

此文档为《Java代码审计零基础入门到项目实战》配套教材，由【闪石星曜CyberSecurity】出品。

请勿对外泄露，一经发现严肃处理！

课程学习中有任何疑问，可添加好友 Power_7089 寻求帮助，为你答疑解惑。

本节讲述JavaWeb代码审计中存在SQL注入漏洞的情况。

SQL注入漏洞是对数据库进行的一种攻击方式。

其主要形成方式是在数据交互中，前端数据通过后台在对数据库进行操作时，由于没有做好安全防护，导致攻击者将恶意代码拼接到请求参数中，被当做SQL语句的一部分进行执行，最终导致数据库被攻击。

可以说所有可以涉及到数据库增删改查的系统功能点都有可能存在SQL注入漏洞。

虽然现在针对SQL注入的防护层出不穷，但大多情况下由于开发人员的疏忽或特定的使用场景，还是会存在SQL注入漏洞的代码。

在 [第一阶段基础 - 1.6 Java数据库操作](#)，我们学习了通过原生的JDBC和主流的Mybatis这两种方式链接操作数据库，下面我们通过示例代码讲讲这两种方式存在SQL注入的情况。

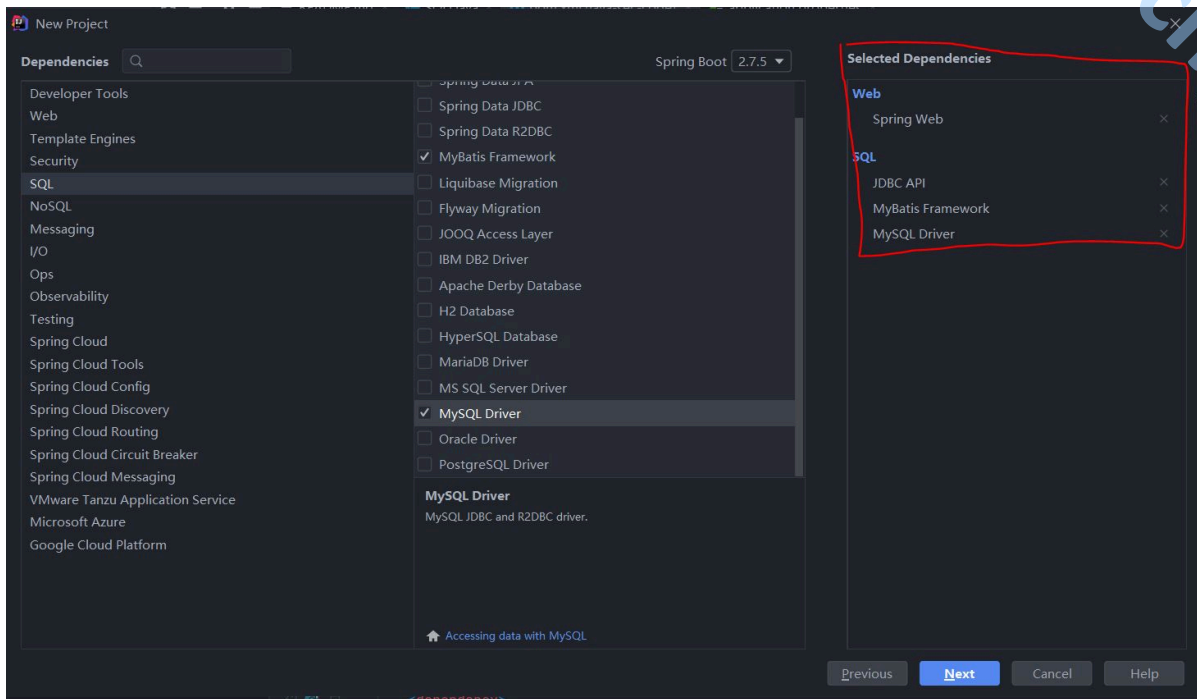
零基础的朋友建议学习过 [第一阶段基础 - 1.6 Java数据库操作](#) 后，再学习以下内容。

零、创建工程

1、新建工程

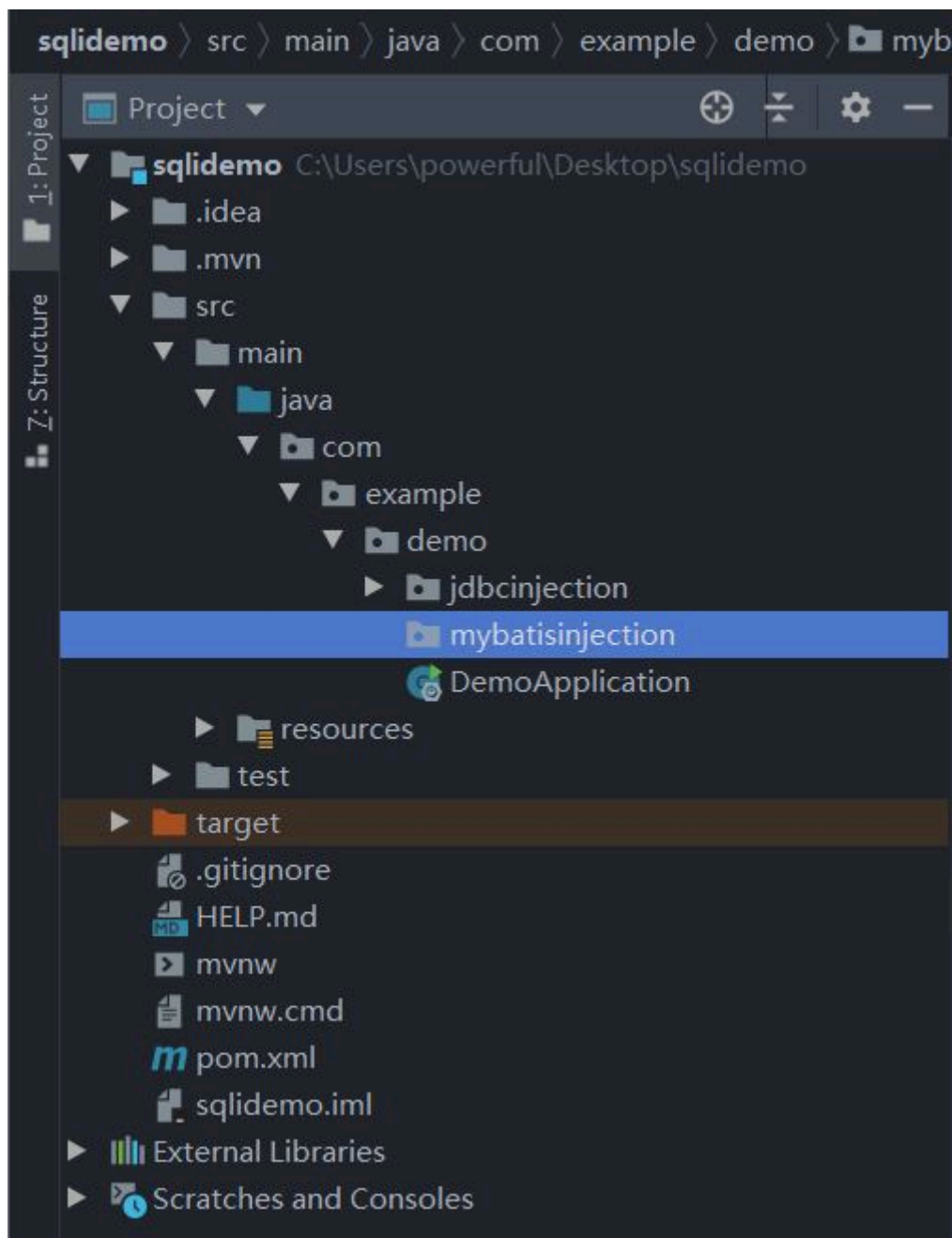
老规矩，创建一个名为 `sqliDemo` 的SpringBoot工程。用于下面示例代码的调试与演示。

