблице, заполняются значениями *NULL*.

```sql
SELECT *
FROM
  Person -- Левая таблица
  RIGHT OUTER JOIN
  City   -- Правая таблица
    ON Person.CityId = City.Id
```

Результат:

Person.Name	Person.CityId	City.Id	City.Name
Андрей	1	1	Москва
Сергей	1	1	Москва
Леонид	2	2	Санкт-Петербург
NULL	NULL	3	Казань

FULL OUTER JOIN

Оператор полного внешнего соединения FULL OUTER JOIN соединяет две таблицы. Порядок таблиц для оператора неважен, поскольку оператор является симметричным.
Заголовок таблицы-результата является объединением (конкатенацией) заголовков соединяемых таблиц.
Тело результата логически формируется следующим образом. Пусть выполняется соединение первой и второй таблиц по предикату (условию) p. Слова «первой» и «второй» здесь не обозначают порядок в записи (который не важен), а используются лишь для различения таблиц.

1. В результат включается внутреннее соединение (*INNER JOIN*) первой и второй таблиц по предикату *p*.
2. В результат добавляются те записи первой таблицы, которые не вошли во внутреннее соединение на шаге 1. Для таких записей поля, соответствующие второй таблице, заполняются значениями *NULL*.
3. В результат добавляются те записи второй таблицы, которые не вошли во внутреннее соединение на шаге 1. Для таких записей поля, соответствующие первой таблице, заполняются значениями *NULL*.

```
SELECT *
FROM
  Person
  FULL OUTER JOIN
  City
    ON Person.CityId = City.Id
```

Результат:

Person.Name	Person.CityId	City.Id	City.Name
Андрей	1	1	Москва
Сергей	1	1	Москва
Леонид	2	2	Санкт-Петербург
NULL	NULL	3	Казань
Григорий	4	NULL	NULL

CROSS JOIN

Оператор перекрёстного соединения, или декартова произведения CROSS JOIN соединяет две таблицы. Порядок таблиц для оператора неважен, поскольку оператор является симметричным. Заголовок таблицы-результата является объединением (конкатенацией) заголовков соединяемых таблиц.
Тело результата логически формируется следующим образом. Каждая строка одной таблицы соединяется с каждой строкой второй таблицы, давая тем самым в результате все возможные сочетания строк двух таблиц.

```
SELECT *
FROM
  Person
  CROSS JOIN
  City
```

или

```
SELECT *
FROM
  Person,
  City
```

Результат:

Person.Name	Person.CityId	City.Id	City.Name
Андрей	1	1	Москва
Андрей	1	2	Санкт-Петербург
Андрей	1	3	Казань
Леонид	2	1	Москва
Леонид	2	2	Санкт-Петербург
Леонид	2	3	Казань
Сергей	1	1	Москва
Сергей	1	2	Санкт-Петербург
Сергей	1	3	Казань
Григорий	4	1	Москва
Григорий	4	2	Санкт-Петербург
Григорий	4	3	Казань

Если в предложении **WHERE** добавить условие соединения (предикат p), то есть ограничения на сочетания кортежей, то результат эквивалентен операции **INNER JOIN** с таким же условием:

```
SELECT *
FROM
  Person,
  City
WHERE
  Person.CityId = City.Id
```

