SQL Injection

1. What is SQL Injection?

- **Quick definition:** An attack where an attacker injects specially-crafted input so it changes the SQL your application executes.
- **Impact:** Unauthorized data exposure, data modification, deletion, or execution of harmful commands.

2. Golden rule

- **Never build SQL queries by concatenating strings with user input.**
- Always separate *query template* from *data* using parameterized queries / prepared statements / ORM parameter binding.

3. Practical examples (short & copy-friendly)

3.1 Vulnerable example (do NOT use)

```
// Dangerous: DO NOT use this with user input
String sql = "SELECT * FROM users WHERE username = '" + username + "' AND password = '" + password + "'";
```

```
Statement st =
conn.createStatement();
ResultSet rs =
st.executeQuery(sql);
```

3.2 Safe example using PreparedStatement (JDBC)

```
String sql = "SELECT * FROM users
WHERE username = ? AND password =
?";
PreparedStatement ps =
conn.prepareStatement(sql);
ps.setString(1, username);
ps.setString(2, password);
ResultSet rs = ps.executeQuery();
```

3.3 Hibernate / JPA — use parameter binding

```
Query<User> q =
session.createQuery("from User u
where u.name = :name",
User.class);
q.setParameter("name", userInput);
List<User> list = q.list();
```

4. Why these are safe

- PreparedStatement and parameter binding separate the SQL structure from the values. Values are treated strictly as data — even if they contain quotes or semicolons, they won't be executed as code.

5. Quick checklist to prevent SQL Injection

1. Replace any string-concatenated SQL with PreparedStatement or ORM parameter binding.
2. Validate inputs (length, type, range) but do not rely on validation alone.
3. Apply the principle of least privilege to the DB account.
4. Avoid dynamic SQL that includes table/column names from user input.
5. Enable logging and monitoring; consider a Web Application Firewall (WAF).
6. Run automated SAST/DAST scans periodically.

6. Quick test (use only on test/staging)

- Try entering this payload in a text field: `' OR '1'='1`
- If the query returns all rows or allows bypassing authentication, the query is vulnerable.

7. Special cases & practical tips

- Stored Procedures: accept parameters and avoid building SQL inside them via concatenation.
- ORMs (Hibernate/JPA): safe if you use parameter binding; but native SQL or concatenated HQL remains dangerous.
- PreparedStatement is sufficient for most JDBC use cases.

8. Quick before/after transformation example

- **Before (vulnerable):**

```
String sql = "SELECT * FROM users WHERE
email='" + email + "'";
```

- **After (safe):**

```
String sql = "SELECT * FROM users WHERE
email = ?";
PreparedStatement ps =
conn.prepareStatement(sql);
ps.setString(1, email);
```

9. References (for further reading)

- OWASP SQL Injection Prevention Cheat Sheet
- PortSwigger SQL Injection resources
- JDBC PreparedStatement documentation
- Hibernate / JPA parameter binding examples