- ①、打开IDEA，选择 `Create New Porject`。
- ②、左侧选择 `Spring Initializr`，配置默认即可，点击Next。
- ③、在 `Spring Initializr Project Settings` 页面，将 `Java Version` 设置为8，将 `Type` 选择为 `Maven Project` 其他配置项默认即可，点击Next。
- ④、在依赖选项界面，我们选择 `web -> Spring web`，`SQL -> JDBC API`，`SQL -> Mybatis Framework`，`SQL -> Mysql Driver`，注意，一共需要勾选四个依赖。最后Next。如下图所示：



⑤、最后一步给项目起个名字，就叫 `sqlDemo` 吧。其他默认即可。最后点击Finish。

⑥、在 `src/main/java/com/example/demo/` 下新建两个package，分别名为 `jdbcinjection` 和 `mybatisinjection`，最终目录结构如下图所示：



2、创建数据库

①、启动Mysql，使用cmd进入mysql。创建一个名为 sqlidemo 的数据库，如下图所示：

```
CREATE DATABASE sqlidemo;
```

```
mysql> CREATE DATABASE sqlidemo;  
Query OK, 1 row affected (0.00 sec)  
  
mysql> 
```

②、先切换使用 sqlidemo 数据库。然后创建 users 数据表，如下图所示：

USE `sqlidemo`;
CREATE TABLE IF NOT EXISTS `users`(
 `id` INT UNSIGNED AUTO_INCREMENT,
 `username` VARCHAR(255) NOT NULL,
 `password` VARCHAR(255) NOT NULL,
 PRIMARY KEY (`id`)
)ENGINE=InnoDB DEFAULT CHARSET=utf8;

```
mysql> use sqlidemo;  
Database changed  
mysql> CREATE TABLE IF NOT EXISTS `users`(  
  ->   `id` INT UNSIGNED AUTO_INCREMENT,  
  ->   `username` VARCHAR(255) NOT NULL,  
  ->   `password` VARCHAR(255) NOT NULL,  
  ->   PRIMARY KEY (`id`)  
  -> )ENGINE=InnoDB DEFAULT CHARSET=utf8;  
Query OK, 0 rows affected (0.02 sec)  
  
mysql> 
```

③、向 info 数据表中添加具体数据，如下图所示：

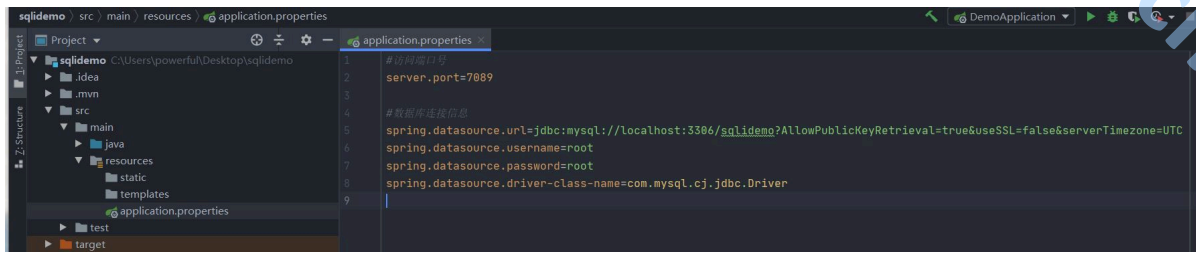
```
INSERT INTO `users` VALUES (1, 'admin', 'admin');  
INSERT INTO `users` VALUES (2, 'power7089', 'power7089');
```

```
mysql> INSERT INTO `users` VALUES (1, 'admin', 'admin');  
Query OK, 1 row affected (0.00 sec)  
  
mysql> INSERT INTO `users` VALUES (2, 'power7089', 'power7089');  
Query OK, 1 row affected (0.00 sec)
```

3、修改配置文件

打开 `src/main/resources/application.properties` 配置文件，将以下数据库连接信息添加至配置文件中，如下图所示：

```
#访问端口号  
server.port=7089  
  
#数据库连接信息  
spring.datasource.url=jdbc:mysql://localhost:3306/sqlidemo?  
AllowPublicKeyRetrieval=true&useSSL=false&serverTimezone=UTC  
spring.datasource.username=root  
spring.datasource.password=root  
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
```



一、Jdbc 中 SQL 注入

1、动态拼接

SQL语句动态拼接导致的SQL注入漏洞是先前最为常见的场景。

其主要原因是后端代码将前端获取的参数动态直接拼接到SQL语句中使用 `java.sql.Statement` 执行SQL语句从而导致SQL注入漏洞的出现。

在这里关键点有两个：①、动态拼接参数。②、使用 `java.sql.Statement` 执行SQL语句。

1.1、java.sql.Statement

Statement 对象用于执行一条静态的 SQL 语句并获取它的结果。

`createStatement()`：创建一个 Statement 对象，之后可使用 `executeQuery()` 方法执行SQL语句。

`executeQuery(String sql)` 方法：执行指定的 SQL 语句，返回单个 ResultSet 对象。

官方文档：

```
java.sql.Statement:
https://docs.oracle.com/javase/7/docs/api/java/sql/Statement.html
createStatement() 方法:
https://docs.oracle.com/javase/8/docs/api/java/sql/Connection.html#createStatement--
executeQuery() 方法:
https://docs.oracle.com/javase/8/docs/api/java/sql/Statement.html#executeQuery-java.lang.String-
```

1.2、示例代码

在 `src/main/java/com/example/demo/jdbcinjection` 下新建一个名为 `JdbcDynamicController` 的 Java Class，并键入以下代码，最终如下图所示（截图不全请自行敲写下面代码进行调试）：

```
package com.example.demo.jdbcinjection;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

import java.sql.*;

/**
 * 编号7089
 * 动态拼接
 */
```

```

* http://localhost:7087/sqli/jdbc/dynamic?id=1
*/
@RestController
@RequestMapping("/sqli")
public class JdbcDynamicController {

    private static String driver = "com.mysql.jdbc.Driver";

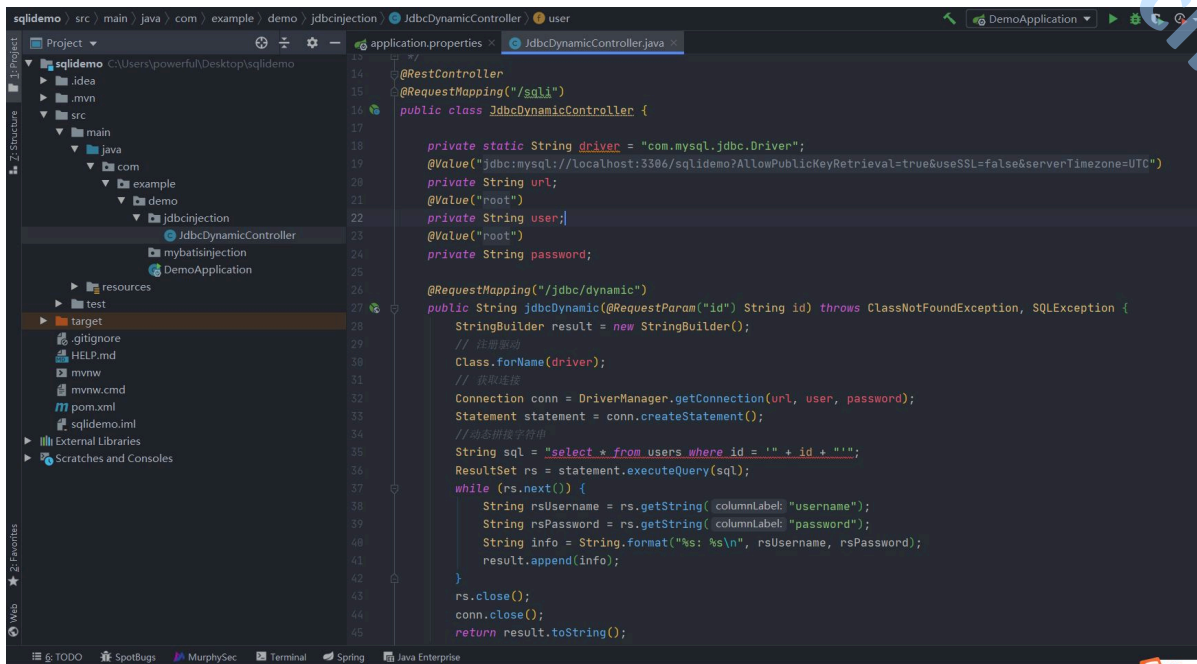
    @Value("${spring.datasource.url}")
    private String url;

    @Value("${spring.datasource.username}")
    private String user;

    @Value("${spring.datasource.password}")
    private String password;

    @RequestMapping("/jdbc/dynamic")
    public String jdbcDynamic(@RequestParam("id") String id) throws
ClassNotFoundException, SQLException {
        StringBuilder result = new StringBuilder();
        // 注册驱动
        Class.forName(driver);
        // 获取连接
        Connection conn = DriverManager.getConnection(url, user, password);
        Statement statement = conn.createStatement();
        //动态拼接字符串
        String sql = "select * from users where id = '" + id + "'";
        ResultSet rs = statement.executeQuery(sql);
        while (rs.next()) {
            String rsUsername = rs.getString("username");
            String rsPassword = rs.getString("password");
            String info = String.format("%s: %s\n", rsUsername, rsPassword);
            result.append(info);
        }
        rs.close();
        conn.close();
        return result.toString();
    }
}