Таким образом, выражения

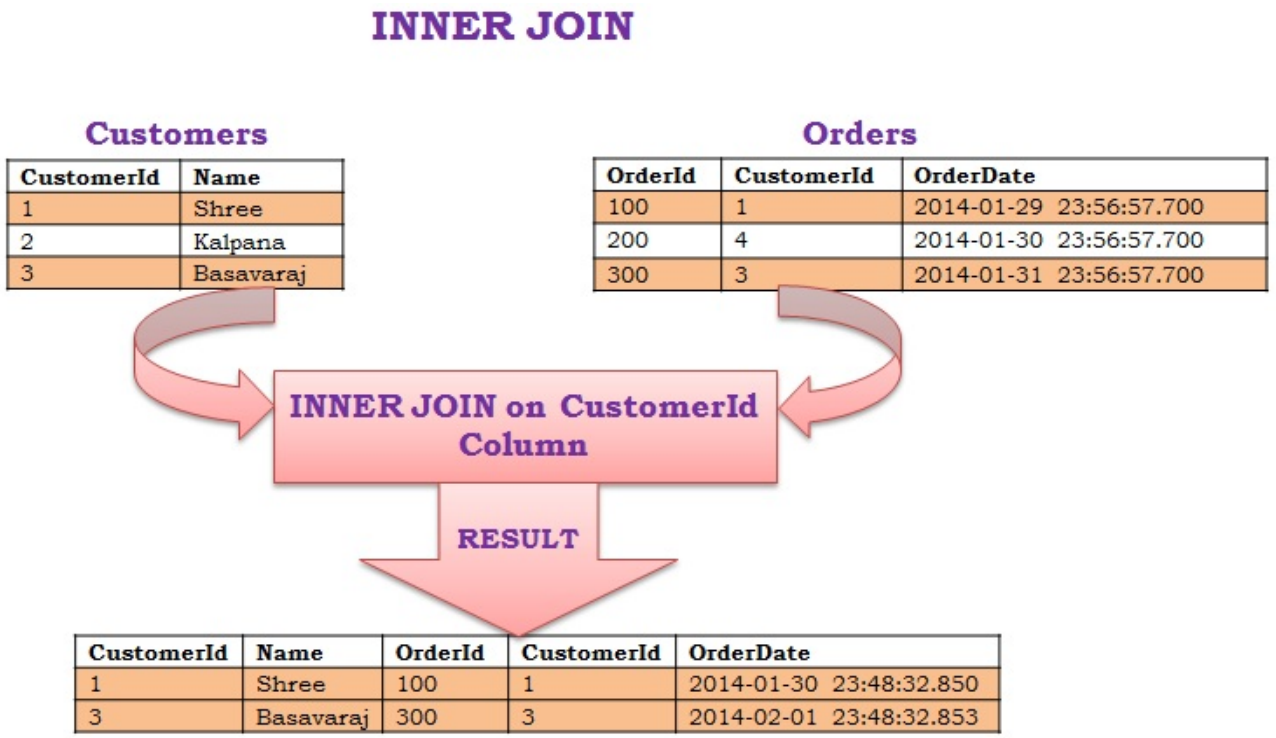
```
t1 CROSS JOIN t2 WHERE p
```

и

```
t1 INNER JOIN t2 ON p
```

синтаксически являются альтернативными формами записи одной и той же логической операции внутреннего соединения по предикату p. Синтаксис CROSS JOIN + WHERE для операции соединения называют устаревшим, его не рекомендует стандарт SQL ANSI.

В таком случае лучше использовать INNER JOIN.



БД MySQL

MySQL – это одна из самых популярных и самых распространенных СУБД (система управления базами данных) в интернете. Она не предназначена для работы с большими объемами информации, но ее применение идеально для интернет сайтов, как небольших, так и достаточно крупных.

MySQL отличатся хорошей скоростью работы, надежностью, гибкостью. Работа с ней, как правило, не вызывает больших трудностей. Поддержка сервера MySQL автоматически включается в поставку PHP.

Немаловажным фактором является ее бесплатность. MySQL распространяется на условиях общей лицензии GNU (GPL, GNU Public License).

Ранее для долговременного хранения информации мы работали с файлами: помещали в них некоторое количество строчек, а затем извлекали их для последующей работы. Задача длительного хранения информации очень часто встречается в программировании Web-приложений: подсчёт посетителей в счётчике, хранение сообщений в форуме, удалённое управление содержанием информации на сайте и т.д.

Между тем, профессиональные приёмы работы с файлами очень трудоёмки: необходимо заботится о помещении в них информации, о её сортировке, извлечении, при этом не нужно забывать, что все эти действия будут происходить на сервере хост-провайдера, где с очень большой вероятностью стоит один из вариантов Unix - следовательно, нужно так же заботится о правах доступа к файлам и их размещении. При этом объём кода значительно возрастает, и совершить ошибку в программе очень просто.

