Hibernate

Hibernate

- an ORM framework for Java ORM: object-relational mapper; a solution for the object-relational mismatch. maps data stored relationally in a RDB to an object-oriented structure.
- written by Gavin King in 2002 first, he wrote Hibernate, then Oracle took him on to write the JPA JPA: Java Persistence API; interfaces meant for mapping/managing relational data with objects after he wrote the JPA, he rewrote Hibernate to implement it
- used to be xml-based, now is annotation-based it is still possible to map your beans using xml, but it is not standard annotations are easier to use because they go directly within the bean; no need to go back and forth referencing files. because of this, they are also more readable and more maintainable.
- hibernate.cfg.xml: the Hibernate configuration file in src/main/resources important info/metadata that goes in this file (and sets up the SessionFactory): - database connection info (info that would previously go in database.properties) - hibernate setup (SQL dialect, SQL comments, DDL-auto) - object mappings (either references to the annotated classes or to hbm.xml files)
- mapping
    - with xml, <mapping resource="BeanName.hbm.xml> in hibernate.cfg.xml
    - with annotations, in hibernate.cfg.xml xml mapping: BeanName.hbm.xml

mapping annotations imported from JPA unless otherwise specified...

- @Entity: must go at the top of a bean class so that Hibernate recognizes it as a mapped entity
- @Table: must go at the top of a class that is mapped to a table in the database
    - optional @Table(name="bean_name") if name in DB is different than the object name
- @Id: goes above the primary key field
- @SequenceGenerator(name="seqGen", sequenceName="bean_seq", allocationSize=1)
    - name is an arbitrary name for the sequence.
    - it is only referenced in the GeneratedValue annotation sequenceName is the name of the sequence in the database allocationSize is how much the sequence increments when you create an object
- @GeneratedValue(generator="seqGen", strategy=GenerationType.SEQUENCE) generator is the name given in the @SequenceGenerator annotation
- @GeneratedValue(strategy=GenerationType.IDENTITY) strategy is how the incrementation occurs (like if your RDB uses auto-incrementation, etc)
- @Column: implicitly is above all fields in classes annotated with @Table
    - whether or not it is best practice to use it explicitly or not seems to be debated
    - only NEED to use it specifically if column name is different in DB than field name, typically for multi-word fields (because Java naming conventions are camelCase and SQL naming conventions are snake_case) @Column(name="column_name")
- @Transient: over a field that should not be persisted to the database

    - this is for if you have some sort of info about an object that you use temporarily in the front/back end logic but does not need to be saved and is not mapped to anything in the database.
    - for example, in TRMS, this could be used for the amount received for a request after the event type percentage covered calculation is performed, because this calculation can easily be repeated as needed; there is no need for it to be in the database because all necessary info is already there
- @OneToMany/@ManyToOne/@OneToOne: over a field that is an object/collection of objects from another table
    - which one to use depends on what the relationship is between the objects.
    - for example, one person owns many cats, but each cat only has one owner, so person to cat is one-to-many. in the Person bean, @OneToMany would go over the collection of Cat field (Set cats)
    - can include fetch and cascade properties @OneToMany(fetch=FetchType.EAGER, cascade=CascadeType.ALL)
- @JoinColumn(name="column_name"): goes with the above multiplicity relationship annotations.
    - specifies which column (foreign key) in the database connects these objects.
    - use the JavaDocs to clarify which table the column should be coming from.
- @ManyToMany: over a field that is a collection of objects from another table
    - for example, a cat can have many special needs, and a special need can be associated with many cats so @ManyToMany would go over the Cat's collection of SpecialNeed objects.
- @JoinTable(name="table_name", joinColumn=@JoinColumn(name="column_name"), inverseJoinColumn=@JoinColumn(name="column_name"))
    - goes with multiplicity annotations that use a join table/multiplicity table in the DB (usually this is many-to-many, but not necessarily: see person's cats in CatApp2)
    - joinColumn references the foreign key to the "owner" of the relationship (e.g. person owns cats) inverseJoinColumn references foreing key to what is "owned" (e.g. cats owned by person)
- BEWARE OF RECURSIVE MAPPING (infinite loops) this happens in a situation where two objects reference each other. for example, if a Person has a set of Cats, but a Cat has a Person, this will be a recursive mapping to fix this:
    - have the fetch type be LAZY for at least one side of the relationship
    - avoid referencing the full object (only ID) in hashcode, equals, & toString methods