```



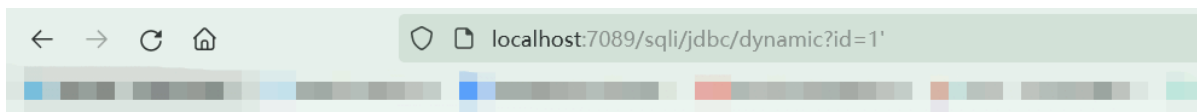
1.3、代码解读

- ①、根据上图行数为基准，第14到33就不多说了，设置连接数据库信息并连接数据库。
- ②、第33行，创建了一个 Statement 对象。
- ③、第35行，动态拼接了id。
- ④、第36行，使用 executeQuery() 执行了SQL语句。
- ⑤、第37行到结束，做了获取信息，关闭连接等操作。

通过前面的描述关键点在②③④，大家可以运行程序，访问

<http://localhost:7089/sqli/jdbc/dynamic?id=1> 自行调试。

下图使用单引号（'）：



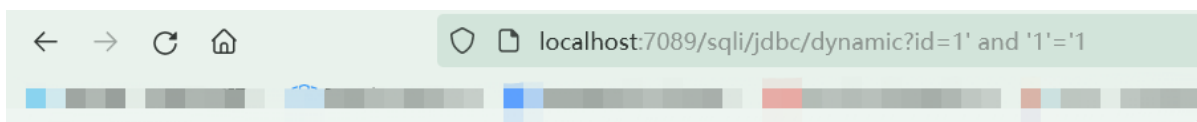
Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Fri Oct 28 14:58:34 CST 2022

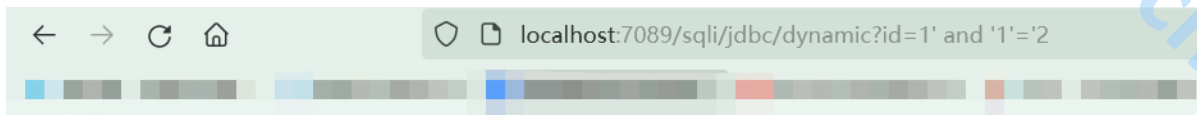
There was an unexpected error (type=Internal Server Error, status=500).

下图使用 ' and '1'='1'：



admin: admin

下图使用 ' and '1'='2'：



2、错误的预编译

在动态拼接中是使用Statement执行SQL语句。如果使用 PreparedStatement 预编译参数化查询是能够有效防止SQL注入的。

但如果没有正确的使用 PreparedStatement 预编译还是会存在SQL注入风险的。

1.1、java.sql.PreparedStatement

PreparedStatement是继承Statement的子接口。

PreparedStatement 会对SQL语句进行预编译，不论输入什么，经过预编译后全都以字符串来执行SQL语句。

PreparedStatement 会先使用 ? 作为占位符将 SQL 语句进行预编译，确定语句结构，再传入参数进行执行查询。如下述代码：

```
String sql = "select * from users where username = ?";
PreparedStatement preparedStatement = conn.prepareStatement(sql);
preparedStatement.setString(1, username);
```

官方文档：

```
https://docs.oracle.com/javase/7/docs/api/java/sql/PreparedStatement.html
https://docs.oracle.com/javase/tutorial/jdbc/basics/prepared.html
```

1.2、示例代码

但如果没有正确的使用 PreparedStatement 预编译还是会存在SQL注入风险的。

简单来说，可能由于开发人员疏忽或经验不足等原因，虽然使用了预编译 PreparedStatement，但没有根据标准流程对参数进行标记，依旧使用了动态拼接SQL语句的方式，进而造成SQL注入漏洞。

