Database Application Programming and Security considerations

Chapter 6

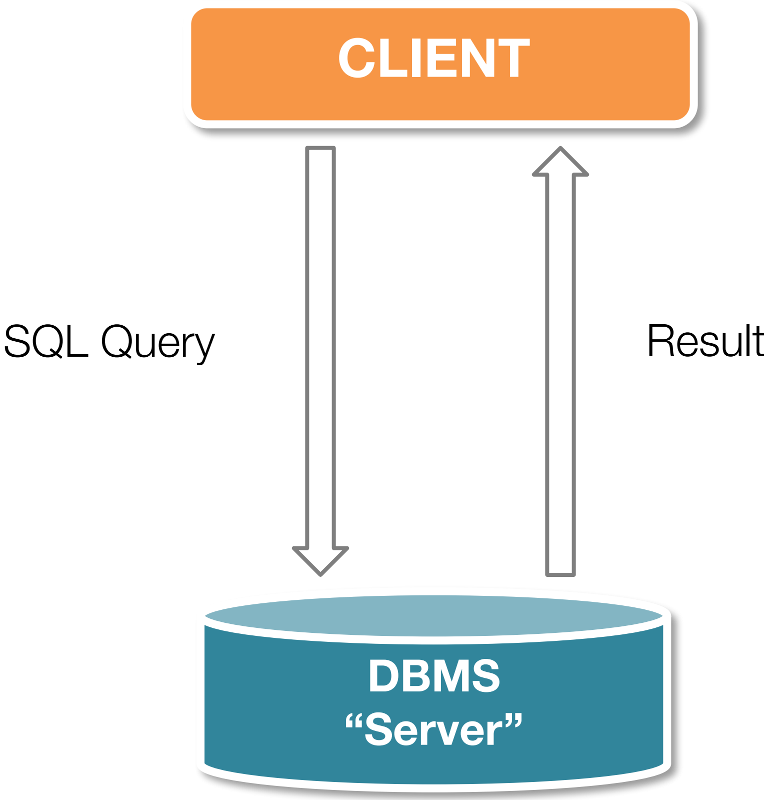
(JDBC Section)

Databases “In the Wild”

- So far, we’ve talked about the DBMS as a standalone system
  - Access interactively by writing SQL queries (e.g., using SQL*Plus)
- In practice, DBMS is often part of a larger software infrastructure
  - Multi-tiered system architecture
  - Access database from another program

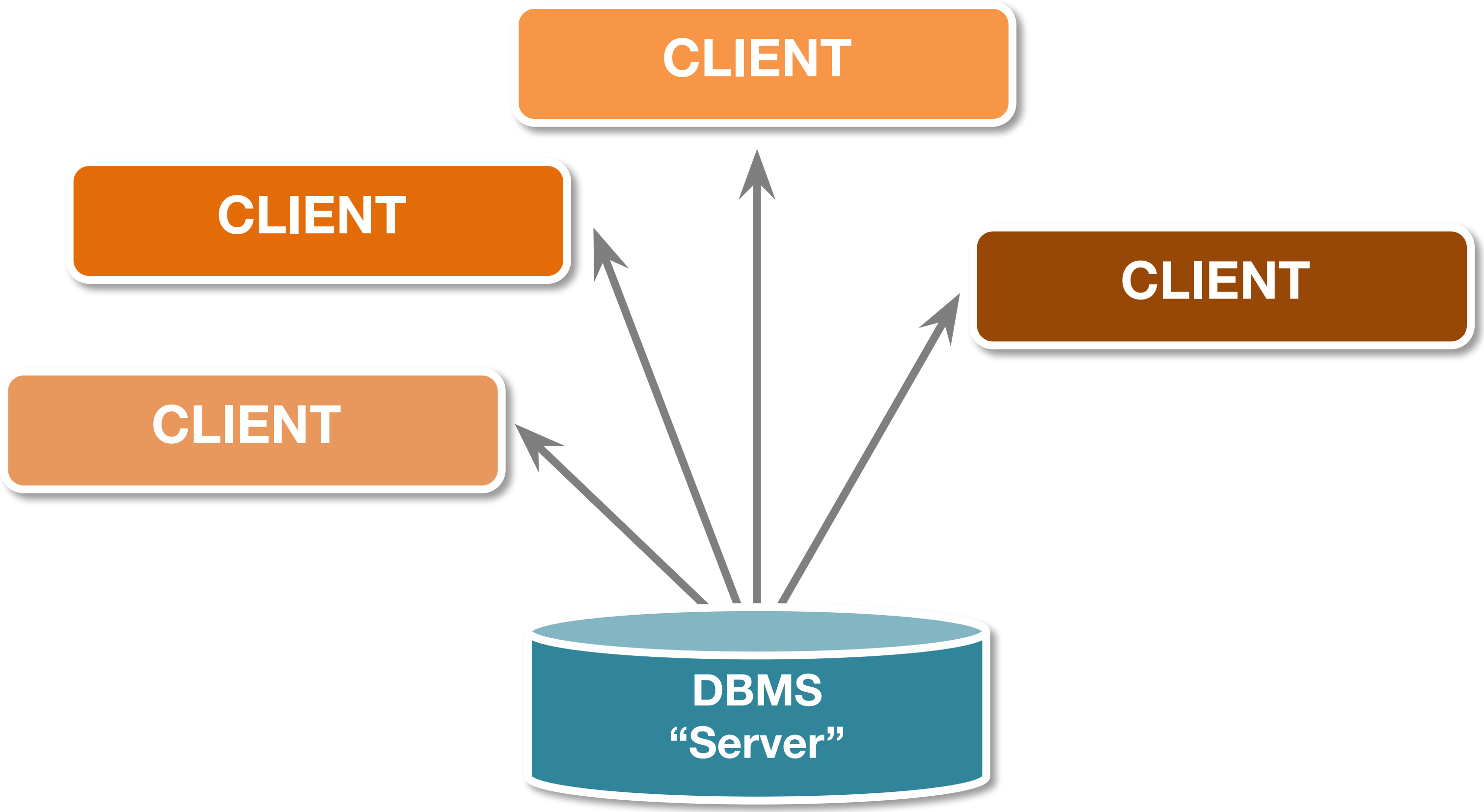
Database “Ecosystem” (1)

