Databases and ORMs

CSCI 4448/5448: Object-Oriented Analysis & Design

Lecture 34

Acknowledgement & Materials Copyright

- I'd like to start by acknowledging Dr. Ken Anderson
- Ken is a Professor and the Chair of the Department of Computer Science
- Ken taught OOAD on several occasions, and has graciously allowed me to use his copyrighted material for this instance of the class
- Although I will modify the materials to update and personalize this class, the original materials this class is based on are all copyrighted © Kenneth M. Anderson; the materials are used with his consent; and this use in no way challenges his copyright

Goals of the Lecture

- Lightning database review
- ORMs in General
- Hibernate in Particular
- ORM alternatives

Where the wild data lives

- Spreadsheets
- Custom serialized data – Java Serialization, Python Pickles
- Other structured representations – CSV, JSON, XML, YAML
- Relational databases
- No-SQL databases
- Custom storage formats

Relational Databases

- A database is a utility for saving persistent data and for searching and relating various data elements and sources
- The most common databases are relational databases
- Relational databases store data in tables, and use key references to relate how one data row in a table might relate to another data row in another table
- Data tables are created using a definition language, often SQL (Structured Query Language), and other SQL elements are used to Create, Read, Update, and Delete data (the so-called CRUD operations)
- SQL is fairly standard across database applications
- https://www.fullstackpython.com/databases.html

No-SQL Databases

- There are a number of data storage tools that do not use SQL, and are designed for specific performance or usage profiles
- Typical NoSQL data stores include
  - Key-value pairs – examples: Redis, Memcached
  - Document-oriented – example: Mongo-DB
  - Column-family tables – example: Cassandra
  - Graphs – examples: Neo4j, Cayley, Titan
- Redis, for instance, is often used for web applications where fast response time for session data is required

https://www.fullstackpython.com/no-sql-datastore.html

Common Database Tools

- Relational Databases
  - SQLite – built into Python, limited to single connections
  - MySQL – easiest to pick up and use
  - PostgreSQL – most feature rich of the set
- NoSQL Databases
  - Redis – Go to tool for most data caching speed-dependent applications
  - MongoDB – stores JSON documents for more complex data
- AWS Data Storage (typical Cloud support)
  - AWS Elasticache offers Redis and Memcached implementations
  - AWS Aurora – MySQL, PostgreSQL compatible
  - AWS RDS (Relational Database Service) – six common relational database engines

Typical Redis interaction with Python

- This is a “Hello, World” example from an online Redis tutorial https://redislabs.com/lp/python-redis/
- Steps include
  - Install Redis
  - Install redis-py
  - Python data access
- More information at redis-py site https://github.com/andymccurdy/redis-py/

At the Linux command line:

sudo apt-get install redis-server

pip3 install redis-py

Python:

```
#!/usr/bin/python3
import redis

# connect to your redis instance from redis-py using defaults
r = redis.Redis( host='localhost', port=6379, db=0)

# write to redis using key “greet”, value “Hello World!”
r.set(“greet”,”Hello World!”)

# read from redis using the key “greet”
value = r.get(“greet”)
print(value)     # Outputs “Hello World!”
```

Typical in-line MySQL interaction with Python

- This example is from an online MySQL tutorial https://pythonspot.com/mysql-with-python/
- Steps include
  - Install MySQL
  - Database setup
  - Python data access

- The equivalent of this in Java is JDBC, Java Database Connectivity, which provides simple SQL query processing in line

- What's wrong with this?

At the Linux command line:

```
sudo apt-get install python-mysqldb

mysql -u USERNAME –p
mysql&gt; CREATE DATABASE pythonspot;
mysql&gt; USE pythonspot;
CREATE TABLE IF NOT EXISTS examples (
  id int(11) NOT NULL AUTO_INCREMENT,
  description varchar(45),
  PRIMARY KEY (id)
);
INSERT INTO examples(description) VALUES ("Hello World");
INSERT INTO examples(description) VALUES ("MySQL Example");
INSERT INTO examples(description) VALUES ("Flask Example");
```

Python:

```
#!/usr/bin/python3
import MySQLdb

db = MySQLdb.connect(host="localhost",  # your host
                     user="root",       # username
                     passwd="root",     # password
                     db="pythonspot")   # name of the database

# Create a Cursor object to execute queries
cur = db.cursor()
# Select data from table using SQL query.
cur.execute("SELECT * FROM examples")
# print the first and second columns
for row in cur.fetchall() :
    print row[0], " ", row[1]
```