Все эти проблемы решает использование базы данных. Базы данных сами заботятся о безопасности информации и её сортировке и позволяют извлекать и размещать информацию при помощи одной строчки. Код с использованием базы данных получается более компактным, и отлаживать его гораздо легче. Кроме того, не нужно забывать и о скорости - выборка информации из базы данных происходит значительно быстрее, чем из файлов.

Таким образом, основное достоинство базы данных заключается в том, что она берёт на себя всю работу с жёстким диском и делает это очень эффективно.

Введение в JDBC – statement, preparedstatement

JDBC (англ. Java DataBase Connectivity — соединение с базами данных на Java) — платформенно-независимый промышленный стандарт взаимодействия Java-приложений с различными СУБД, реализованный в виде пакета java.sql, входящего в состав Java SE.

JDBC основан на концепции так называемых драйверов, позволяющих получать соединение с базой данных по специально описанному URL. Драйверы могут загружаться динамически (во время работы программы). Загрузившись, драйвер сам регистрирует себя и вызывается автоматически, когда программа требует URL, содержащий протокол, за который драйвер отвечает.

Интерфейсы

JDBC API содержит два основных типа интерфейсов: первый — для разработчиков приложений и второй (более низкого уровня) — для разработчиков драйверов.

Соединение с базой данных описывается классом, реализующим интерфейс java.sql.Connection. Имея соединение с базой данных, можно создавать объекты типа Statement, служащие для исполнения запросов к базе данных на языке SQL.

Существуют следующие виды типов Statement, различающихся по назначению:

- `java.sql.Statement` — Statement общего назначения;
- `java.sql.PreparedStatement` — Statement, служащий для выполнения запросов, содержащих подставляемые параметры (обозначаются символом '?' в теле запроса);
- `java.sql.CallableStatement` — Statement, предназначенный для вызова хранимых процедур.

Интерфейс `java.sql.ResultSet` позволяет легко обрабатывать результаты запроса.

Преимуществами JDBC считают:

- Лёгкость разработки: разработчик может не знать специфики базы данных, с которой работает;
- Код практически не меняется, если компания переходит на другую базу данных (количество изменений зависит исключительно от различий между диалектами SQL);
- Не нужно устанавливать громоздкую клиентскую программу;
- К любой базе можно подсоединиться через легко описываемый URL.

Данный пример использует свободный драйвер JDBC для MySQL

```
package javaapplication1;
import java.sql.*;

public class Main {

    public static void main(String[] args) throws SQLException {
        /**
         * эта строка загружает драйвер DB.
         * раскомментируйте если прописываете драйвер вручную
         */
        //Class.forName("com.mysql.jdbc.Driver");

        Connection conn = DriverManager.getConnection(
            "jdbc:mysql://localhost:3306/db_name",
            "user", "password");

        if (conn == null) {
            System.out.println("Нет соединения с БД!");
            System.exit(0);
        }

        Statement stmt = conn.createStatement();
        ResultSet rs = stmt.executeQuery("SELECT * FROM users");

        while (rs.next()) {
            System.out.println(rs.getRow() + ". " + rs.getString("firstname")
                    + "\t" + rs.getString("lastname"));
        }

        /**
         * stmt.close();
         * При закрытии Statement автоматически закрываются
         * все связанные с ним открытые объекты ResultSet
         */
        stmt.close();
    }
}
```