“Client-Server Architecture”



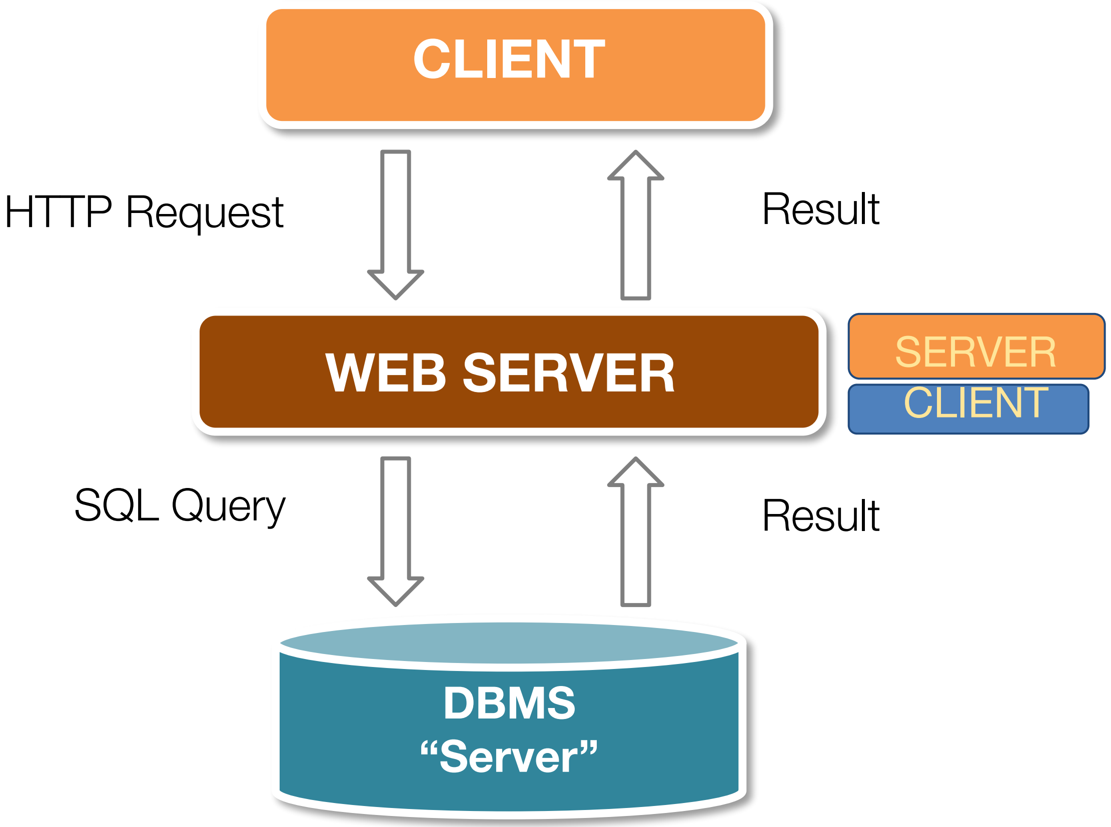
Many Clients

“Client-Server Architecture”