在 src\main\java\com\example\demo\jdbcinjection 下新建一个名为 JdbcPreparedStatement 的 Java Class，并键入以下代码，最终如下图所示（截图不全请自行敲写下面代码进行调试）：

```
package com.example.demo.jdbcinjection;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

import java.sql.*;
```



```

/**
 * 编号7089
 * 错误的预编译
 * http://localhost:7089/sqli/jdbc/preparestaement?username=admin
 * 安全代码
 * http://localhost:7089/sqli/jdbc/sec?username=admin
 */
@RestController
@RequestMapping("/sqli")
public class JdbcPrepareStatement {
    private static String driver = "com.mysql.jdbc.Driver";

    @Value("${spring.datasource.url}")
    private String url;
    @Value("${spring.datasource.username}")
    private String user;
    @Value("${spring.datasource.password}")
    private String password;

    @RequestMapping("/jdbc/sec")
    public String jdbcsec(@RequestParam("username") String username) throws
        SQLException, ClassNotFoundException {

        StringBuilder result = new StringBuilder();
        Class.forName(driver);
        Connection conn = DriverManager.getConnection(url, user, password);
        // 安全代码
        String sql = "select * from users where username = ?";
        PreparedStatement preparestatement = conn.prepareStatement(sql);
        preparestatement.setString(1, username);
        ResultSet rs = preparestatement.executeQuery();
        while (rs.next()) {
            String resUsername = rs.getString("username");
            String resPassword = rs.getString("password");
            String info = String.format("%s: %s\n", resUsername, resPassword);
            result.append(info);
        }
        rs.close();
        conn.close();
        return result.toString();
    }

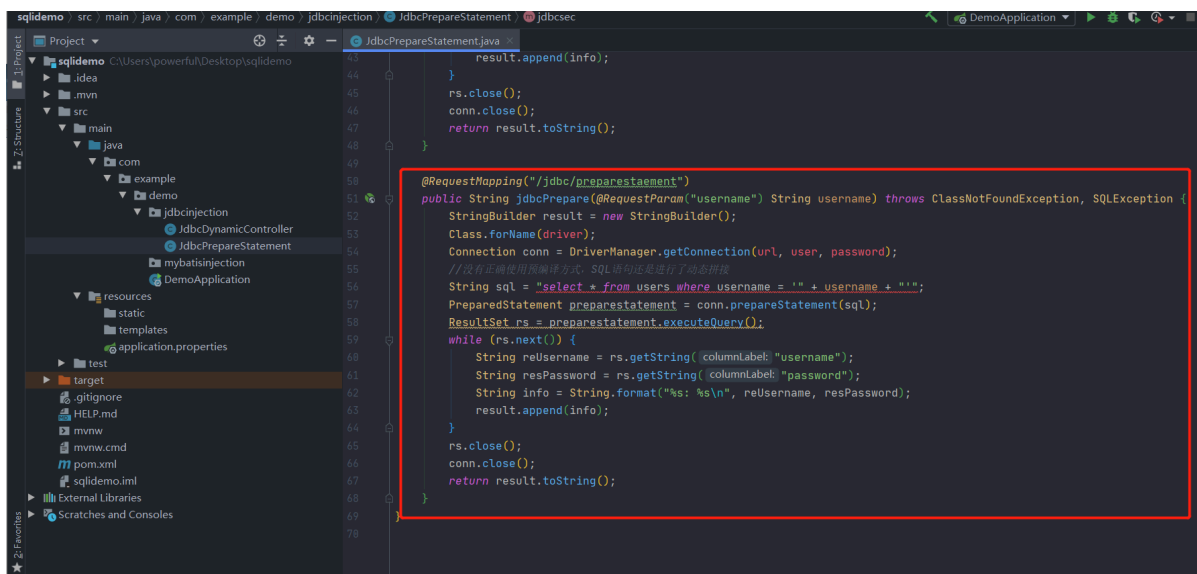
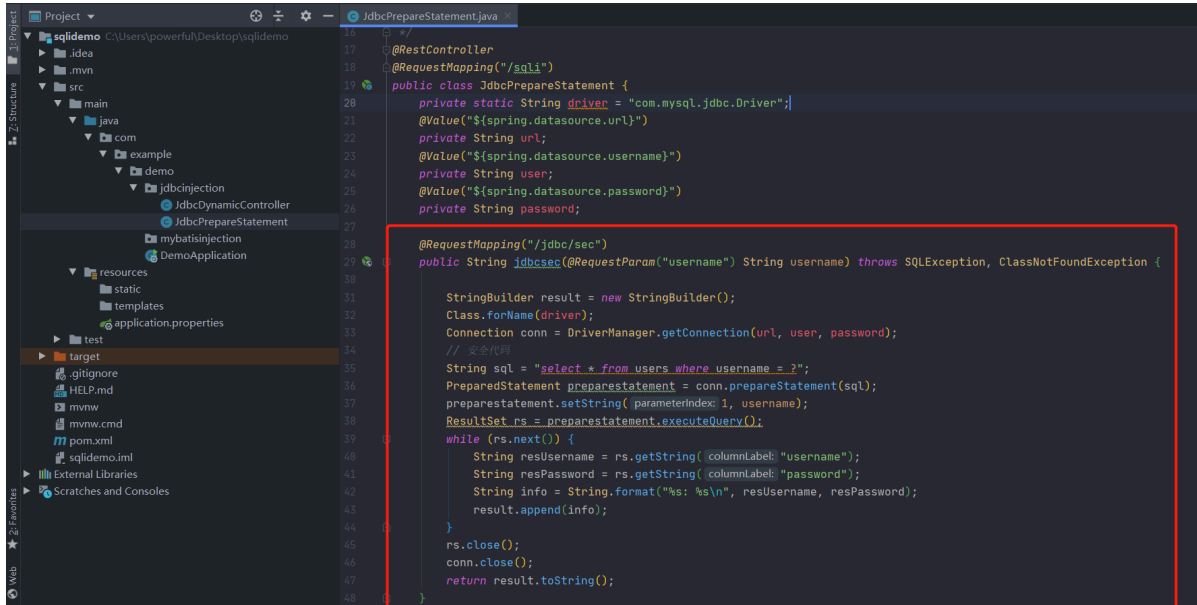
    @RequestMapping("/jdbc/preparestaement")
    public String jdbcPrepare(@RequestParam("username") String username) throws
        ClassNotFoundException, SQLException {
        StringBuilder result = new StringBuilder();
        Class.forName(driver);
        Connection conn = DriverManager.getConnection(url, user, password);
        //没有正确使用预编译方式，SQL语句还是进行了动态拼接
        String sql = "select * from users where username = '" + username + "'";
        PreparedStatement preparestatement = conn.prepareStatement(sql);
        ResultSet rs = preparestatement.executeQuery();
        while (rs.next()) {
            String reUsername = rs.getString("username");
            String resPassword = rs.getString("password");

```

```

        String info = String.format("%s: %s\n", reUsername, resPassword);
        result.append(info);
    }
    rs.close();
    conn.close();
    return result.toString();
}
}

```



1.3、代码解读

①、根据上图行数为准，关注第35到38行，为预编译 PreparedStatement 正确使用方式，防止了SQL注入漏洞。

```

String sql = "select * from users where username = ?";
PreparedStatement preparedStatement = conn.prepareStatement(sql);
preparedStatement.setString(1, username);
ResultSet rs = preparedStatement.executeQuery();

```

②、根据上图行数为准，关注第56到58行，虽然使用了预编译 PreparedStatement 方式处理SQL语句，但由于SQL语句依旧是动态拼接形式，从而造成了SQL注入漏洞。

```
String sql = "select * from users where username = '" + username + "'";
PreparedStatement preparestatement = conn.prepareStatement(sql);
ResultSet rs = preparestatement.executeQuery();
```

访问 <http://localhost:7089/sqli/jdbc/preparestatement?username=admin> 和 <http://localhost:7089/sqli/jdbc/sec?username=admin> 自行调试观察结果。

3、order by注入

在SQL语句中，`order by` 语句用于对结果集进行排序。`order by` 语句后面需要是字段名或者字段位置。

在使用 `PreparedStatement` 预编译时，会将传递任意参数使用单引号包裹进而变为了字符串。

如果使用预编译方式执行 `order by` 语句，设置的字段名会被数据库认为是字符串，而不在是字段名。

因此，在使用 `order by` 时，就不能使用 `PreparedStatement` 预编译了。

1.1、示例代码

在 `src/main/java/com/example/demo/jdbcinjection` 下新建一个名为 `jdbcordeby` 的Java Class，并键入以下代码，最终如下图所示（截图不全请自行敲写下面代码进行调试）：

```
package com.example.demo.jdbcinjection;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

import java.sql.*;

/**
 * 编号7089
 * order by 注入
 * http://localhost:7089/sqli/jdbc/orderby?id=1
 */
@RestController
@RequestMapping("/sqli")
public class Jdbcordeby {
    private static String driver = "com.mysql.jdbc.Driver";
    @Value("${spring.datasource.url}")
    private String url;
    @Value("${spring.datasource.username}")
    private String user;
    @Value("${spring.datasource.password}")
    private String password;

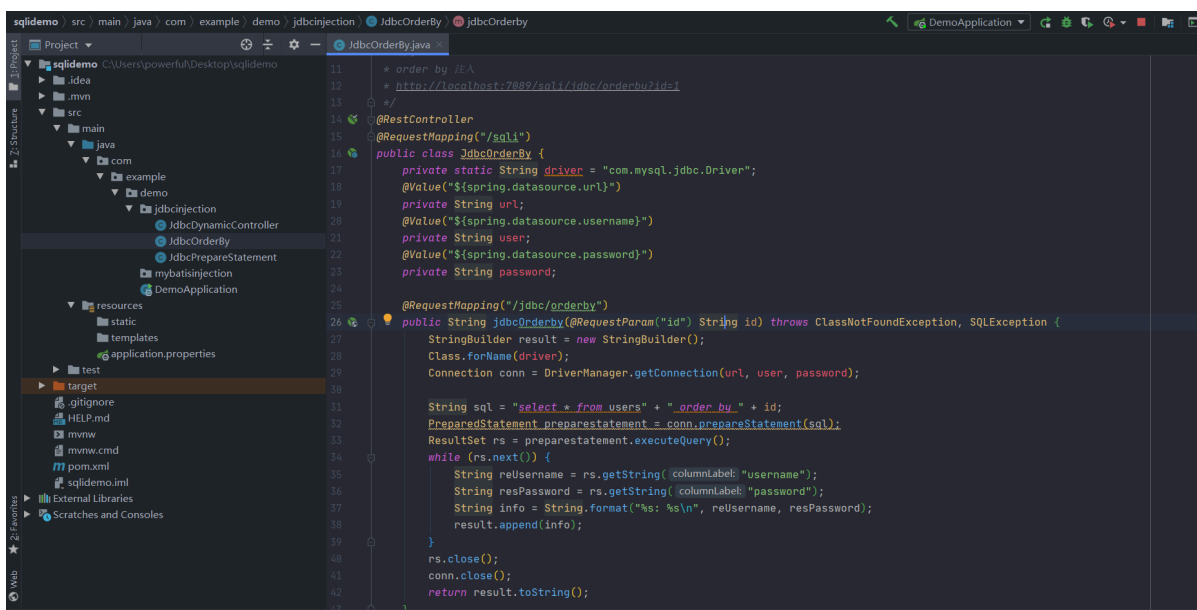
    @RequestMapping("/jdbc/orderby")
    public String jdbcOrderby(@RequestParam("id") String id) throws
        ClassNotFoundException, SQLException {
        StringBuilder result = new StringBuilder();
        Class.forName(driver);
```

```

Connection conn = DriverManager.getConnection(url, user, password);

String sql = "select * from users" + " order by " + id;
PreparedStatement preparestatement = conn.prepareStatement(sql);
ResultSet rs = preparestatement.executeQuery();
while (rs.next()) {
    String reUsername = rs.getString("username");
    String resPassword = rs.getString("password");
    String info = String.format("%s: %s\n", reUsername, resPassword);
    result.append(info);
}
rs.close();
conn.close();
return result.toString();
}
}