Еще пример:

```
package jdbcproj;

import oracle.jdbc.driver.OracleDriver;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class Jdbcproj {

private static final String DB_DRIVER = "oracle.jdbc.driver.OracleDriver";
        private static final String DB_CONNECTION = "jdbc:oracle:thin:@localhost:1521:ACTPRO";
        private static final String DB_USER = "user";
        private static final String DB_PASSWORD = "password";

        public static void main(String[] argv) {

                try {

                        selectRecordsFromDbUserTable();

                } catch (SQLException e) {

                        System.out.println(e.getMessage());

                }

        }

        private static void selectRecordsFromDbUserTable() throws SQLException {

                Connection dbConnection = null;
                Statement statement = null;

                String selectTableSQL = "SELECT USER_ID, USERNAME from DBUSER";

                try {
                        dbConnection = getDBConnection();
                        statement = dbConnection.createStatement();

                        System.out.println(selectTableSQL);
```

```java
private static void selectRecordsFromDbUserTable() throws SQLException {

        Connection dbConnection = null;
        Statement statement = null;

        String selectTableSQL = "SELECT USER_ID, USERNAME from DBUSER";

        try {
                dbConnection = getDBConnection();
                statement = dbConnection.createStatement();

                System.out.println(selectTableSQL);

                // execute select SQL stetement
                ResultSet rs = statement.executeQuery(selectTableSQL);

                while (rs.next()) {

                        String userid = rs.getString("USER_ID");
                        String username = rs.getString("USERNAME");

                        System.out.println("userid : " + userid);
                        System.out.println("username : " + username);

                }

        } catch (SQLException e) {

                System.out.println(e.getMessage());

        } finally {

                if (statement != null) {
                        statement.close();
                }

                if (dbConnection != null) {
                        dbConnection.close();
                }

        }

}

private static Connection getDBConnection() {
```

```
    private static Connection getDBConnection() {

        Connection dbConnection = null;

        try {

            Class.forName(DB_DRIVER);

        } catch (ClassNotFoundException e) {

            System.out.println(e.getMessage());

        }

        try {

            dbConnection = DriverManager.getConnection(DB_CONNECTION, DB_USER,
                    DB_PASSWORD);
            return dbConnection;

        } catch (SQLException e) {

            System.out.println(e.getMessage());

        }

        return dbConnection;

    }

}
```

Пример использования объекта JDBC PreparedStatement — получение списка записей

Объект типа PreparedStatement может использоваться не только для выполнения операций создания, обновления и удаления записей в таблице базы данных, но также и для получения списка записей с заданием условий поиска. Для выполнения запроса на получение данных применяется метод executeQuery().

```
package ru.j4web.examples.java.jdbc.jdbcpreparedstatementselectexample;

import java.sql.Connection;
import java.sql.Date;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.logging.Level;
import java.util.logging.Logger;

public class JDBCPreparedStatementSelectExample {

    private static final String DB_URL = "jdbc:mysql://dev-server/sampledb"
            + "?user=sampleuser&password=samplepassword";
    private static final String SQL_STATEMENT = "SELECT user_id, username, "
            + "created_by, creation_date FROM users WHERE user_id > ?";

    public static void main(String[] args) {

        try (Connection connection = DriverManager.getConnection(DB_URL);
                PreparedStatement statement
                = connection.prepareStatement(SQL_STATEMENT)) {
```

```
    public static void main(String[] args) {

        try (Connection connection = DriverManager.getConnection(DB_URL);
                PreparedStatement statement
                = connection.prepareStatement(SQL_STATEMENT)) {

            statement.setInt(1, 7);

            ResultSet result = statement.executeQuery();

            while(result.next()) {
                System.out.println("================================");
                Integer userId = result.getInt("user_id");
                String username = result.getString("username");
                String createdBy = result.getString("created_by");
                Date creationDate = result.getDate("creation_date");
                System.out.println("Next row: user_id = " + userId
                        + "; username = " + username + "; created_by = "
                        + createdBy + "; creation_date = " + creationDate);
            }

        } catch (SQLException ex) {
            Logger.getLogger(JDBCPreparedStatementSelectExample.class
                    .getName()).log(Level.SEVERE, null, ex);
        }
    }
}
```

```
--- exec-maven-plugin:1.2.1:exec (default-cli) @ JDBCPreparedStatementSelectExample ---
================================
Next row: user_id = 8; username = user_8; created_by = j4web.ru; creation_date = 2016-03-02
================================
Next row: user_id = 9; username = user_9; created_by = j4web.ru; creation_date = 2016-03-02
================================
Next row: user_id = 10; username = user_10; created_by = j4web.ru; creation_date = 2016-03-02
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
```

Использование переменных связывания в SQL для повышения производительности приложений и обеспечения их безопасности

Использование переменных связывания в SQL-выражениях может повысить общую производительность и безопасность вашего веб-приложения, а также является чрезвычайно мощной защитой от атак с использованием SQL-инъекций.

