Hibernate Tutorial Notes
A framework for persisting / saving java objects in a database

ORM — object-to-relational mapping
- the developer defines mapping between java class and database table

CRUD — Create - Read - Update - Delete

- **Hibernate vs. JDBC** ?

- hibernate uses JDBC for all database communications

First of all create database with:

```
       String jdbcUrl = "jdbc:mysql://localhost:3306/hb_student_tracker?
user55L=false";
       String user = "hbstudent";
       String pass = "hbstudent";
```

need a hibernate config file
—> java annotations

Entity Class — Java class that is mapped to a database table

- **Java Annotations**

1. map class to database — @Table — on top of object class
2. map fields to database columns — @Column(name="column_name") — on fields
    (note need @Id on id field)
    (if column name == field name, then annotation not needed)

SessionFactory
- Reads the hibernate config file
- Create Session objects
- Heavy-weight object, meaning only create once in app

Session
- Wraps a JDBC connection
- Main object used to save/retrieve objects
- Short-lived object
- Retrieved from SessionFactory

****** Code: hibernate-tutorial/…/CreateStudentDemo.java ******

Primary Key (**e.g. id**)
- Unique identifies each row in a table
- Must be a unique value

    - Cannot contain NULL values
- **@GeneratedValue(strategy=GnerationType. …)**
    - AUTO — pick an appropriate strategy for the particular data
    - IDENTITY — assign primary keys using identity column
    - SEQUENCE — assign primary keys using a database sequence
    - TABLE — assign primary keys using an underlying database table to ensure uniquencess
    - can also customize strategy
        - create subclass org.hibernate.id.DequenceGenerator
        - override method: public Serializable generate(…)
            - much to worry about

- **Modify auto-increase**

1. SQL bench: ALTER TABLE hb_student_tracker.student auto_increment=3000 —> id start from 3000
2. reset table to blank: truncate hb_student_tracker.student

- Retrieve a java object with hibernate

**** **Code: hibernate-tutorial/…/ReadStudentDemo.java** ****

- **Query objects**
    - Query language for retrieving objects
    - similar in nature to SQL

****** Code: hibernate-tutorial/…/QueryStudentDemo.java ******

- **Update objects**
    - single row
    - multiple rows

****** Code: hibernate-tutorial/…/QueryStudentDemo.java ******

- **Delete objects**

**** **Code: hibernate-tutorial/…/DeleteStudentDemo.java ******

Project

Customer Relationship Management (CRM)

- List customer
- add customer
- update customer
- delete customer

DAO — data access object — helper class to access database

- **Some useful annotations:**
    - **@Transactional — automatically call begin and end transaction**
    - **@Repository — DAO implementations**
        - **automatically register the DAO implementation**

        - **spring also provides translation of any JDBC related exceptions**

- **RequestMapping method**
    - **GET: (@GetMapping("/…")**
        - **good for debugging**
        - **bookmark or email URL**
        - **limitations on data length (1000 char)**
    - **POST: (@PostMapping("/…")**
        - **can't bookmark or email URL**
        - **no limitations on data length**
        - **can also send binary data**

- **Service layer**
    - **service facade** design pattern
    - intermediate layer for custom business logic
    - integrate date from multiple sources (DAO/repositories)
    - **annotaion: @Service**

1. define service interface
2. define service implementation
    1. inject the customerDAO

Service will manage transaction

AOP — Aspect-Oriented Programming

- **Advantages:**
    - **reusable**
    - **resolve code tangling**
    - **resolve code scatter**
    - **applied selectively based on configuration**
- **Disadvantages:**
    - **too many aspects and app flow is hard to follow**
    - **minor performance cost for aspect execution**

- **Add logging code**

- **AOP Terminologies**
    - **Aspect — module of code for a cross-cutting concern (logging, security, …)**
    - **Advice — what action is taken and when it should be applied**
    - **Join Point — when to apply code during program execution**
    - **Pointcut — a predicate expression for where advice should be applied**
- **Advice Types**
    - **Before advice — run before the method**
    - **After finally advice — run after the method (finally)**

    - After returning advice — run after the method (success execution)
    - After throwing advice — run after method (if exception thrown)
    - Around advice — run before and after method
- Weaving
    - connecting aspects to target objects to create an advised object
    - Different types of weaving
        - compile-time
        - load-time
        - run-time
    - Regarding performance: run-time weaving is the slowest
- AOP Frameworks
    - Two leading AOP frameworks for java
        - Spring AOP
        - AspectJ
- Spring AOP Support
    - spring provides AOP support
    - key component of Spring
        - Security, transactions, caching etc
    - Uses run-time weaving of aspects
- AspectJ
    - original AOP framework
    - provide complete support for AOP
    - rich support for
        - joint points: method-level, constructors, field
        - code weaving: compile-time, post compile-time and load-time
- Spring AOP Comparison
    - Advantages:
        - simpler to use than aspectJ
        - use proxy pattern
        - can migrate to aspectJ when using @Aspect annotation
    - Disadvantages:
        - only supports method-level join points
        - can only apply aspects to beans created by spring app context
        - minor performance cost for aspect execution (tun-time weaving)
- AspectJ Comparison
    - Advantages: support all join points
    - works with any POJO not just beans from app context
    - faster performance compared to spring app
    - complete AOP support
- Disadvantages:
    - compile-time weaving requires extra compilation step
    - aspectJ pointcut syntax can become complex

- AOP @Before Advice [@Beofore("execution(modifier(optional) return_type complete_method_directory))")] (* means any)

- Most common use
    - logging, security, transaction
- audit logging
    - who, what, when, where
- API management
    - how many times has a method been called user
    - analytics: what are peak times? what is average load? who is top user?

Pointcut — A predicate expression for where advice should be applied
use pointcut expression language

- Parameter Pattern Wildcards
    - for param-pattern
        - () — matches a method with no args
        - (*) — matches a method with one arg of any type
        - (..) — matches a method with 0 or more args of any type
- Pointcut Expression Examples
    - match on method params
        - match addAccount methods with no args
            - @Before("execution(* addAccount())")
        - match addAccount methods that have Account param
            - @Before("execution(* addAcount(com.larry.aopdemo.Account))")
        - match addAccount methods with any number of args
            - @Before("execution(* addAccount(..))")
        - match on methods in a package
            - @Before("execution(* com.larry.aopdemo.dao.*.*(..))")