### **PROMPT: @EmployeeDashboardServiceWithVulnerability.java**

### **Analyse the source code properly**

### **ISSUES:**

### **- Find the Security vulnerabilities**

### **- Performance bottlenecks**

### 

### **AI Siggestion:**

### **ANALYSIS OF EmployeeDashboardServiceWithVulnerability.java**

### **ISSUES**

#### **1. Security Vulnerabilities**

* Critical Vulnerability: SQL Injection

The most severe issue in this code is a classic SQL Injection vulnerability.

* Location: Line 16
* Apply to EmployeeDash...
* String sql = "SELECT \* FROM employee\_dashboard WHERE employee\_id = '" + employeeId + "'";
* Explanation: The application constructs an SQL query by directly concatenating a raw, unvalidated input string (employeeId) into the query. An attacker can provide a specially crafted input to manipulate the query's logic.
* Example of Attack:

As noted in the code's comments, if an attacker provides the following input for employeeId:' OR '1'='1The resulting SQL query becomes:

sql

Apply to EmployeeDash...

SELECT \* FROM employee\_dashboard WHERE employee\_id = '' OR '1'='1'

The OR '1'='1' condition is always true, causing the WHERE clause to evaluate to true for every single row in the table. This would bypass the intended security check and return all employee dashboard data to the attacker, leading to a massive data breach. More complex inputs could be used to modify or delete data (DROP TABLE, etc.) if the database user has sufficient permissions.

* Remediation: Always use Prepared Statements (parameterized queries). This approach separates the SQL code from the data, ensuring that user input is treated as literal data and not as executable code.

#### **2. Performance Bottlenecks**

* Major Bottleneck: Resource Leaks due to Improper Cleanup
* Location: Lines 28-30 inside the try block.
* Apply to EmployeeDash...
* rs.close();
* stmt.close();
* conn.close();
* Explanation: The database resources (Connection, Statement, ResultSet) are closed only if the code executes successfully. If an exception occurs at any point within the try block (e.g., the query fails, a network error occurs), the catch block is executed, and the .close() methods are never called. This is a resource leak. Each leaked connection consumes memory and holds a session open on the database server. Over time, this will exhaust the available connections, leading to application failure and potentially crashing the database server.
* Remediation: Use a finally block or, preferably, the try-with-resources statement (available since Java 7) to guarantee that all resources are closed automatically, even if an exception occurs.
* Minor Bottleneck: No Connection Pooling
* Location: Line 13
* Apply to EmployeeDash...
* conn = DriverManager.getConnection(...);
* Explanation: The code establishes a new database connection every time the viewEmployeeDashboard method is called. Creating a database connection is a very slow and resource-intensive operation, involving network handshakes and session setup on the database. In a real-world application with many concurrent users, this would be a major performance bottleneck, causing high latency and overwhelming the database.
* Remediation: Use a connection pool (e.g., HikariCP, C3P0). A connection pool maintains a set of open connections that can be borrowed and returned by the application, eliminating the overhead of creating new connections for every request.
* Minor Bottleneck: Inefficient Query with SELECT \*
* Location: Line 16
* Apply to EmployeeDash...
* String sql = "SELECT \* FROM employee\_dashboard ...";
* Explanation: Using SELECT \* retrieves every column from the table, even if the application only uses a few of them. This increases the amount of data transferred over the network from the database to the application and uses more memory in the application itself.
* Remediation: Explicitly list the required columns in the query.
* Apply to EmployeeDash...
* SELECT employee\_name, department, status FROM employee\_dashboard WHERE ...

This ensures only the necessary data is requested, which is more efficient.