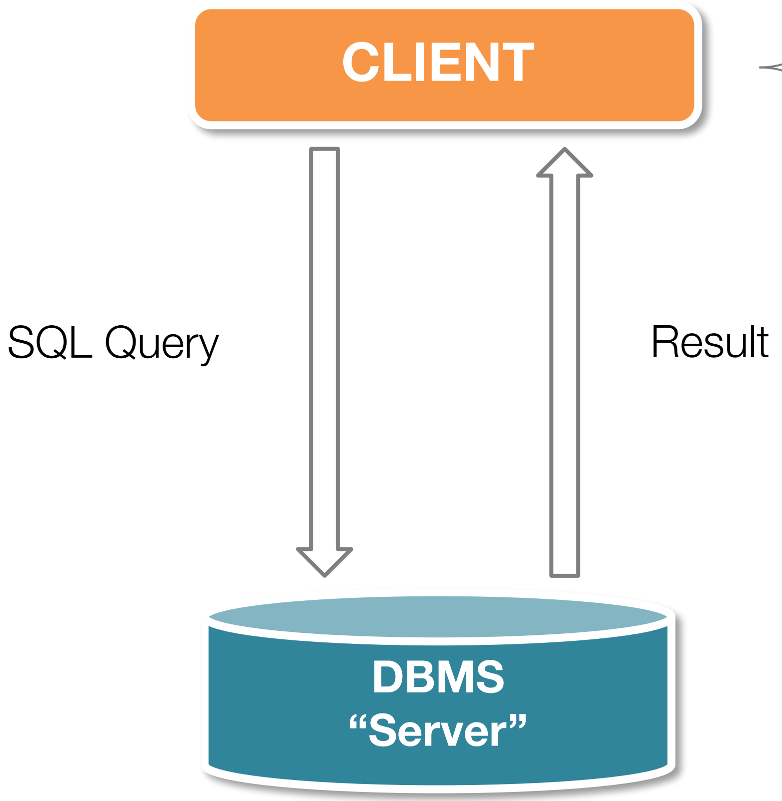


Database “Ecosystem” (2)

“3-Tier Architecture” (Common to add more tiers, too)



Embedding SQL in Application



Client is often a program, written in a language like Java, Python, or C++

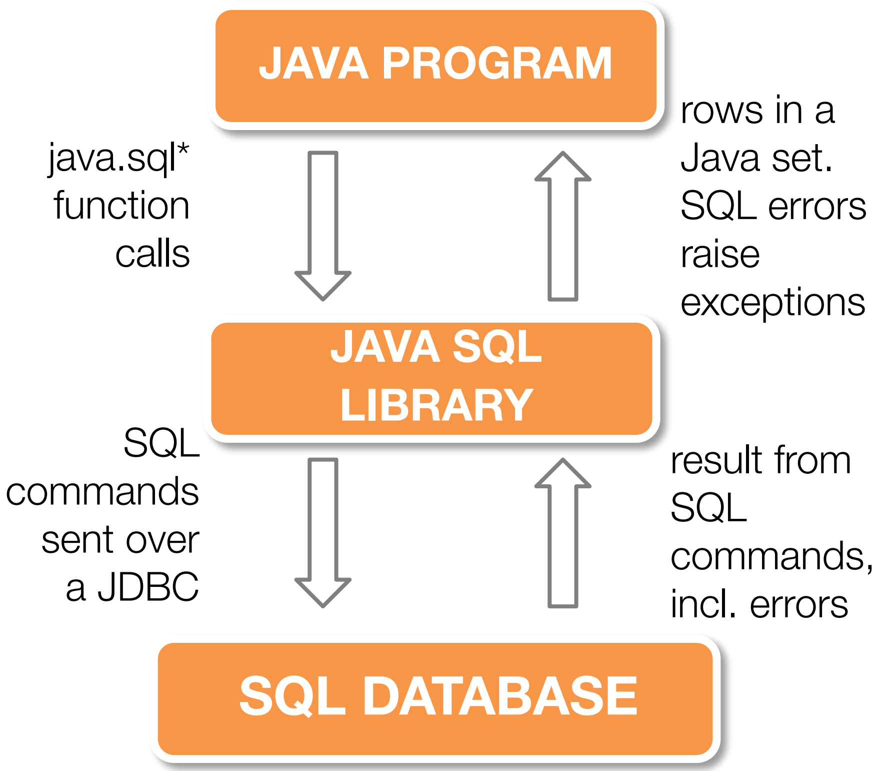
CHALLENGE:
How to access SQL from application code?