Переменная связывания— это переменная, обозначенная подстановочным символом, таким как (?), :name или @name. Подстановочный символ зависит от используемого сервера базы данных с поддержкой SQL . Реальное значение подстановочного символа вводится в процессе функционирования, непосредственно перед исполнением SQL-выражения.

Оптимизация производительности SQL-выражений

Рассмотрим типичную Java-программу, в которой SQL-выражения написаны с использованием литералов. Для каждого цикла создается новое SQL-выражение. Каждый раз, когда цикл встречается с новым значением, создается и исполняется новый SQL-запрос.

Простой SQL-запрос без переменных связывания

```
sql = "SELECT t.name FROM hr.employees t WHERE employee_id = ";

System.out.println("Start: " + new Date());

for(int i=0; i&lt;10000; i++)
    {
        statement = connection.createStatement();
        resultset = statement.executeQuery(sql + Integer.toString(i));
        if(resultset.next())
        {
            name = resultset.getString("name");
            doSomething(name);
        }
        resultset.close();
        statement.close();
    }

System.out.println("End: " + new Date());
```

Для исполнения этого кода потребовалось приблизительно 11 секунд. Теперь перепишем этот код с использованием подготовленных выражений (prepared statement) и переменных связывания.

Подготовленные выражения с переменными связывания

В следующем примере серверу отправляется запрос с определенной переменной связывания. В процессе исполнения мы связываем Java-переменную i с SQL-выражением. Это позволяет нам использовать один и тот же план исполнения для 10000 запросов, что повышает производительность за счет сведения к минимуму количества разборов SQL.

Простое SQL-выражение с переменными связывания

```
sql = "SELECT t.name FROM hr.employees t WHERE employee_id = ?";
System.out.println("Start: " + new Date());
for(int i=0; i&lt;10000; i++)
    {
        statement = connection.prepareStatement(sql);
        statement.setInt(1, i);
        resultset = statement.executeQuery();
        if(resultset.next())
        {
            name = resultset.getString("name");
            doSomething(name);
        }
        resultset.close();
        statement.close();
    }
            System.out.println("End: " + new Date());
```

Для исполнения этого кода потребовалось приблизительно 7 секунд. Однако обратите внимание, что этот код создает новое выражение для каждого цикла. Мы можем дополнительно улучшить этот результат, создав одно-единственное выражение и используя его повторно в каждом цикле.

Многократное использование выражения

```
sql = "SELECT t.name FROM hr.employees t WHERE employee_id = ?";
statement = connection.prepareStatement(sql);
System.out.println("Start: " + new Date());
for(int i=0; i<10000; i++)
    {
        statement.setInt(1, i);
        resultset = statement.executeQuery();
        if(resultset.next())
        {
            name = resultset.getString("name");
            doSomething(name);
        }
        resultset.close();
    }
            System.out.println("End: " + new Date());

statement.close();
```

Этому Java-коду потребовалось приблизительно 4 секунды для исполнения тех же SQL-операций, что и в первоначальном коде, на исполнение которого ушло 11 секунд.

Типы атак с использованием SQL-инъекций

Атаки с использованием SQL-инъекций в 2013 году заняли первое место в рейтинге угроз безопасности организации Open Web Application Security Project. При атаке с использованием SQL-инъекции в базу данных веб-приложения через поле для ввода информации вставляются злонамеренные SQL-выражения с целью вынудить приложение исполнить эти выражения. Для успешной атаки с использованием SQL-инъекций необходимо, чтобы программный код приложения был уязвимым для ввода данных пользователем.

Атаки с использованием SQL-инъекций используют уязвимость приложения к введенным пользователем данным в результате либо некорректной обработки escape-символов в строковых литералах со встроенными SQL-выражениями, либо в результате отсутствия строгого контроля типов вводимых данных.

Некорректная обработка escape-символов

При атаке первого типа хакер помещает текст, содержащий escape-символы и встроенные SQL-выражения, в поле формы веб-приложения или в атрибут запроса. Если веб-приложение не обрабатывает escape-символы, этот текст — вместе со злонамеренными SQL-выражениями — передается в базу данных для исполнения.

