**Summary – Hibernate**

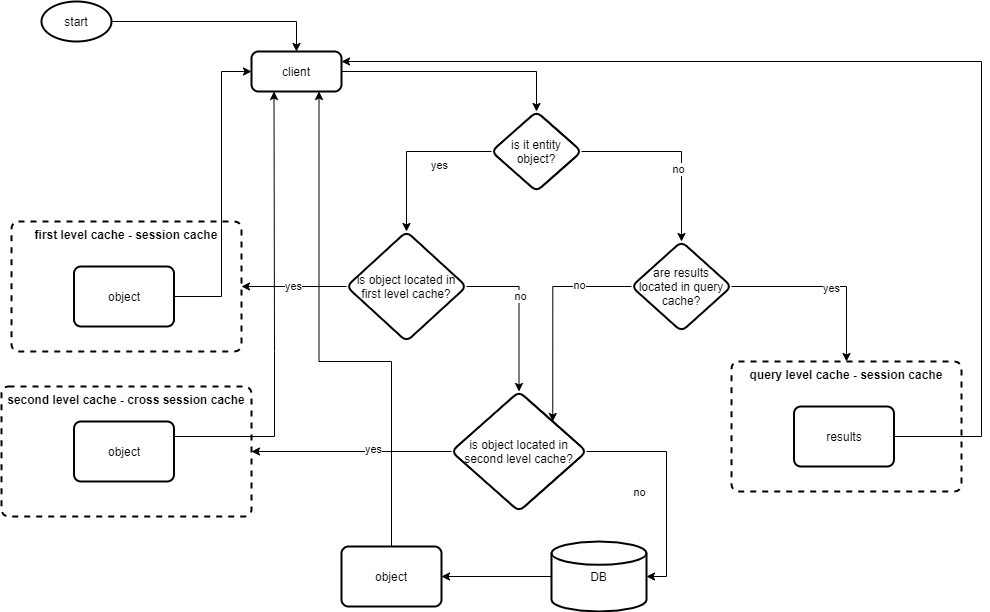
Setup Hibernate:

1. Download the latest hibernate release bundle and put it in a hibernate folder in your PC.
2. Download some DB (in my case postgresql). (user name: "postgres", password: "password", port: 5432)
3. Download postgresql jdbc driver and add it to the project.
4. Download the Postgresql folder to your PC. in my case it's located in: C:\Program Files (x86)\PostgreSQL.
5. To create the connection to the db we need to create a file with a specific name which is: "hibernate.cfg.xml" under resources folder in our project. To fill this file, we need to go to hibernate folder and look for other files with the same name and copy the content of one of them to the file that we created.
6. In the file that we created, modify DB connection settings and SQL dialect properties.
7. Go to pgadmin (which is the postgresql admin) and create db with the same name that mentioned in the above file.
8. To make sure that DB will not be create each time I open a session and commit objects and thus delete all of the data I committed till that point, we need to go to the xml and change property hbm2ddl.auto to update.

Add table and columns:

1. Create a dto model (Object) and annotate the class with @Entity (from javax.persistence)
2. Annotate one field as ID with @Id (from javax.persistence)
3. Define this dto in hibernate.cfg.xml file under mapping element
4. Create session factory that will create a session that will hold transaction that will save the dto and close the transaction

Hibernate caching

Hibernate provides 3 levels of cache: 

1. **First level cache: Session-scoped**, meaning, for example, when I am opening a session and doing a select query and then update and then select query again. There will be only one select query that's executed since the data is cached in session. If I will close the above session and open a new session and perform a select query, two select queries will be executed.
2. **Second level cache: SessionFactory-scoped**, meaning it is shared by all sessions created with the same session factory.
3. **Query level cache**, meaning caches the results from previous queries done by Native SQL or HQL.

Second level cache configuration

1. We need to configure it in hibernate.cfg in element hibernate.cache.region.factory\_class to "org.hibernate.cache.ehcache.EhCacheRegionFactory".
2. We need to tell Hibernate to switch the second level cache; we do that by adding an element "cache.use\_second\_level\_cache": true.
3. We need to add dependencies to maven: hibernate-ehcache and ehcache-core
4. We need to configure the required entity to be cacheable we do that by adding the @Cacheable annotation.
5. We need to add the caching strategy by adding @Cache(usage=CacheConcurrencyStrategy.READ\_ONLY) so Hibernate will know how to cache the object

Special annotations

General

@Entity

* **Applies for**: class
* **Significance**: marks this class as an entity bean.