SQL Integration with PL

- **Problem:** What is the interface between SQL and programming language?
- **A popular solution:**
  - “Embed” SQL in host language
  - Provide an API for processing query results
- We will see one such example next for Java
- Similar embeddings of SQL exist for Python and other popular languages.

JDBC (“Java-Database Connectivity”)

- Connect to a database using a JDBC driver
- Send queries over the JDBC connection
- Receive results into a Java ResultSet



JAVA PROGRAM

java.sql* function calls

rows in a Java set. SQL errors raise exceptions

JAVA SQL LIBRARY

SQL commands sent over a JDBC

result from SQL commands, incl. errors

SQL DATABASE

Downloading Java

- Java should run on CAEN machines.
- In case you need Java on your own computer, a version that worked for me on M1 Mac was installed using brew and temurin (install temurin 8.x.x for best compatibility with P2):
  - https://adoptium.net/installation/
- To test in a terminal:
  - javac and java commands should work
- The above website also has versions for Windows.
- Other JDKs should also work. You likely want Java version 8 (or 1.8) for compatibility with CAEN.

JDBC with Sqlite or Oracle

- Download the latest jdbc driver jar file for sqlite and demo/Sample.java from https://github.com/xerial/sqlite-jdbc
- You
- Compile: javac Sample.java
- Run: java –cp .:*sqlite-jdbc-{version}.jar* Sample
- Alternative: Oracle driver posted with Project 2 and use the Oracle database from a Java program as in Project 2.

JDBC Example

```
Connection conn;
// Insert code here to connect conn to a DB.
// Requires JDBC driver; See sample code in Project 2 for
// Oracle or for Sqlite, Sample.java

String q = "SELECT name FROM Students WHERE GPA > 3.5";
try {
      Statement st = conn.createStatement();
      ResultSet rs = st.executeQuery(q);

      while (rs.next()) {
         String name = rs.getString(”name");
         System.out.println(name);
      }
      rs.close();
      st.close();
}
catch(SQLException e){System.err.println(e.getMessage());}
```

Cursor retrieves rows from result one at a time

Full Javadoc for java.sql available online:
http://download.oracle.com/javase/6/docs/api/

Connections

- Get Connection object:
  - Oracle requires passwords
  - Sqlite3 is file-based and does not require a password
    - Connection conn = ….
- Always close connections before quitting the program
  - conn.close();
  - Similarly, close other Oracle resources.

Cool Trick to Auto-Close Resources

- A cool trick in Java/JDBC to auto-close database connection and other resources automatically.

http://docs.oracle.com/javase/7/docs/technotes/guides/jdbc/jdbc_41.html

JDBC – AutoClose Trick

```
Connection conn;
// Obtain a connection to DB, store in conn
// (Requires JDBC driver; See sample code in Project 2)

String q = "SELECT name FROM Students WHERE GPA > 3.5";
try (Statement st = conn.createStatement()) { // auto-close
     ResultSet rs = st.executeQuery(q);

     while (rs.next()) {
        String name = rs.getString(”name");
        System.out.println(name);
     }
     // rs.close();    Not needed.
     // st.close();   Not needed.
}
catch(SQLException e){System.err.println(e.getMessage());}
```

Full Javadoc for java.sql available online:
http://download.oracle.com/javase/6/docs/api/

Challenges

- DBMS and PL implement different data types
  - "Impedance Mismatch"
- Need to match DB types with PL types, e.g.,

SQL Type	Java Type	ResultSet Method
CHAR	String	getString()
VARCHAR	String	getString()
DOUBLE	Double	getDouble()
INTEGER	Integer	getInt()

JDBC Example

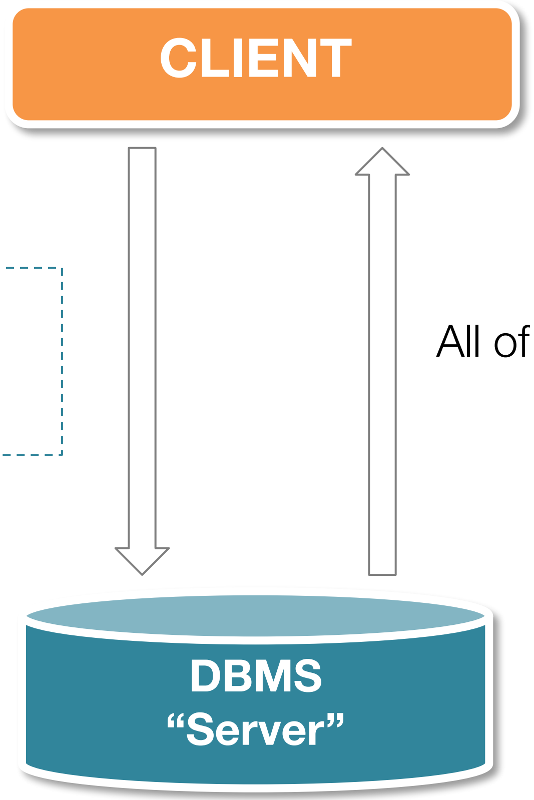
What does the following code snippet do?

```
String query = "SELECT NAME, AGE FROM CITIZENS";
try (Statement st = conn.createStatement()){ // auto-close
      double sum = 0;
      double count = 0;

     ResultSet rs = st.executeQuery(query);
     while (rs.next()) {
        String name = rs.getString("NAME");
        sum += rs.getDouble("AGE");
      count++;
     }
      System.out.println(sum/count);
}
catch(SQLException e){System.err.println(e.getMessage());}
```