```



1.2、代码解读

①、根据上图行数为准，关注第31到33行，SQL语句使用了 `order by` 排序，因此无法对参数进行预编译。进而造成了SQL注入漏洞。如下图所示：

```

String sql = "select * from users" + " order by " + id;
PreparedStatement preparestatement = conn.prepareStatement(sql);
ResultSet rs = preparestatement.executeQuery();

```

```
GET parameter 'id' is vulnerable. Do you want to keep testing the others (if any)? [y/N] N
sqlmap identified the following injection point(s) with a total of 308 HTTP(s) requests:
---
Parameter: id (GET)
  Type: boolean-based blind
  Title: Boolean-based blind - Parameter replace (original value)
  Payload: id=(SELECT (CASE WHEN (2206=2206) THEN 1 ELSE (SELECT 4901 UNION SELECT 8747) END))

  Type: time-based blind
  Title: MySQL >= 5.0.12 AND time-based blind (query SLEEP)
  Payload: id=1 AND (SELECT 2020 FROM (SELECT(SLEEP(5)))OQMX)
---
[16:16:17] [INFO] the back-end DBMS is MySQL
back-end DBMS: MySQL >= 5.0.12
[16:16:18] [WARNING] HTTP error codes detected during run:
500 (Internal Server Error) - 289 times
[16:16:18] [INFO] fetched data logged to text files under 'C:\Users\powerful\AppData\Local\sqlmap\output\localhost'
[16:16:18] [WARNING] your sqlmap version is outdated

[*] ending @ 16:16:18 /2022-10-28/

C:\Users\powerful> sqlmap-master
GoodLuckToday$$$ py -s sqlmap.py -u localhost:7089/sqli/jdbc/orderby?id=1 --batch
```

访问 <http://localhost:7089/sqli/jdbc/orderby?id=1> 自行调试观察结果。

二、Mybatis

MyBatis 是一款优秀的持久层框架，它支持自定义 SQL、存储过程以及高级映射。MyBatis 免除了几乎所有的 JDBC 代码以及设置参数和获取结果集的工作。MyBatis 可以通过简单的 XML 或注解来配置和映射原始类型、接口和 Java POJO（Plain Old Java Objects，普通老式 Java 对象）为数据库中的记录。

Mybatis中文文档：<https://mybatis.org/mybatis-3/zh/index.html>

1、Mybatis中#{ }和\${ }区别

在Mybatis中拼接SQL语句有两种方式：一种是占位符 `#{ }`，另一种是拼接符 `${ }`。

占位符 `#{ }`：对传入的参数进行预编译转义处理。类似 JDBC 中的 `PreparedStatement`。

比如：`select * from user where id = #{number}`，如果传入数值为1，最终会被解析成 `select * from user where id = "1"`。

拼接符 `${ }`：对传入的参数不做处理，直接拼接，进而会造成SQL注入漏洞。

比如：比如：`select * from user where id = ${number}`，如果传入数值为1，最终会被解析成 `select * from user where id = 1`。

`#{ }` 可以有效防止SQL注入漏洞。`${ }` 则无法防止SQL注入漏洞。

因此在对JavaWeb整合Mybatis系统进行代码审计时，应着重审计SQL语句拼接的地方。

除非开发人员的粗心对拼接语句使用了 `${ }` 方式造成的SQL注入漏洞。

在Mybatis中有几种场景是不能使用预编译方式的，比如：`order by`、`in`，`like`。

下面针对这几个场景以实例代码进行讲解。

2、示例代码

在这之前，我们先在项目中编写下面练习必要代码。

具体漏洞代码解读会放在对应的小节中。

（注意：以下代码内容基于 [第一阶段基础 - 1.6 Java数据库操作](#) 内容作了一些改动，如果在编写过程中遇见困难，建议先回顾学习。）

①、在 `src.main.java.com.example.demo.mybatisinjection` 包下新建一个Java Class，名为 `User`，这是一个实体类，和数据表做下映射，键入以下代码，最终如下图所示：

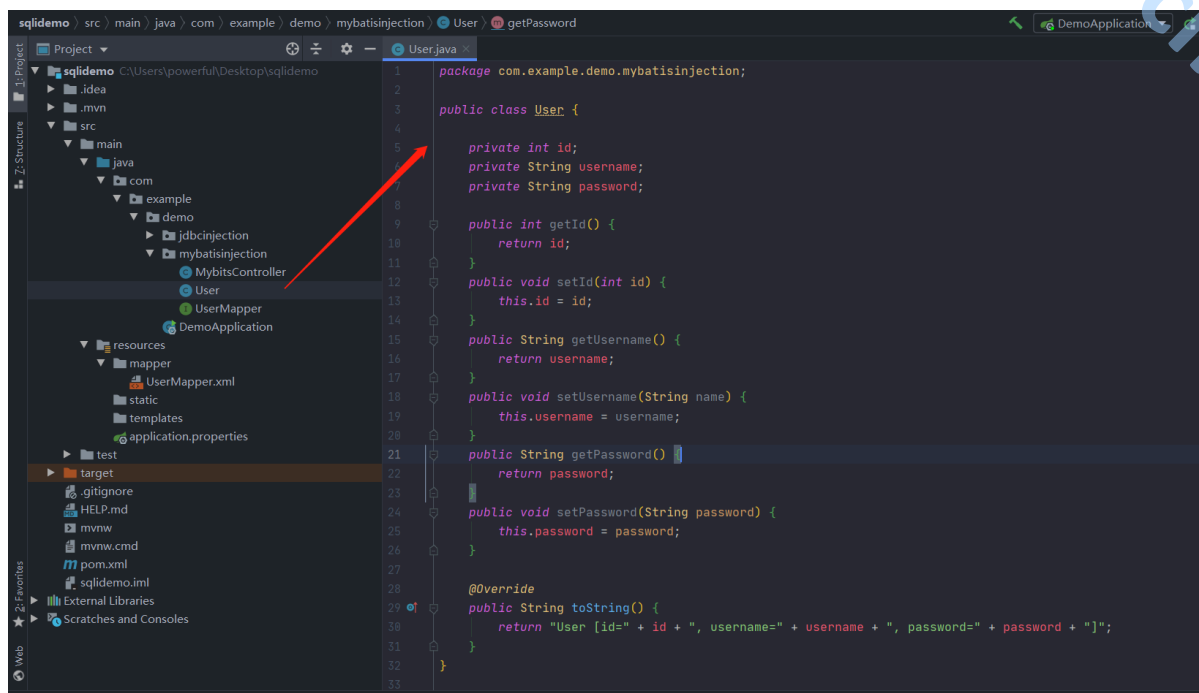
```
package com.example.demo.mybatisinjection;

public class User {

    private int id;
    private String username;
    private String password;

    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getUsername() {
        return username;
    }
    public void setUsername(String name) {
        this.username = name;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }

    @Override
    public String toString() {
        return "User [id=" + id + ", username=" + username + ", password=" +
password + "]\n";
    }
}
```