Эту уязвимость иллюстрирует следующая строка кода:

```
statement := "SELECT * FROM emp WHERE emp_name = '" + empName + "';"
```

Если переменная empName получает значение из поля формы веб-приложения, то атакующий может ввести в поле empName следующий код:

```
' or '1'='1
```

Если код веб-приложения не исключит символ ', он будет включен в SQL-выражение "как есть", что породит следующее новое SQL-выражение:

```
SELECT * FROM emp WHERE emp_name = '' or '1'='1';
```

При исполнении этого кода он возвратит все данные из таблицы emp, поскольку выражение '1'='1' в конструкции WHERE всегда имеет значение "истина". Таким образом атакующий успешно получит данные обо всех сотрудниках в базе данных.

Комментарии и выражения в SQL

Другая распространенная разновидность атаки заключается в злонамеренной инъекции в SQL-выражение символа комментария, который блокирует исполнение остальной части запроса. В качестве такой инъекции могут быть использованы следующие три типа SQL-комментариев:

```
' or '1'='1' -- '
' or '1'='1' ({ '
' or '1'='1' /* '
```

Любое из этих трех выражений, злонамеренно введенное в SQL-код, блокирует остальную часть текущего запроса.
Кроме того, атакующий может добавить злонамеренные SQL-выражения в конец существующего выражения. Например, указанное ниже значение empName приведет к удалению таблицы emp. Если API-интерфейс допускает ввод нескольких выражений, также будут удалены все данные из таблицы userinfo.

```
a';DROP TABLE emp; SELECT * FROM userinfo WHERE 't' = 't
```

При вводе такой строки итоговое SQL-выражение будет иметь вид:

```
SELECT * FROM emp  WHERE emp_name = 'a';DROP TABLE emp; SELECT * FROM userinfo WHERE 't' = 't';
```

Некорректная обработка типов

Второй тип атаки с использованием SQL-инъекций имеет место, когда вводимые пользователем данные не проверяются на соответствие допустимым типам. Например, предположим, что программист не предусмотрел валидацию вводимых пользователем данных для числового поля, показанного в следующем примере:

```
statement := "SELECT * FROM userinfo WHERE id = " + id_var + ";"
```

Принятое значение `id_var` является числовым, однако перед отправкой этого значения в SQL-запрос не производится никакой валидации. Если переменная `id_var` связана с формой заявления, то атакующий сможет задать ее следующим образом:

```
1;DROP TABLE users, yielding the following SQL:

SELECT * FROM userinfo WHERE id=1;DROP TABLE users;
```

В случае успешной передачи этого выражения оно заставит сервер SQL-базы данных удалить таблицу `users` из базы данных.

Предотвращение атак на основе SQL-инъекций

Когда переменная связывания передается в качестве аргумента в подготовленное выражение, JDBC-драйвер автоматически экранирует (escape) эту переменную. В полученных при этом строковых данных эта переменная трактуется как пользовательские данные и не может быть интерпретирована сервером SQL-базы данных как SQL-выражение. Поэтому любые предоставленные пользователями данные должны обрабатываться подобным образом перед их добавлением в SQL-выражение. В данном примере показано добавление переменной связывания user ID к SQL-выражению.

Подготовленные выражения с переменными связывания

```
String selectStatement = "SELECT * FROM User WHERE userId = ? ";
PreparedStatement prepStmt = con.prepareStatement(selectStatement);
prepStmt.setString(1, userId);
ResultSet rs = prepStmt.executeQuery();
```

Метод setString объекта prepStmt экранирует строку userID и добавляет ее в SQL-выражение. Любые злонамеренные SQL-выражения, переданные извне посредством переменной userID, в обработанной таким способом версии входной информации будут представлены в неисполняемом виде.

Использование переменных связывания в SQL-выражениях позволяет повысить показатели производительности исполнения SQL-кода на величину до 30%. Кроме того, переменные связывания – это хорошо известный способ защиты от атак с использованием SQL-инъекций.