What Happens?

Compute the average age



```
SELECT NAME, AGE
FROM CITIZENS
```

All of the citizen records

What if there are millions of citizens?

JDBC Example (Revised)

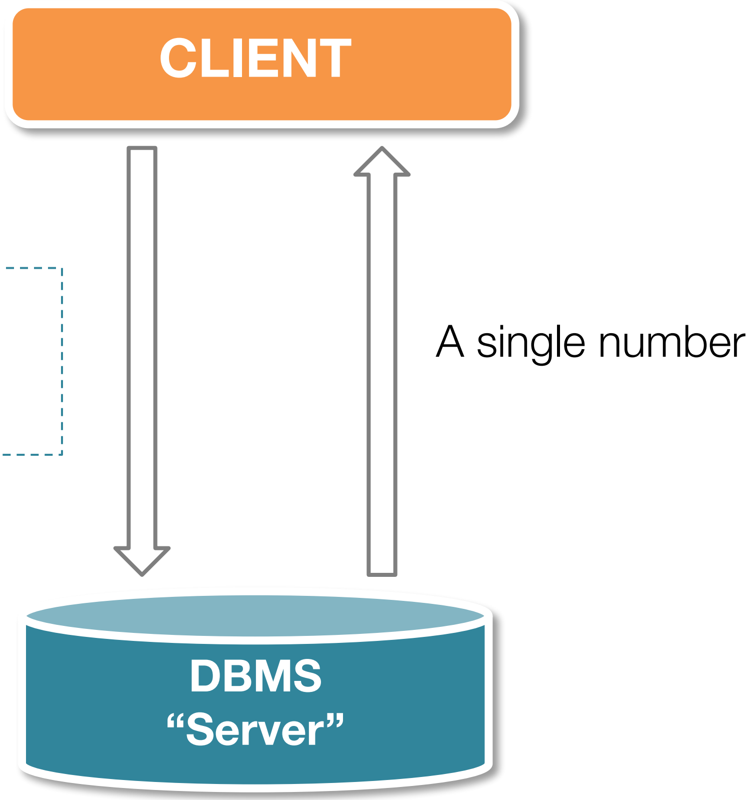
Push the computation “closer” to the data…
Make DBMS do the processing that it does well.

```
String query = "SELECT AVG(AGE) FROM CITIZENS";
Try (Statement st = conn.createStatement()) { // auto-close
     ResultSet rs = st.executeQuery(query);
     while (rs.next()) {
        Double avg = rs.getDouble(1);
          System.out.println(avg);
     }
}
catch(SQLException e){System.err.println(e.getMessage());}
```

What Happens Now?

Compute the average age

```
SELECT AVG(AGE)
FROM CITIZENS
```



Question??

Does the above apply even if there is no aggregation?

