Resposta ao item “Prevenção de SQL Injection”

1. Por que é importante
SQL Injection é uma vulnerabilidade crítica que permite que um atacante injete comandos SQL maliciosos em consultas legítimas
- Ler dados sensíveis
- Modificar ou excluir registros
- Escalar privilégios no banco

Prevenir SQL Injection é essencial para manter a aplicação segura e proteger dados dos usuários.

2. Técnicas principais

2.1. Consultas parametrizadas (Prepared Statements)
Em JDBC, em vez de concatenar strings, use PreparedStatement:

```
String sql = "SELECT * FROM users WHERE username = ? AND status = ?";
try (PreparedStatement ps = conn.prepareStatement(sql)) {
   ps.setString(1, username);      // parâmetro 1
   ps.setString(2, status);        // parâmetro 2
   try (ResultSet rs = ps.executeQuery()) {
      // processa resultados...
   }
}
```

O driver JDBC trata de escapar os valores, eliminando o risco de injeção.

2.2. Uso de ORM com binding de parâmetros
Se usar JPA/Hibernate, sempre passe parâmetros por nome ou posição:

```
String jpql = "SELECT u FROM User u WHERE u.email = :email";
List<User> users = em.createQuery(jpql, User.class)
                     .setParameter("email", emailInput)
                     .getResultList();
```

O provedor de persistência cuidará do escaping dos valores.

2.3. Stored Procedures com parâmetros
Encapsular lógica SQL no banco reduz exposição de queries dinâmicas:

```
-- Exemplo no PostgreSQL
CREATE OR REPLACE FUNCTION get_orders(cust_id INT)
RETURNS TABLE(order_id INT, amount NUMERIC) AS $$
BEGIN
  RETURN QUERY SELECT id, amount FROM orders WHERE customer_id = cust_id;
END;
$$ LANGUAGE plpgsql;
```

Chamando via JDBC:

```
try (CallableStatement cs = conn.prepareCall("{ CALL get_orders(?) }")) {
   cs.setInt(1, customerId);
   try (ResultSet rs = cs.executeQuery()) { ... }
}
```

3. Validação e sanitização de entrada

1. Validação de formato
   - Use regex para emails, por exemplo: ^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$
2. Listas de permissão (whitelists)

- Para campos com valores específicos, compare contra lista fixa.
3. Escapando dados somente quando necessário
- Em casos específicos, utilize APIs do banco, mas nunca como única defesa.

4. Medidas adicionais na camada de banco de dados

- Princípio de menor privilégio: usuário com permissões mínimas.
- Views e funções: exponha apenas o necessário.
- Web Application Firewall (WAF): bloqueie payloads suspeitos.
- Monitoramento e logging: registre queries e crie alertas.

5. Exemplo completo em Spring JDBC Template

```
@Component
public class UserRepository {
    private final NamedParameterJdbcTemplate jdbc;

    public UserRepository(DataSource ds) {
        this.jdbc = new NamedParameterJdbcTemplate(ds);
    }

    public Optional<User> findByUsername(String username) {
        String sql = "SELECT id, username, email FROM users WHERE username = :user";
        MapSqlParameterSource params = new MapSqlParameterSource("user", username);
        return jdbc.query(sql, params, rs -> rs.next()
            ? Optional.of(new User(
                rs.getLong("id"),
                rs.getString("username"),
                rs.getString("email")))
            : Optional.empty());
    }
}
```

6. Conclusão
Ao combinar consultas parametrizadas, uso correto de ORM, validação de entrada e políticas de menor privilégio, você reduz dra