Using an RDBMS directly in-line - Issues

- Granularity
  - Your object model may have more classes than tables in the database, may make operations hard to place and maintain
- Inheritance
  - No inheritance model in a database
- Identity
  - RDBMS has one identity measure – the primary key; Java has several: object identity (a==b), object equality (a.equals(b)), object type matches
- Associations
  - OO languages represent association via object references, an RDBMS uses foreign keys or relational tables
- Navigation
  - Completely and fundamentally different models of accessing data and objects in Java and an RDBMS

- https://www.tutorialspoint.com/hibernate/orm_overview.htm

ORM

- ORM stands for Object-Relational Mapper (or Mapping).
  - It is a programming technique for converting data between relational databases and object oriented programming languages such as Java, C#, Python, etc.
- ORM Elements
  - An API to perform basic CRUD operations on objects of persistent classes
  - A language or API to specify queries that refer to classes and properties of classes
  - A configurable facility for specifying mapping metadata
  - A technique to interact with transactional objects to perform dirty checking, lazy association fetching, and other optimization functions

- https://www.tutorialspoint.com/hibernate/orm_overview.htm

ORM Advantages in Java

- Lets the business code access objects rather than DB tables.
- Hides details of SQL queries from OO logic
- Based on JDBC and/or JPA 'under the hood.'
- No need to deal with the database implementation
- Entities based on business concepts rather than database structure
- Transaction management and automatic key generation
- Fast development of applications

- https://www.tutorialspoint.com/hibernate/orm_overview.htm

Java and Python ORMs

- Java
  - Hibernate
  - Enterprise JavaBeans Entity Beans
  - Java Data Objects
  - Apache OpenJPA
  - Castor
  - TopLink
  - Spring DAO
  - Many other choices…

- https://www.tutorialspoint.com/hibernate/orm_overview.htm

- Python
  - SQLAlchemy
  - Peewee
  - The Django ORM
  - PonyORM
  - SQLObject
  - Tortoise ORM
  - Others…

- https://www.fullstackpython.com/object-relational-mappers-orms.html

Hibernate

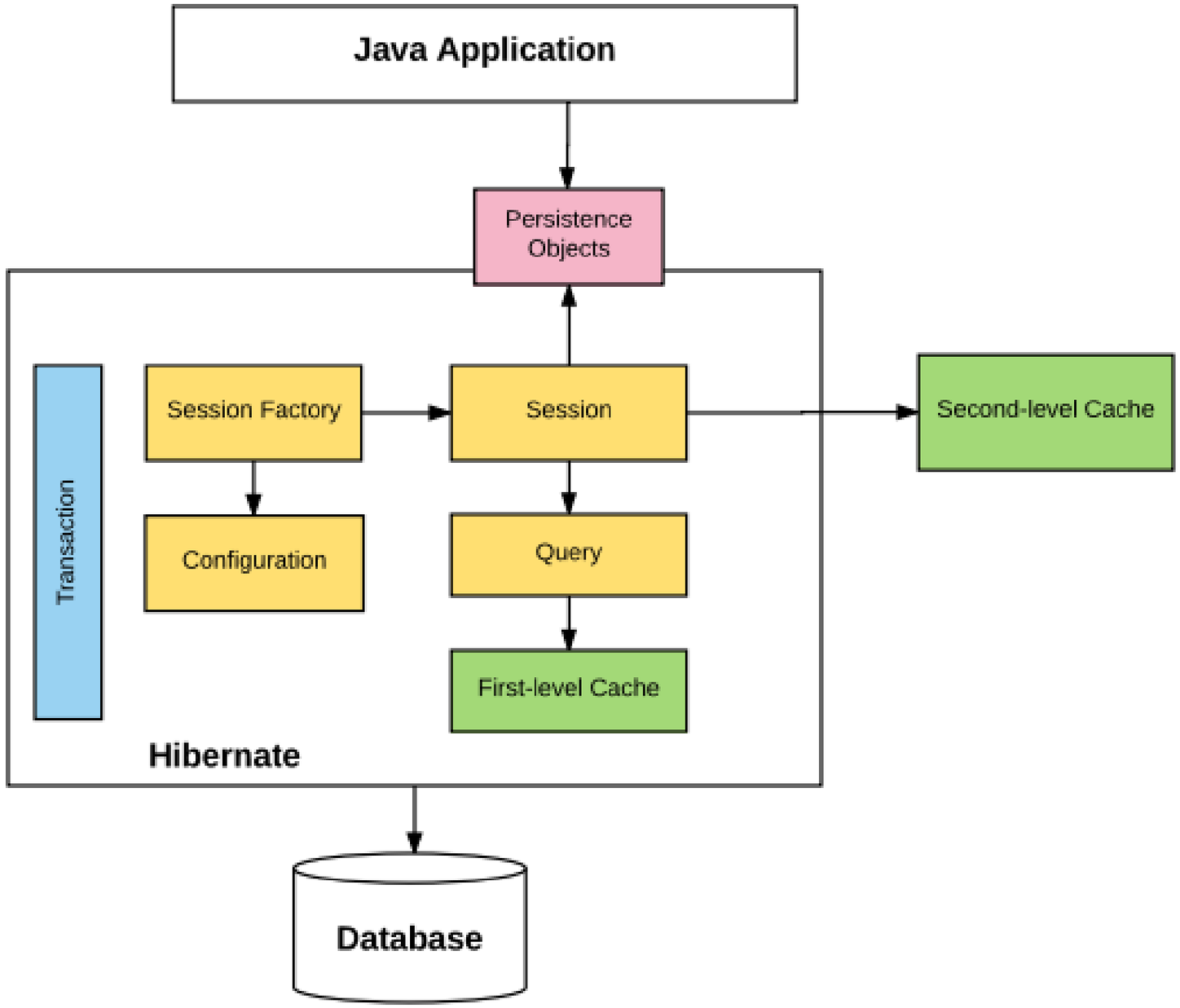
- Hibernate is a popular ORM choice for Java (https://hibernate.org/)
  - NHibernate is a port of Hibernate for .NET
- Implements the Java Persistence API
- Supports many standard databases (MySQL, PostgreSQL, Oracle, SQL Server, etc.)
- Maps Java classes to database tables using XML
- Provides simple APIs for storing/retrieving Java objects directly to/from the database
- Abstracts SQL elements
- Provides for complex associations of objects in the database
- Minimizes database access (lazy fetching)
- Simplifies querying

