Hibernate

* ORM framework
  + Object Relational mapping
* Hibernate has the ability to connect entities in Java to tables in SQL
* XML configuration or annotations
  + Annotations are better.
* Built on top of JDBC.
  + It does use JDBC under the hood.
* Advantages
  + Faster to set up and write DAOs with
  + Be SQL dialect agnostic.
    - Once you set up hibernate it will work for any SQL database.
  + Has support for multiple ways to query your database.

Key Interfaces of Hibernate

* SessionFactory
  + Creates our connection to a database.
  + Reads your hibernate mapping configuration to generate the SQL statements that reflect those mappings.
* Session
  + Primary interface for performing CRUD operations
    - Save()
    - Update()
    - Delete()
    - Get()
      * Designed to eager fetch data
    - Load()
      * Designed to lazy fetch data
* Transaction
  + Used to start and end database transactions,
  + Allows us to programmatically decide in Java when to commit or rollback a set of hibernate commands.
* Query
  + Used to create queries
    - Criteria API
      * Very OOP centric.
      * Criteria interface that you can add restrictions too.
      * Then execute that criteria to get records that match it.
    - HQL
      * Hibernate Query Language
      * OOP hybrid of SQL and Java
    - Native SQL
      * A regular SQL statement
      * Avoid if possible.
* Configuration\*\*\*\*\*\*\*\*\*\*
  + Used to store information about how to connect to the database.
  + This is what the SessionFactory uses to build itself.
  + By default it will look at the hibernate.cfg.xml to get information.
  + It is actually an object

Hibernate.cfg.xml

* Main configuration file for hibernate
  + Db location
  + Username
  + Password
  + Classes that should be mapped
  + Show SQL
  + SessionFactory modes
    - Create-drop
    - Create
    - Alter
    - Nothing

Annotations

* Annotations are used to map entities to tables.
* Set up relationships between entities.
* Important annotations
  + @Entity
    - Marks a class as an entity
  + @Table
    - What table this class is mapped to
  + @Column
    - What column this field is mapped to
  + @Id
    - Put over fields that are primary keys.
  + @GeneratedValue
    - Put over fields that have generated values.
    - (strategy=StrategyType.IDENTITY)
  + @JoinColumn
    - Put over fields that are also a foreign key
  + @OneToMany
    - (mappedBy=”the field in JAVA that has @JoinColumn on it”)
    - Put over a set in an Entity that has children
      * Directory => Movie
  + @ManyToMany
    - Put over a set in an entity in a many to many relationship with another entity
      * Actors ⬄Movie
  + @JoinTable
    - Put over a set in an entity in a many to many relationship with another entity.
    - @JoinColumn
      * Refers back the current entity
    - inverseJoinColumns
      * What columns refer to other entity you are connecting to.

States of an object in Hibernate

* Transient
  + Object that is not saved to the database.
  + Has an id of 0.
  + Has no corresponding record in the database.
* Persistent
  + Object that is saved to the database.
  + It has a non 0 id.
  + It is still in an active session.
* Detached
  + Object that corresponds to a record in the database.
  + It has a non 0 id.
  + It was attached to an active session but that session has since been closed.

Cascading in Hibernate

* Objects can cascade in Hibernate.
  + You can create a new transient movie.
  + You can add it to a persistent director.
  + You update the director.
  + The movie is also saved and is added to the database.
* Define what cascades you want
  + Persist cascaded (No nested objects can be deleted only updated/or created)
    - You could add a transient movie to a director and save it
    - You could remove a movie from a directory and then update the director.
      * Movie will NOT be deleted in the database
  + Remove cascaded (Nested Objects can be deleted)

ACID properties of Transactions

* Atomic
  + A transaction either works or it does not.
  + There is no half worked.
  + All the statements get committed at the end of a transaction.
  + OR all the statements get rolled back.
* Consistency
  + The database is never in a saved state that is inconsistent.
  + The database moves between consistent states without an persistent intermediary state.
  + Quantum leaps
    - One state to another.
    - There was never really an intermediary.
* Isolated

| **Isolation Level** | **Dirty Read** | **Non Repeatable Read** | **Phantom** |
| --- | --- | --- | --- |
| Read uncommitted | Yes | Yes | Yes |
| Read committed | No | Yes | Yes |
| Repeatable read | No | No | Yes |
| Serializable | No | No | No |

* + In reality a database will be used by many users simultaneously.
  + Serializable
    - No concurrent transaction allowed.
    - They just queue up.
    - 100% consistency guaranteed.
  + As your database gets higher isolation levels the rate of processing transactions decreases because transaction do not run concurrently anymore.
* Durable
  + Any failures should be graceful and not result in a loss of data
    - A transaction failing for any reason should NOT cause any problems
    - No data loss.
    - No data corruption.
    - It’s like it never happened.