②、在 `src.main.java.com.example.demo.mybatisinjection` 文件下新建一个名为 `UserMapper` 的 Java Interface，键入以下代码，最终如下图所示：

```
package com.example.demo.mybatisinjection;

import org.apache.ibatis.annotations.Mapper;
import org.apache.ibatis.annotations.Param;
import org.apache.ibatis.annotations.Select;
import org.springframework.web.bind.annotation.RequestParam;

import java.util.List;

/**
 * 编号7089
 */
@Mapper
public interface UserMapper {

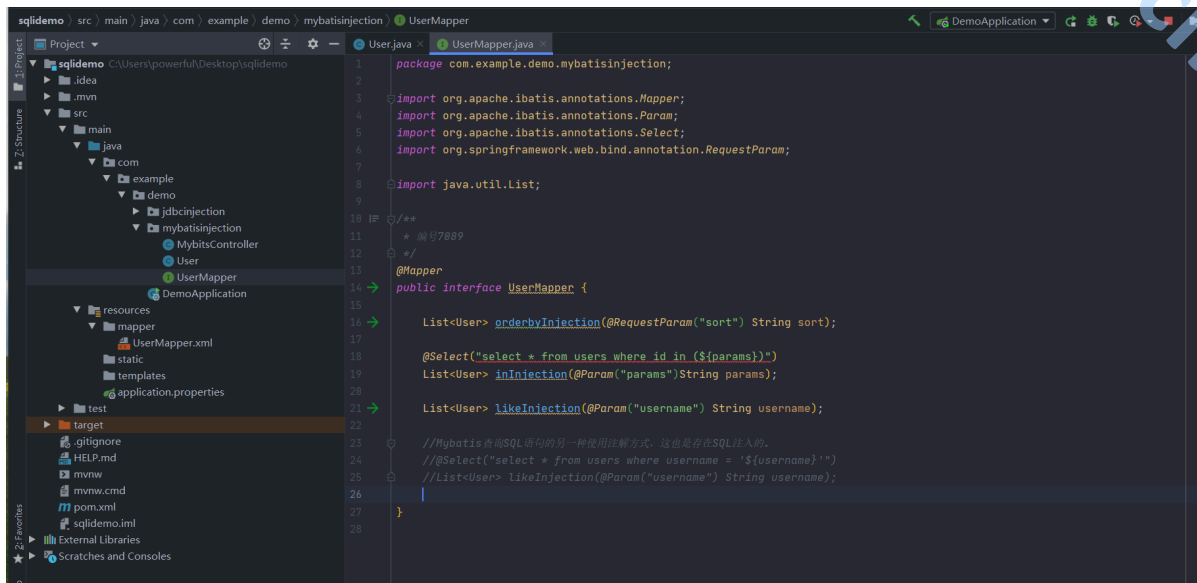
    List<User> orderByInjection(@RequestParam("sort") String sort);

    @Select("select * from users where id in (${params})")
    List<User> inInjection(@Param("params")String params);

    List<User> likeInjection(@Param("username") String username);

    //Mybatis查询SQL语句的另一种使用注解方式，这也是存在SQL注入的。
    //@Select("select * from users where username = '${username}'")
    //List<User> likeInjection(@Param("username") String username);

}
```

③、在 `src.main.resources` 文件下先新建名为 `mapper` 的文件夹，再新建一个名为 `UserMapper.xml` 文件，与 `dao` 层的 `UserMapper` 做好映射绑定，键入以下代码，最终如下图所示：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd">

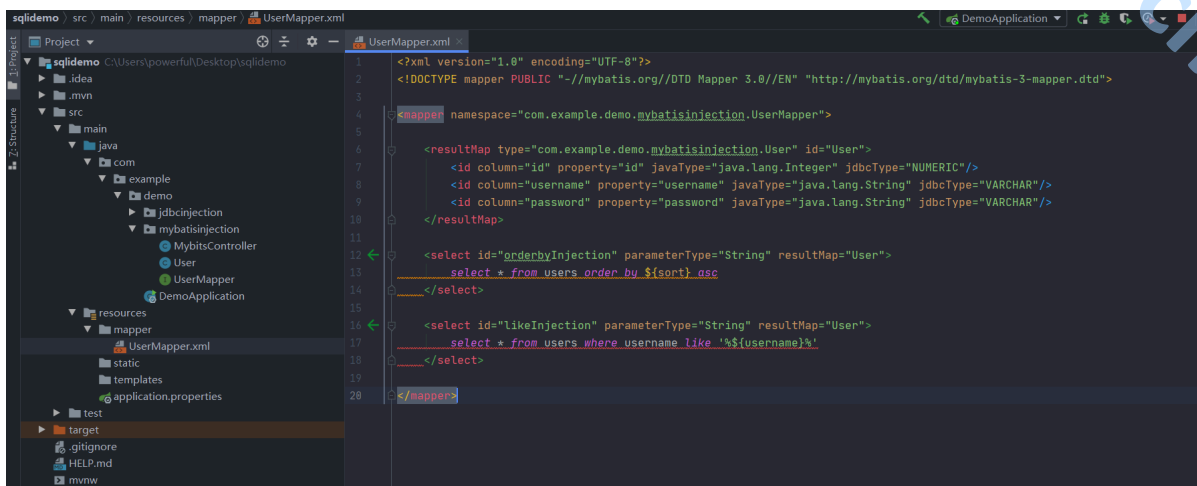
<mapper namespace="com.example.demo.mybatisinjection.UserMapper">

    <resultMap type="com.example.demo.mybatisinjection.User" id="User">
        <id column="id" property="id" javaType="java.lang.Integer"
jdbcType="NUMERIC"/>
        <id column="username" property="username" javaType="java.lang.String"
jdbcType="VARCHAR"/>
        <id column="password" property="password" javaType="java.lang.String"
jdbcType="VARCHAR"/>
    </resultMap>

    <select id="orderByInjection" parameterType="String" resultMap="User">
        select * from users order by ${sort} asc
    </select>

    <select id="likeInjection" parameterType="String" resultMap="User">
        select * from users where username like '${username}%'
    </select>

</mapper>
```



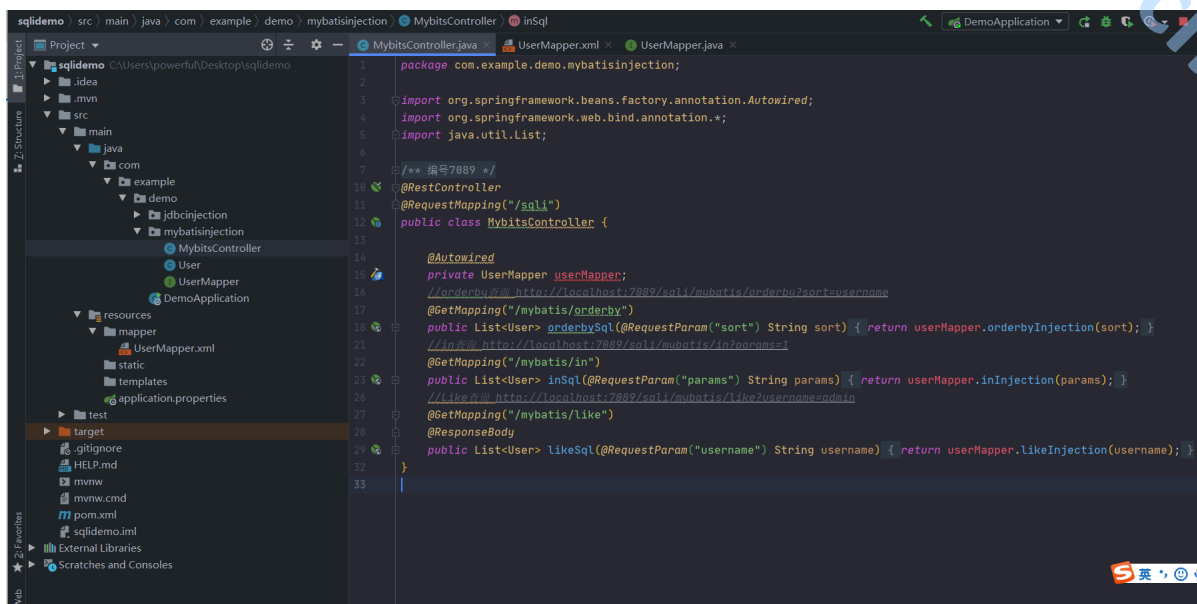
④、在src.main.java.com.example.demo.mybatisinjection文件下新建一个名为MybatisController的Java class，键入以下代码，最终如下图所示：