Consider the following:

```
SELECT NAME, AGE
FROM CITIZENS
```

A

```
SELECT NAME, AGE
FROM CITIZENS
WHERE SALARY < 1000
```

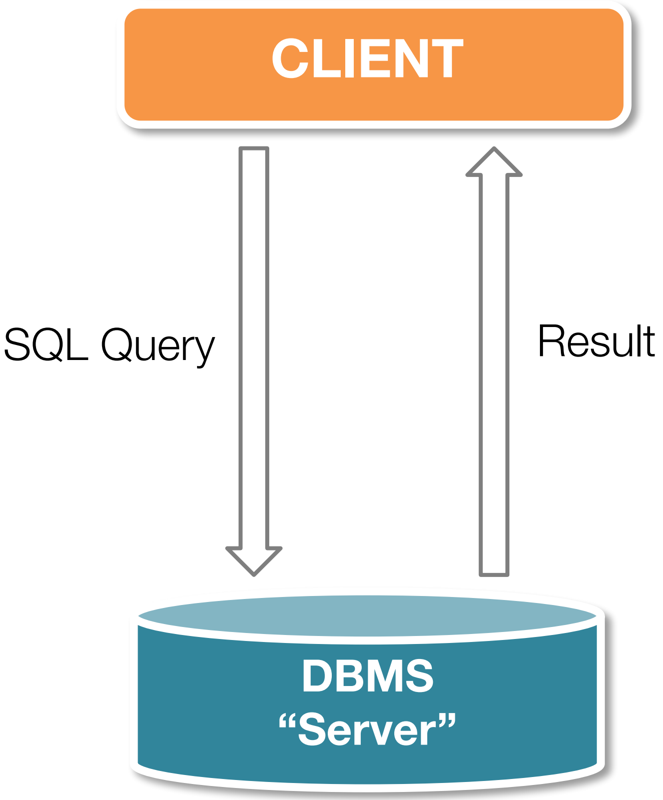
B

Select which query you would prefer to send in SQL (and do the rest in Java)

Challenges

- What computation to do in the database vs. the application program?
- Rules of Thumb:
    - Avoid fetching more data than necessary
    - “Push” data processing to the DBMS when possible

Stored Procedures and UDFs



Database applications you write (e.g., using JDBC) are usually located outside the DBMS.

Exceptions:

- User-Defined Functions (UDFs)
- Stored Procedures
- Stored and executed within the DBMS

SQL FUNCTIONS/STORED PROCEDURES

- You have already seen these!!
  - E.g. LOG(<number>), NEXTDAY(<date>), LENGTH(<string>), …
- Aggregate functions
  - Many more than the big 5!
  - STDDEV, percentiles, …
- User defined functions
  - CREATE FUNCTION <name> AS …
- Can use a function to specify column value in a table
- Stored Procedures are more general than functions.
  - Can use arbitrary DML, including INSERT/UPDATE/DELETE

PreparedStatement

- Statement (Base class)
    - Arbitrary SQL query
    - DBMS parses and optimizes each query
- PreparedStatement
    - Parameterized SQL query
    - DBMS “pre-compiles” the query (parses, optimizes, and stores query execution plan)
    - Can be used multiple times
    - Amortizes optimization cost across multiple uses

Statement vs. PreparedStatement

```
public List<String> getNames (int age) {
   String q = "SELECT name FROM Students WHERE age = " + age;
   Statement st = conn.createStatement();
   ResultSet rs = st.executeQuery(q);
   …
}
```

```
PreparedStatement ps;
public List<String> getNames (int age) {
   String q = "SELECT name FROM Students WHERE age = ?";
   if (!ps) ps = conn.prepareStatement(q);
   ps.setDouble(1, age);
   ResultSet rs = ps.executeQuery();
   …
}
```

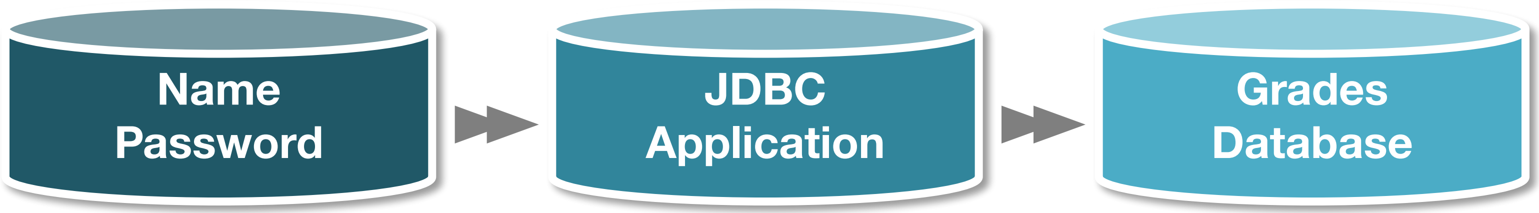
PreparedStatement

- PreparedStatement also does some type checking on parameters and removal of escape characters
- Helpful for preventing some security problems (e.g., SQL Injection)

Security Issues / SQL Injection

- Common vulnerability in database applications
  - SQL Injection is simple, yet surprisingly common
  - Can be prevented with defensive coding

Web Front-End



Name
Password

JDBC
Application

Grades
Database

Students
(Name,Grade,Password)

SQL Injection – Example

JDBC

```
Authenticate (String n, String p) {
…
String query =
"SELECT grade FROM students WHERE name = '" + n + "'
AND password = '" + p + "' ";
…
}
```

Input:

```
name = bart; password = mypword
```

SQL:

```
SELECT grade FROM students WHERE name = 'bart' AND
password = 'mypword'
```