@Table

* **Applies for**: class
* **Significance:** allows you to specify the details of the table that will be used to persist the entity in the database.

provides four attributes, allowing you to override the name of the table, its catalogue, and its schema, and enforce unique constraints on columns in the table.

@NamedNativeQuery(name="UserDetails.byName", query="select \* from user\_details where user\_name = ?", resultClass=UserDetails.class)

* **Applies for**: class
* **Significance:** allows you to save a native query by name and then use it when needed.
* **Example:**

NativeQuery anotherQuery = session.getNamedNativeQuery("UserDetails.byName");

anotherQuery.setParameter(1, "other name");

List<UserDetails> otherUsers = anotherQuery.getResultList();

@Id

* **Applies for**: field
* **Significance:** declares the identifier property of this entity bean, equal to primary key.

@GeneratedValue

* **Other must-have annotations**: @Id
* **Applies for**: field
* **Significance:** generate a unique id for each record

@Column(name)

* **Applies for**: field
* **Significance:** defines the table’s column

@Temporal(TemporalType)

* **Applies for**: field
* **Significance:** defines the level of precision of the time object that will be saved to DB (date, date and time etc.)

@Transient

* **Applies for**: field
* **Significance:** tells hibernate to ignore this field when creating the table so it will not create a corresponding column

@Lob

* **Applies for**: field
* **Significance:** defines very big objects that saves a huge amount of text in the column.

@GenericGenerator(name, strategy)

* **Applies for**: field
* **Significance:** allows you to define an Hibernate specific id generator for table id.

@NotFound(action)

* **Applies for:** field
* **Significance:** when field is null in DB, Hibernate will throw an exception. We can ignore this exception using this annotation.

Caching

@Cacheable

* **Applies for**: class
* **Significance:** makes an entity eligible for second-level caching.

@Cache(usage)

* **Applies for**: class
* **Significance:** mentions the caching strategy for the entity. (read only, read- write, etc.).

Associasion

@EmbeddedId

* **Applies for**: field
* **Significance:** defines an embedded object as a primary key. this will create a primary key from the combination of the fields in the object

@Embeddable

* **Applies for:** bean
* **Significance:** makes this object's fields embedded as columns at another object's table.

@Embedded

* **Applies for:** field
* **Significance:** makes this field’s object embedded as columns at the entity object's table.

@AttributeOverrides and @AttributeOverride(name, column)

* **Applies for**: field
* **Significance:** when we have more than one embedded object of the same type, we need to give the columns of one object different names.

@ElementCollection(fetch)

* **Applies for**: field
* **Significance:** we use thisannotationwhen we have some collection as a field, and we must create for it a separate table that’s related to the entity with a foreign key. It has 2 fetch strategies: lazy(default) and eager. this means that by default, any collection that is a member of an entity class will not be fetched when fetching the entity object from DB. it will be fetched only when calling its getter method since it saves time and resources. to tell Hibernate that we want to fetch the collection when entity object is fetched, we must use the eager strategy.

@JoinTable(name, joinColumns, inverseJoinColumns) and @JoinColumn(name)

* **Applies for**: field
* **Significance:** defines a name of table that overrides the default name that Hibernate gives.
  1. Name property is the name of the table
  2. joinColumns property is the name of the generated column in the entity that the field located in
  3. inverseJoinColumns property is the name of the foreign key in the new table.
* **Example:**

@Entity

public class Customer implements Serializable {

@JoinTable(name = "CustomerPassports",

joinColumns = @JoinColumn(name="customer\_fk"),

inverseJoinColumns = @JoinColumn(name="passport\_fk")

)

private Passport passport;

}

@CollectionId(columns, generator, type)

* **Applies for**: field
* **Significance:** we use thisannotationwhen we have some collection as a field, and we must create for it a separate table that’s related to the entity with a foreign key. It allows us to define a primary key, its type and generator.

@OneToOne(mappedBy, cascade)

* **Applies for**: field
* **Significance:** defines one to one relationship.There are three cases for one-to-one associations:
  1. associated entities share the same primary keys values
  2. a foreign key is held by one of the entities (note that this FK column in the database should be constrained unique to simulate one-to-one multiplicity)
  3. an association table is used to store the link between the 2 entities (a unique constraint must be defined on each fk to ensure the one to one multiplicity)

regarding the properties:

* 1. mappedBy defines the name of the object that’s holding the fk. It’s put on the entity with the fk.
  2. Cascade defines which DB operation we want to cascade to related entities of the current entity without doing the same operations on them also
* **Example:**

Case 1:

@Entity

public class Body {

@Id

private Long Id;

@OneToOne(cascade = CascadeType.ALL)

@PrimaryKeyJoinColumn

private Heart heart;