```
package com.example.demo.mybatisinjection;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;
import java.util.List;

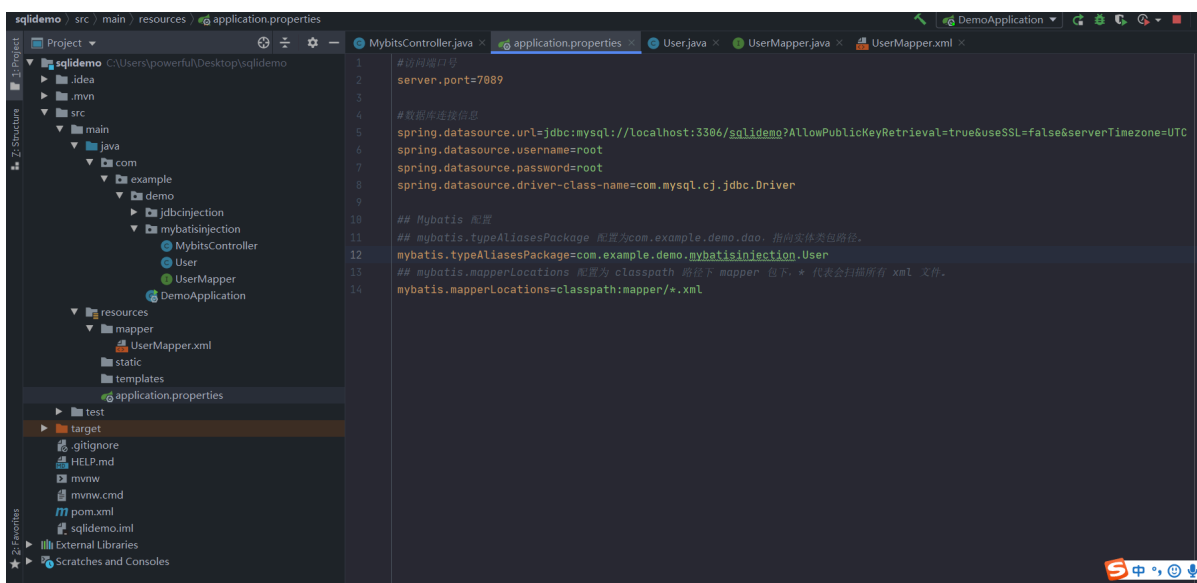
/**
 * 编号7089
 */
@RestController
@RequestMapping("/sqli")
public class MybitsController {

    @Autowired
    private UserMapper userMapper;
    //orderBy查询 http://localhost:7089/sqli/mybatis/orderby?sort=username
    @GetMapping("/mybatis/orderby")
    public List<User> orderBySql(@RequestParam("sort") String sort) {
        return userMapper.orderbyInjection(sort);
    }
    //in查询 http://localhost:7089/sqli/mybatis/in?params=1
    @GetMapping("/mybatis/in")
    public List<User> inSql(@RequestParam("params") String params) {
        return userMapper.inInjection(params);
    }
    //Like查询 http://localhost:7089/sqli/mybatis/like?username=admin
    @GetMapping("/mybatis/like")
    @ResponseBody
    public List<User> likesql(@RequestParam("username") String username){
        return userMapper.likeInjection(username);
    }
}
```



⑤、最后，在 `src/main/resources/application.properties` 配置文件中添加以下代码，最终如下图所示：

```
## Mybatis 配置
## mybatis.typeAliasesPackage 配置为com.example.demo.dao，指向实体类包路径。
mybatis.typeAliasesPackage=com.example.demo.dao
## mybatis.mapperLocations 配置为 classpath 路径下 mapper 包下，* 代表会扫描所有 xml 文件。
mybatis.mapperLocations=classpath:mapper/*.xml
```




3、order by 注入

ORDER BY语句：用于对查询结果的排序，asc为升序，desc为降序。默认为升序。

比如：`select * from users order by username asc;`

与JDBC预编译中order by注入一样，在 `order by` 语句后面需要是字段名或者字段位置。因此也不能使用Mybatis中预编译的方式。

在 `src/main/resources/mapper/UserMapper.xml` 中第13行是order by查询，我们可以看到使用了拼接符`${}`，从而造成了SQL注入漏洞，如下图所示：



```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
3
4  <mapper namespace="com.example.demo.mybatisinjection.UserMapper">
5
6      <resultMap type="com.example.demo.mybatisinjection.User" id="User">
7          <id column="id" property="id" javaType="java.lang.Integer" jdbcType="NUMERIC"/>
8          <id column="username" property="username" javaType="java.lang.String" jdbcType="VARCHAR"/>
9          <id column="password" property="password" javaType="java.lang.String" jdbcType="VARCHAR"/>
10     </resultMap>
11
12     <select id="orderByInjection" parameterType="String" resultMap="User">
13         select * from users order by ${sort} asc
14     </select>
15
16     <select id="likeInjection" parameterType="String" resultMap="User">
17         select * from users where username like '${username}%'
18     </select>
19
20 </mapper>

```

查看 `src\main\java\com\example\demo\mybatisinjection\MybitsController.java`，从 Controller 层我们可以确定 URL 访问路径和请求参数，为 `/sql/mybatis/orderby?sort=xxx`。

最终拼接为 `http://localhost:7089/sql/mybatis/orderby?sort=admin`。

大家自行访问调试 SQL 注入漏洞。

4、in 注入

IN 语句：常用于 where 表达式中，其作用是查询某个范围内的数据。

比如：`select * from where field in (value1,value2,value3,...);`

如上所示，in 在查询某个范围数据是会用到多个参数，在 Mybatis 中如果直接使用占位符 `#{}` 进行查询会将这些参数看做一个整体，查询会报错。

因此很多开发人员可能会使用拼接符 `${}` 对参数进行查询，从而造成了 SQL 注入漏洞。

比如：`select * from users where id in (${params})`

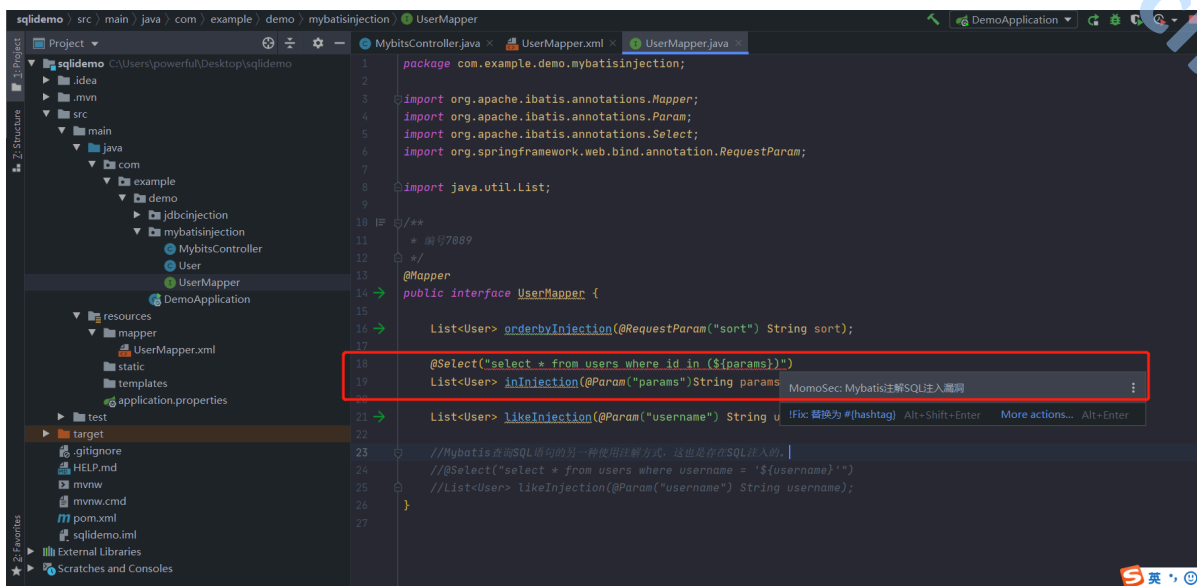
正确的做法是需要使用 `foreach` 配合占位符 `#{}` 实现 IN 查询。比如：