SQL Injection – Example

JDBC

```
Authenticate (String n, String p) {
…
String query =
"SELECT grade FROM students
 WHERE name = '" + n + "'
 AND password = '" + p + "' ";
…
}
```

Input:

```
name = lisa; password = n' OR 'x'='x
```

SQL:

```
SELECT grade FROM students WHERE name = 'lisa' AND
password = 'n' OR 'x' = 'x'
```

SQL Injection – Example

JDBC

```
Authenticate (String n, String p) {
…
String query =
"SELECT grade FROM students
 WHERE name = '" + n + "'
 AND password = '" + p + "' ";
…
}
```

Attacker: Passes in the following strings for n and p

```
n= foo; p= n'; UPDATE students SET grade= 'A
```

Resulting value of query:

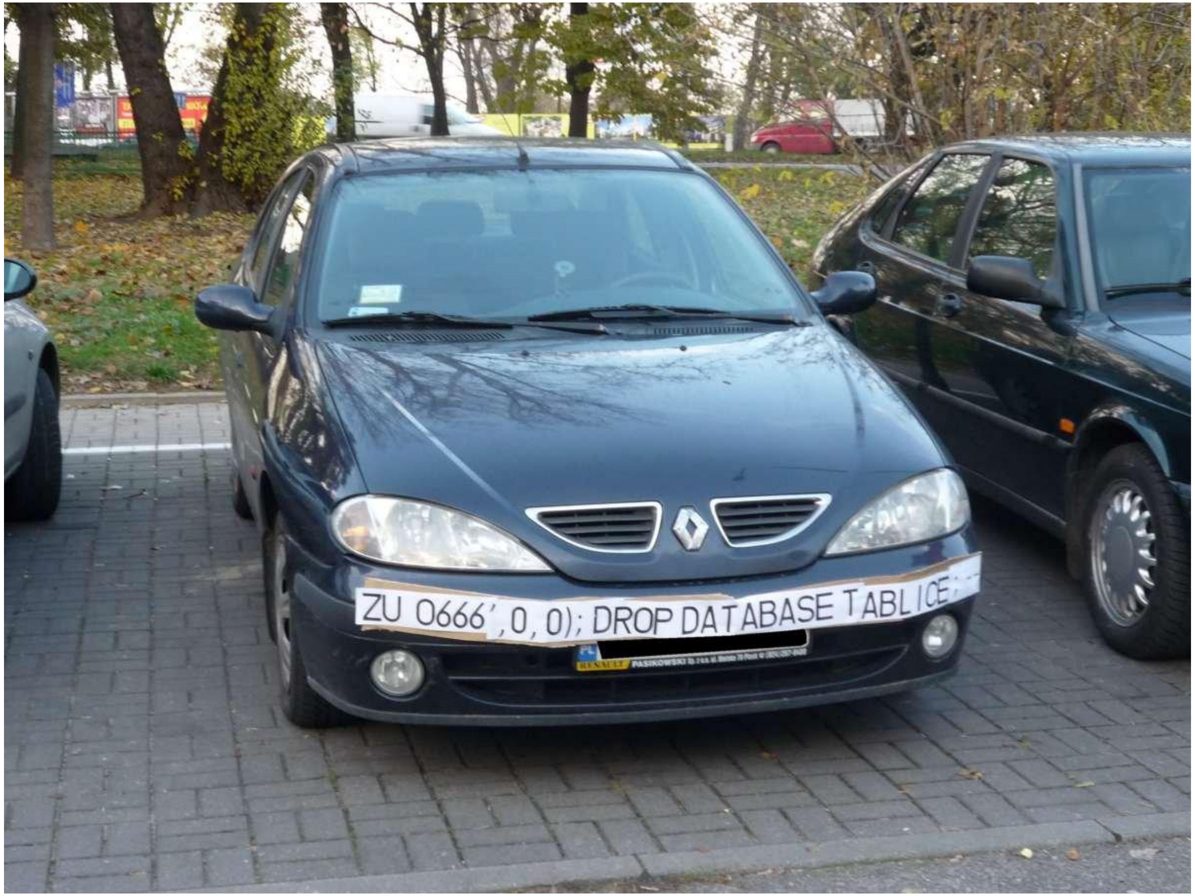
```
SELECT grade FROM students WHERE name = 'foo' AND
password = 'n'; UPDATE students SET grade = 'A'
```

SQL Injection

- Cute little cartoon on How Little Bobby Tables Ruined the Internet (Worth reading through as to why it works):

  https://medium.com/@johnteckert/how-little-bobby-tables-ruined-the-internet-d714c20d2ce0

SQL Injection



SQL Injection – Prevention #1 (Sanitize)

- **Check and sanitize any untrusted inputs**
- E.g., any browser input is fundamentally untrusted and potentially malicious
- So, sanitize before inserting into query so that an unauthorized sql query cannot be injected.