}

@Entity

public class Heart {

@Id

private Long Id;

}

Case 2:

@Entity

public class Customer implements Serializable {

@OneToOne(cascade = CascadeType.ALL)

@JoinColumn(name="passport\_fk")

private Passport passport;

}

@Entity

public class Passport implements Serializable {

@OneToOne(mappedBy = "passport")

private Customer customer;

}

Case 3:

@Entity

public class Customer implements Serializable {

@OneToOne(cascade = CascadeType.ALL)

@JoinTable(

name = "CustomerPassports",

joinColumns = @JoinColumn(name="customer\_fk"),

inverseJoinColumns = @JoinColumn(name="passport\_fk")

)

private Passport passport;

}

@Entity

public class Passport implements Serializable {

@OneToOne(mappedBy = "passport")

private Customer customer;

}

@OneToMany

* **Applies for:** field
* **Significance:** defines one to many relationship. There are three cases for one-to-many associations:
  1. Bidirectional – both entities are annotated
  2. Unidirectional – one entity is annotated
  3. Unidirectional with join table – one entity is annotated, and a joined table is created
  4. Default - Unidirectional with join table
* **Example:**

Case 1:

@Entity

public class Troop {

@OneToMany(mappedBy="troop")

public Set<Soldier> soldiers;

}

@Entity

public class Soldier {

@ManyToOne

@JoinColumn(name="troop\_fk")

public Troop troop;

}

Case 2:

@Entity

public class Customer implements Serializable {

@OneToMany(cascade=CascadeType.ALL, fetch=FetchType.EAGER)

@JoinColumn(name="CUST\_ID")

public Set<Ticket> tickets;

}

Case 3:

@Entity

public class Trainer {

@OneToMany

@JoinTable(

name="TrainedMonkeys",

joinColumns = @JoinColumn( name="trainer\_id"),

inverseJoinColumns = @JoinColumn( name="monkey\_id")

)

public Set<Monkey> trainedMonkeys;

}

Case 4:

@Entity

public class Trainer {

@OneToMany

public Set<Tiger> trainedTigers;

}

@ManyToOne

* **Applies for:** field
* **Significance:** defines many to one relationship. There are two cases for one-to-one associations:
  1. With joined column
  2. With joined table
* **Example:**

Case 1:

@Entity()

public class Flight implements Serializable {

@ManyToOne( cascade = {CascadeType.PERSIST, CascadeType.MERGE})

@JoinColumn(name="COMP\_ID")

public Company company;

}

Case 2:

@Entity()

public class Flight implements Serializable {

@ManyToOne( cascade = {CascadeType.PERSIST, CascadeType.MERGE} )

@JoinTable(name="Flight\_Company",

joinColumns = @JoinColumn(name="FLIGHT\_ID"),

inverseJoinColumns = @JoinColumn(name="COMP\_ID")

)

public Company company;

}

@ManyToMany

* **Applies for:** field
* **Significance:** defines many to many relationship. There are two cases for many-to-many associations:
  1. Bidirectional – both entities are annotated
  2. Unidirectional – one entity is annotated
* **Example:**

Case 1:

@Entity

public class Employer implements Serializable {

@ManyToMany(

targetEntity=org.hibernate.test.metadata.manytomany.Employee.class,

cascade={CascadeType.PERSIST, CascadeType.MERGE}

)

@JoinTable(

name="EMPLOYER\_EMPLOYEE",

joinColumns=@JoinColumn(name="EMPER\_ID"),

inverseJoinColumns=@JoinColumn(name="EMPEE\_ID")

)

public Collection employees;

}

@Entity

public class Employee implements Serializable {

@ManyToMany(

cascade = {CascadeType.PERSIST, CascadeType.MERGE},

mappedBy = "employees",

targetEntity = Employer.class

)

public Collection employers;

}

Case 2:

@Entity

public class Store {

@ManyToMany(cascade = CascadeType.PERSIST)

public Set<City> cities;

}

inheritance

@Inheritance(strategy)

* **Applies for:** bean
* **Significance:** indicate that there are three strategies we can take:
  1. Table per class - a separate table for each class (father and children).
  2. Single table – one table for father and children. Different children are distinguished by a discriminator column.
  3. Joined - this will create the children's tables with only the children's specific fields and the father's table with the father's fields. this is the most normalized way of inheritance.

@DiscriminatorColumn(name, discriminatorType)

* **Other must-have annotations**: @Inheritance
* **Applies for:** bean
* **Significance:** defines the value used to differentiate a class in the hierarchy. Used when @Inheritance strategy is single table.