COMP 430
Intro. to Database Systems
SQL from application code
Some issues

• How to connect to database
  • Where, what type, user credentials, ...
• How to send SQL commands
• How to get communicate data to/from DB
  • Data type conversion

Details vary by language & library.
public class Example {
    
    public static void main(String[] args) {
        Connection connection = null;
        Class.forName("com.Microsoft.jdbc.sqlserver.SQLServerDriver");
        String url = "jdbc:microsoft:sqlserver://localhost:1433;DatabaseName=MYDB";
        connection = DriverManager.getConnection(url, "USERNAME", "PASSWORD");
        ...
    }
}

Need sqljdbc4.jar in CLASSPATH.
Connecting to database – Python + SQLAlchemy example

```python
engine = create_engine("mssql+pyodbc://USERNAME:PASSWORD@localhost/MYDB" + 
    "driver=SQL+Server+Native+Client+10.0")
```

Key pieces:
- Driver
- Host
- DB name
- User credentials
Connecting to database

Those *connection strings* share a standard syntax, although some arguments can be specified separately by library.
connection = ...;
Statement stmt1 = connection.createStatement();
stmt1.executeUpdate("CREATE TABLE Student" +
                    "(id INT, first VARCHAR(50), last VARCHAR(50))");
...
Statement stmt2 = connection.createStatement();
ResultSet result = stmt2.executeQuery("SELECT id FROM Student");
...
Problems with string representation

• Two data conversions: application $\rightarrow$ string, string $\rightarrow$ DB

• Minimal API
  • Requires SQL knowledge
  • SQL commands not limited to an API

• Lacks structure
  • Arbitrary data representation in application
  • No static checking
Plain strings allows SQL injection attacks
What are some bad input strings?

studentId = getRequestString("StudentID");
String query = "SELECT first, last FROM Student WHERE id = " + studentId;
Statement stmt = connection.createStatement(query);

WHERE clause irrelevant: 123456789 OR 1=1

Destructive behavior: 123456789; DROP TABLE Student

Many variations on these themes.
Techniques for preventing injection attacks

• Validate input used in SQL command strings
• Build SQL commands via parameterized APIs (next)
• Access DB via stored procedures

• Tightly manage user permissions
Simple parameterization – Java example

```java
studentId = getRequestString("StudentID");
Statement stmt = connection.prepareStatement("SELECT first, last FROM Student WHERE id=?");
Stmt.setInt(1, Integer.parseInt(studentId));
ResultSet result = Stmt.executeQuery(stmt);
while (result.next()) {
    String first = result.getString("first");
    String last = result.getString("last");
    ...
}
```

Essentially a cursor.
Object-Relational Mapping (ORM)

A high-level overview
Primary goal – persistent objects

• DB viewed as a tool for implementing persistent objects.

• Relational DB assumed for practical or legacy reasons.
  • But, DB isn’t organized in terms of objects.

• Programmer think in terms of application & objects.
  • Specify which objects persistent
  • Interaction with DB largely(?) hidden
Focus on data

O/R Mapping

Objects in Memory → Mapping Logic ← Relational Database
Focus on data

```
public class Student {
    private int id;
    private String firstName;
    private String lastName;

    public Student(...) {} 
    ... /* getters & setters */
}
```

```
CREATE TABLE Student ( 
    id INT AUTOINCREMENT, 
    first_name VARCHAR(50), 
    last_name VARCHAR(50), 
    PRIMARY KEY (id) 
);
```
What an ORM can provide

• Generate code for simple object/table-record mapping

• API for OO-style CRUD (Create-Read-Update-Delete) of objects

• API for OO-style queries

• Manage how & when data is moved
Generate object (& table?) from specification

**Hibernate**

```xml
<hibernate-mapping>
  <class name="Student" table="Student">
    <id name="id" type="int" column="id">
      <generator class="native"/>
    </id>
    <property name="firstName" column="first_name" type="string"/>
    <property name="lastName" column="last_name" type="string"/>
  </class>
</hibernate-mapping>
```

**SQLAlchemy**

```python
class Student(Base):
    __tablename__ = 'Student'
    id = Column(Integer, primary_key=True, autoincrement=True)
    first_name = Column(String)
    last_name = Column(String)
```
OO-style CRUD

- Object constructor
- Object getters & setters
- Methods corresponding to CREATE, INSERT, UPDATE, DELETE

```python
engine.execute(Student.insert(), {'first_name': 'John', 'last_name': 'Smith'})
```
**OO-style queries**

- Potentially no explicit SQL in application code.
- But programmer still needs to understand SQL concepts.

```python
resultset = session.query(User, Document, DocumentPermission)
    .join(Document)
    .join(DocumentPermission)
    .filter(User.email == 'john_smith@foo.com')
```
How & when data is moved

- **Granularity:**
  - Application-level traditionally accesses an attribute at a time.
  - DB-level access one or more records at a time.

- **Consistency**
  - Are application & DB data guaranteed to be consistent?

- **Eager vs. lazy loading**
Some ORM problems

• Complicated
  • Hard to learn
  • Some would say bloated

• Fragmented
  • Many frameworks, many standards
  • Hard to move from one to another

• Only partially successful