Weird Password rules

- Systems have seemingly weird password rules.
  - E.g., Only letters, numbers, and the symbols "#", "_" and "$" are acceptable in a password.

Would the above restriction have prevented the SQL injection attack with n and p on our previous slide?

- Answer: ?

Prevention #2. Always Use PreparedStatement

JDBC

```
String q = "SELECT grade FROM student
   where name = ? AND password = ?";
 ps = conn.prepareStatement(q);
…
Authenticate (String n, String p) {
…
   ps.setString(1, n);
   ps.setString(2, p);

   ResultSet rs = ps.executeQuery();…
}
```

Input:

```
n= foo;
p= n'; UPDATE students SET grade= ‘A
```

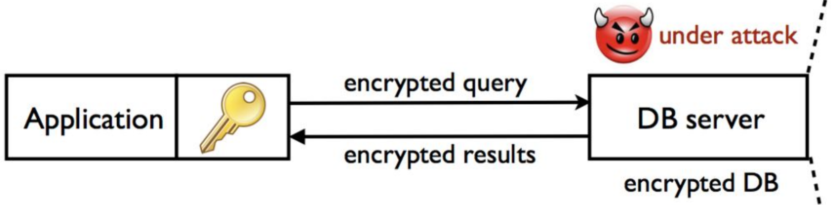
Use of PreparedStatement here prevents the SQL injection attack.

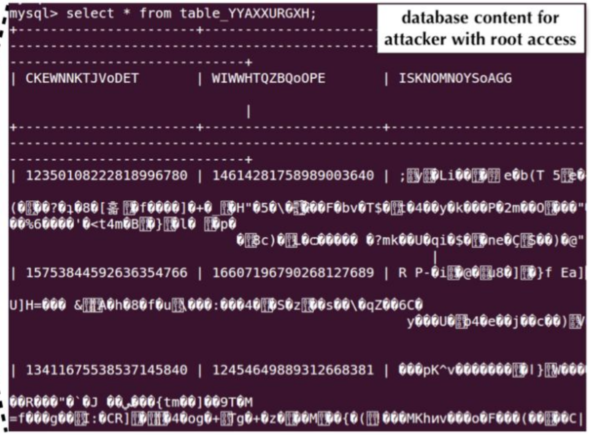
Prevention #3: DB Access Control

- Limit JDBC app's rights by assigning it low-privilege account (just enough for it to get its work done)
  - Most DBMSes allow restricting read/write access to tables or columns (and sometimes rows) based on user id, E.g.,
    - GRANT SELECT ON TABLE T to userid;
    - DENY INSERT ON TABLE T TO userid
- Views can also help restrict access

Prevention #4: Encryption

- At-rest encryption: Prevents data loss if disk or computer gets stolen. But, data in plaintext at run-time
- Encrypted databases such as CryptDB and Mylar: Data kept encrypted. Queries also encrypted




database content for attacker with root access

```
mysql> select * from table_YYAXXURGXH;
+-----------------------+-----------------------
----------------------------------------------
------------------------------+
| CKEWNNKTJVoDET        | WIWWHTQZBQoOPE        | ISKNOMNOYSoAGG

                              |
+-----------------------+-----------------------+-----------------------
-------------------------------------------------------------------------
------------------------------+
| 12350108222818996780 | 14614281758989003640 | ;�y��Li����� e�b(T 5�e�

(����?�₁�8�[�� ��f����]�+� ��H"�5�\����F�bv�T$���4��y�k���P�2m��O����"
��%6����'�<t4m�B��}��l� ��p�
                                 ��8c)���L�c����� �?mk��U�qi�$���ne�Ç�$��)�@"
                                                          |
| 15753844592636354766 | 16607196790268127689 | R P-�i��@���u8�]��}f Ea]

U]H=��� &��A�h�8�f�u��\���:���4���S�z���s��\�qZ��6C�
                                                       y���U���b4�e��j��c��)��

| 13411675538537145840 | 12454649889312668381 | ���pK^v���������!}�W���

��R���"�`�J ��,���{tm��]��9T�M
=f���g����:�CR]������4�og�+��Tg�+�z����M���{�(�� ���MKhuv���o�F���(�����C|
```

Summary

- DBMS is often part of a larger software infrastructure
  - E.g., Database-backed web applications
- Integrating SQL with application code is a messy problem
  - Efficiency: Push computation “close” to data
  - Security: Sanitize input and other defenses