Hibernate elements:

- Hibernate ORM – base services and associate with database
- Hibernate EntityManager – implementation of Java Persistence APIs
- Hibernate Validator – data and class validations tool
- Hibernate Envers – audit logging and history
- Hibernate Search – Query API for full text search
- Hibernate OGM – object/grid mapper for NoSQL databases

Hibernate Architecture

- Configuration: in hibernate.properties or hibernate.cfg.xml files; represents an entire set of mappings of an application Java Types to an SQL database.
- Session Factory: User application requests Session Factory for a session object
- Session: Interaction between the application and the database (org.hibernate.Session class)
- Query: Query the database for one or more stored objects using NamedQuery and Criteria API
- First-level cache: Default cache used by Hibernate Session object while interacting with the database
- Transaction: Enables data rollback as needed
- Persistent objects: Plain Old Java Objects (POJOs) persisted as one of the rows in a related table in the database by Hibernate
- Second-level cache: Used to store objects across sessions; uses a cache provider (ex. EhCache)
- https://howtodoinjava.com/hibernate-tutorials/



Typical Java Hibernate Example

- Three parts shown here:
- Establishing the data storage for an EmployeeEntity
- Configuring the Hibernate Session Factory
- Using it to store an object

- https://howtodoinjava.com/hibernate/hibernate-3-introduction-and-writing-hello-world-application/

```
package hibernate.test.dto;

import java.io.Serializable;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;
import javax.persistence.UniqueConstraint;

import org.hibernate.annotations.OptimisticLockType;

@Entity
@org.hibernate.annotations.Entity(optimisticLock = OptimisticLockType.ALL)
@Table(name = "Employee", uniqueConstraints = {
      @UniqueConstraint(columnNames = "ID"),
      @UniqueConstraint(columnNames = "EMAIL") })
public class EmployeeEntity implements Serializable {

   private static final long serialVersionUID = -1798070786993154676L;

   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   @Column(name = "ID", unique = true, nullable = false)
   private Integer employeeId;

   @Column(name = "EMAIL", unique = true, nullable = false, length = 100)
   private String email;

   @Column(name = "FIRST_NAME", unique = false, nullable = false, length = 100)
   private String firstName;

   @Column(name = "LAST_NAME", unique = false, nullable = false, length = 100)
   private String lastName;

   // Accessors and mutators for all four fields
}
```

Typical Java Hibernate Example

- Three parts shown here:
- Establishing the data storage for an EmployeeEntity
- Configuring the Hibernate Session Factory
- Using it to store an object