```

<!-- where in 查询场景 -->
<select id="select" parameterType="java.util.List" resultMap="BaseResultMap">
    SELECT *
    FROM user
    WHERE name IN
    <foreach collection="names" item="name" open="(" close=")" separator=",">
        #{name}
    </foreach>
</select>

```

在 `src\main\resources\mapper\UserMapper.xml` 中第 18 行是 in 查询，我们可以看到使用了拼接符 `${}` ，从而造成了 SQL 注入漏洞，如下图所示：



查看 `src/main/java/com/example/demo/mybatisinjection/MybitsController.java`，从Controller层我们可以确定URL访问路径和请求参数，为 `/sql/mybatis/in?params=1`。

最终拼接为 `http://localhost:7089/sql/mybatis/in?params=1`。

大家自行访问调试SQL注入漏洞。

5、like 注入

LIKE语句：在一个字符型字段中检索包含对应子串的。

比如：`select * from users where username like admin`

使用like语句进行查询时如果使用占位符 `#{}` 查询时程序会报错（大家可自行调试）。

比如：`select * from users where username like '%#{username}%'`

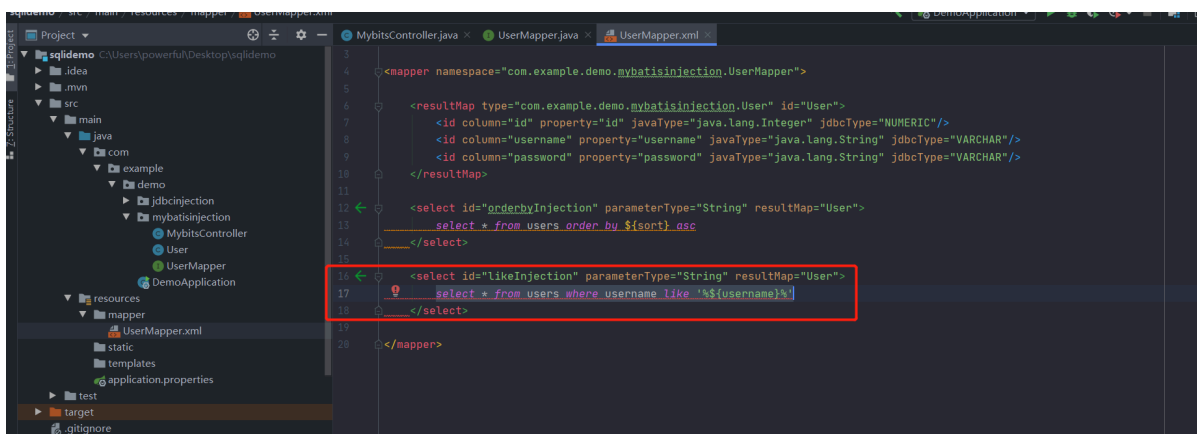
因此经验不足的开发人员可能会使用拼接符 `${}` 对参数进行查询，从而造成了SQL注入漏洞。

比如：`select * from users where username like '${username}%'`

下面代码是正确的做法，可以防止SQL注入漏洞，如下。

```
SELECT * FROM users WHERE name like CONCAT("%", #{name}, "%")
```

在 `src/main/resources/mapper/UserMapper.xml` 中第17行是like查询，可以看到使用了拼接符 `${}`，从而造成了SQL注入漏洞，如下图所示：



查看 `src/main/java/com/example/demo/mybatisinjection/MybitsController.java`, 从 Controller层我们可以确定URL访问路径和请求参数, 为 `/sql/mybatis/like?username=admin`。

最终拼接为 `http://localhost:7089/sql/mybatis/like?username=admin`。

大家自行访问调试SQL注入漏洞。

三、SQL注入漏洞修复

原文: <https://gist.github.com/retanqj/5fd369524a18ab68a4fe7ac5e0d121e8>

3.1、表,字段名称

(Select, Order by, Group by 等)

```
// 插入数据用户可控时, 应使用白名单处理
// example for order by

String orderBy = "{user input}";
String orderByField;
switch (orderBy) {
    case "name":
        orderByField = "name";break;
    case "age":
        orderByField = "age"; break;
    default:
        orderByField = "id";
}
```

3.2、JDBC

```
String name = "foo";

// 一般查询场景
String sql = "SELECT * FROM users WHERE name = ?";
PreparedStatement pre = conn.prepareStatement(sql);
pre.setString(1, name);
ResultSet rs = pre.executeQuery();

// like 模糊查询场景
String sql = "SELECT * FROM users WHERE name like ?";
PreparedStatement pre = conn.prepareStatement(sql);
pre.setString(1, "%" + name + "%");
ResultSet rs = pre.executeQuery();

// where in 查询场景
String sql = "select * from user where id in (";
Integer[] ids = new Integer[]{1,2,3};

StringBuilder placeholdersSql = new StringBuilder(sql);
for(int i=0,size=ids.length;i<size;i++) {
    placeholdersSql.append("?");
}
```

```

        if (i != size-1) {
            placeholdersSql.append(",");
        }
    }
    placeholdersSql.append(")");

    PreparedStatement pre = conn.prepareStatement(placeholdersSql.toString());
    for(int i=0,size=ids.length;i<size;i++) {
        pre.setInt(i+1, ids[i]);
    }
    ResultSet rs = pre.executeQuery();

```

3.3、Spring-JDBC

```

JdbcTemplate jdbcTemplate = new JdbcTemplate(app.dataSource());

// 一般查询场景
String sql = "select * from user where id = ?";
Integer id = 1;
UserDO user = jdbcTemplate.queryForObject(sql,
    BeanPropertyRowMapper.newInstance(UserDO.class), id);

// like 模糊查询场景
String sql = "select * from user where name like ?";
String like_name = "%" + "foo" + "%";
UserDO user = jdbcTemplate.queryForObject(sql,
    BeanPropertyRowMapper.newInstance(UserDO.class), like_name);

// where in 查询场景
NamedParameterJdbcTemplate namedJdbcTemplate = new
    NamedParameterJdbcTemplate(app.dataSource());

MapSqlParameterSource parameters = new MapSqlParameterSource();
parameters.addValue("names", Arrays.asList("foo", "bar"));

String sql = "select * from user where name in (:names)";
List<UserDO> users = namedJdbcTemplate.query(sql, parameters,
    BeanPropertyRowMapper.newInstance(UserDO.class));

```

3.4、Mybatis XML Mapper

```

<!-- 一般查询场景 -->
<select id="select" parameterType="java.lang.String" resultMap="BaseResultMap">
    SELECT *
    FROM user
    WHERE name = #{name}
</select>

<!-- like 查询场景 -->
<select id="select" parameterType="java.lang.String" resultMap="BaseResultMap">
    SELECT *
    FROM user
    WHERE name like CONCAT("%", #{name}, "%")
</select>

```


fcmit.cc

```
<!-- where in 查询场景 -->
<select id="select" parameterType="java.util.List" resultMap="BaseResultMap">
    SELECT *
    FROM user
    WHERE name IN
    <foreach collection="names" item="name" open="(" close=")" separator=",">
        #{name}
    </foreach>
</select>
```

3.5、Mybatis Criteria

```
public class UserDO {
    private Integer id;
    private String name;
    private Integer age;
}

public class UserDOExample {
    // auto generate by Mybatis
}

UserDOMapper userMapper = session.getMapper(UserDOMapper.class);
UserDOExample userExample = new UserDOExample();
UserDOExample.Criteria criteria = userExample.createCriteria();

// 一般查询场景
criteria.andNameEqualTo("foo");

// like 模糊查询场景
criteria.andNameLike("%foo%");

// where in 查询场景
criteria.andIdIn(Arrays.asList(1,2));

List<UserDO> users = userMapper.selectByExample(userExample);
```