SessionFactory:

- gets a Session
- Hibernate interacts with the database through Sessions (sort of like us using the Connection object with plain JDBC) set up as singleton in HibernateUtil

cascade

- setting cascades for relationships between objects allows all connected objects to be affected by certain operations, i.e. all operations cascade from the parent object to whatever objects it references through foreign keys

LAZY vs. EAGER

- when objects are fetched eagerly, the entire object is retrieved as soon as the parent object is retrieved
- when they are fetched lazily, only a proxy is retrieved until the object is used within the session

proxy:

- an object which is empty aside from its unique identifier (primary key)

Hibernate object states

- transient:
    - an object that does not yet match what is in the database
    - this could be a newly created object that does not yet have an ID associated with it,
    - or it could be a newly updated object whose updates have not be written to the database yet
- persistent:
    - an object that is associated with a session (in cache) and matches what is in the database
        - this could be an object that was just pulled from the database,
        - or was just created an assigned a primary key (ID) value upon doing so.
- detached:
    - an object that is not in a session, but matches what is in the database
        - this could be an object that was pulled from the database or whose updates were just written to the database, then the session was flushed/closed.
        - for example, the person object after a user is already logged in.

Hibernate session cache

- automatic dirty checking: hibernate tracks the state of an object (whether it is different from what is in the DB) then runs any necessary updates (DML) when the session is flushed.
    - can force session flush with .flush() method. this takes stress off the database because fewer separate statements are run, fewer separate connections are made, etc.
        - if an object is persistent (associated with a session, already exists in DB), any necessary updates are executed automatically when session is flushed (no .update() needed)
    - info about persistent objects are stored in the session cache
    - 1st level cache (L1 cache): cache for a single session, included by default

    - 2nd level cache (L2 cache): cache between sessions, requires 3rd party software (e.g. Ehcache)

DAO/CRUD methods

- `.get() vs .load()`
    - get will retrieve the full object from the DB and return null if the object doesn't exist
    - load will retrieve a proxy & throw an exception if the object doesn't exist
- `.update() vs .merge()`
    - update will make the object persistent: takes a transient object (that is NOT in a session) and make it persistent; actually changes what is in the database. throws exceptions
    - merge will take a persistent object with changes that would be written on session flush and copies the DB data to the object
- `.save()`
    - runs an insert statement for the object, assigns primary key to the object in Java
- `.persist()`
    - will run an insert statement on session flush/close or commit, only on transient objects
- `.delete()`
    - removes the object from the DB
- `.saveOrUpdate()`
    - will save the object if it is not in the DB (no matching ID), otherwise will update

Writing DML/DQL

- native queries:
    - just writing the plain SQL paired with NativeQuery object
    - parameter syntax is :paramName
    - not best practice to mix languages (Java & SQL) if not necessary, will need changed if SQL dialect changes

HQL: Hibernate Query Language

- allows DQL
- SQL dialect does not matter (is portable)
- more object-oriented than regular SQL
- paired with Query object
- `String s = "from Person"; // pulls everything mapped in Person class`
- `Query<Person> q = session.createQuery(s, Person.class);`
- `List<Person> p = q.getResultList();`

Criteria API

- Java-based
- for querying data programmatically
- most separated from regular SQL

- no language-mixing - pure Java
- keeps code very object-oriented
- can be hard to learn/read in unfamiliar
- only DQL

named queries

```
@NamedQueries({@NamedQuery(name="getAll", query="from Person") //above
class
public class PersonClass {}

Query<Person> q = session.createNamedQuery("getAll", Person.class) //in
DAO method
```

- these are good for when you are not using a designated DAO layer to separate all data access
    - as they allow you to keep often-used queries in an easy-to-find place (with their relevant bean)
- they may also be useful for using native queries or other queries that may need to be changed at some point, so they are easy to find when those changes must occur.