- https://howtodoinjava.com/hibernate/hibernate-3-introduction-and-writing-hello-world-application/

```
package hibernate.test;

import java.io.File;

import org.hibernate.SessionFactory;
import org.hibernate.cfg.AnnotationConfiguration;

public class HibernateUtil
{
   private static final SessionFactory sessionFactory = buildSessionFactory();

   private static SessionFactory buildSessionFactory()
   {
      try {
         // Create the SessionFactory from hibernate.cfg.xml
         return new AnnotationConfiguration().configure(
               new File("hibernate.cgf.xml")).buildSessionFactory();
      } catch (Throwable ex) {
         // Make sure you log the exception, as it might be swallowed
         System.err.println("Initial SessionFactory creation failed." + ex);
         throw new ExceptionInInitializerError(ex);
      }
   }

   public static SessionFactory getSessionFactory() {
      return sessionFactory;
   }

   public static void shutdown() {
      // Close caches and connection pools
      getSessionFactory().close();
   }
}
```

Typical Java Hibernate Example

- Three parts shown here:
- Establishing the data storage for an EmployeeEntity
- Configuring the Hibernate Session Factory
- Using it to store an object

- https://howtodoinjava.com/hibernate/hibernate-3-introduction-and-writing-hello-world-application/

```
package hibernate.test;

import hibernate.test.dto.EmployeeEntity;
import org.hibernate.Session;

public class TestHibernate {

    public static void main(String[] args) {
        Session session = HibernateUtil.getSessionFactory().openSession();
        session.beginTransaction();

        // Add new Employee object
        EmployeeEntity emp = new EmployeeEntity();
        emp.setEmail("demo-user@mail.com");
        emp.setFirstName("demo");
        emp.setLastName("user");

        session.save(emp);

        session.getTransaction().commit();
        HibernateUtil.shutdown();
    }
}
```

Alternative: jOOQ

- jOOQ – Java Object Oriented Querying
- Focuses on using SQL for table creation and complex queries in a near SQL syntax for its DSL (Domain Specific Language)
- Generates Java representation of database scheme, uses JDBC to call SQL queries
- Not an ORM – 1:1 mapping to underlying SQL elements

Standard SQL Query

```
SELECT AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME,
COUNT(*)
FROM AUTHOR
JOIN BOOK ON AUTHOR.ID = BOOK.AUTHOR_ID
WHERE BOOK.LANGUAGE = 'DE'
AND BOOK.PUBLISHED > DATE '2008-01-01'
GROUP BY AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME
HAVING COUNT(*) > 5
ORDER BY AUTHOR.LAST_NAME ASC NULLS FIRST
LIMIT 2
OFFSET 1
```

jOOQ Query

```
create.select(AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME,
count())
    .from(AUTHOR)
    .join(BOOK).on(AUTHOR.ID.equal(BOOK.AUTHOR_ID))
    .where(BOOK.LANGUAGE.eq("DE"))
    .and(BOOK.PUBLISHED.gt(date("2008-01-01")))
    .groupBy(AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME)
    .having(count().gt(5))
    .orderBy(AUTHOR.LAST_NAME.asc().nullsFirst())
    .limit(2)
    .offset(1)
```

Alternative: ODBMS

- Another approach is to use a true Object Oriented database system
  - Not generally used to the level of RDBMS and ORMs
- Besides the OO elements you'd expect, an ODBMS must provide for
  - Persistance, storage management, recovery, and query approaches
  - Concurrency and ACID principles – atomicity, consistency, isolation, and durability
- The ODMG – Object Database Management Group – standards body for looking at object models, object definition language, object query language, and language bindings
- Examples: Caché, DB4o, ObjectDB, ObjectStore, others
- Java examples of DB4o: https://www.ibm.com/developerworks/library/j-db4o2/index.html
- http://www.odbms.org/wp-content/uploads/2013/11/lecture_12_objectDatabases.pdf
- https://db-engines.com/en/ranking/object+oriented+dbms

Should you use an ORM?

- Points by Martin Fowler from his article “OrmHate”: https://martinfowler.com/bliki/OrmHate.html
- He feels most hate against ORMs is unwarranted
  - Comments that they are complex, hard to use, perform poorly, or have leaky abstraction
- He points out object/relational mapping is hard
  - synchronizing two very different representations of data (Java in-memory vs. data being stored in an RDBMS)
- Leaky abstraction is somewhat true – you’re repeating some representation of the data – but you’re gaining in the operations against that data
- And you still need to understand and be responsible for the data design for your applications – a good data model, relational or no-SQL, is essential
  - Using an ORM is not an excuse for not understanding SQL
- No one answer for every application
- If you need to map to objects to relational data, use an ORM, you’re better off