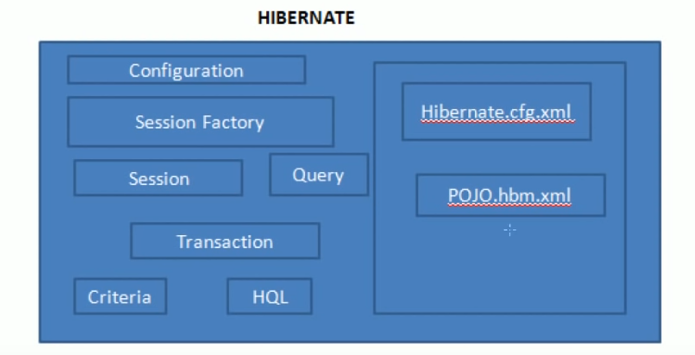
An ORM(Object Relational Mapping) tool , its framework for persistence(save) data to DB.

<https://academy.vertabelo.com/blog/illustrated-guide-multiple-join/>

**Why Hibernate?**

* Mapping member variable to columns.
* Mapping relationship.
* Handling Data Type.
* Managing changes to object state.
* Relation way Vs Object oriented way.

Hibernate Architecture:



ORM(Object Relational Mapping)

Hbm (Hibernate Bean Mapping)

Ddl (data definition language)

1. Hibernate Configuration: hibernate.cfg.xml

<?xml version=*"1.0"* encoding=*"utf-8"*?>

<!DOCTYPE hibernate-configuration SYSTEM

"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

<session-factory>

<property name=*"hibernate.dialect"*>org.hibernate.dialect.Oracle10gDialect</property>

<property name=*"hibernate.connection.driver\_class"*>oracle.jdbc.driver.OracleDriver </property>

<property name=*"hibernate.current\_session\_context\_class"*>thread</property>

<!-- List of XML mapping files -->

<mapping resource=*"GcpSareaTmpData.hbm.xml"*/>

</session-factory>

</hibernate-configuration>

Some of properties:

Dialect : is used to which Database(DB)(MySQL, oracle etc) should hibernate to communicate with DB.

Driver\_class: is used to connect to DB.

Show\_sql : print the sql query.

Hbm2ddl.auto : create/update/none //(create) if table is not present it will create table or Drop table and recreate it. (update) will only update table if there is change Entity properties. Production we have to use update only.

1. Create POJO class: with default constructor and Setter & getter (if using Annotation based then add the Annotation mainly @Entity and @Id)
2. Mapping Entity: GcpSareaTmpData.hbm.xml

<?xml version=*"1.0"* encoding=*"utf-8"*?>

<!DOCTYPE hibernate-mapping PUBLIC

"-//Hibernate/Hibernate Mapping DTD//EN"

"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping>

<class name=*"com.att.pmoss.cti.GcpSareaTmpData"* table=*"GCPSAREATMP"*>

<meta attribute=*"class-description"*>

This class contains the dns, portname, configuredPortSpeed and authorizedAllocationRatio

</meta>

<id name=*"ptnii"* column=*"PTNII"* type=*"string"*/>

<property name=*"portIdentifier"* column=*"PORTIDENTIFIER"* type=*"string"*/>

<property name=*"configuredPortSpeed"* column=*"CONFIGUREDPORTSPEED"* type=*"string"*/>

<property name=*"authorizedAllocationRatio"* column=*"AUTHORIZEDALLOCATIONRATIO"* type=*"string"*/>

<property name=*"reservationInEffectIndicator"* column=*"RESERVATIONINEFFECTINDICATOR"* type=*"string"*/>

<property name=*"weightFactor"* column=*"WEIGHTFACTOR"* type=*"string"*/>

<property name=*"portLock"* column=*"PORTLOCK"* type=*"string"*/>

<property name=*"protectionMode"* column=*"PROTECTIONMODE"* type=*"string"*/>

<property name=*"authorizedDynamicBw"* column=*"AUTHORIZEDDYNAMICBW"* type=*"string"*/>

<property name=*"espEnforcedMaxEvcCount"* column=*"ESPENFORCEDMAXEVCCOUNT"* type=*"string"*/>

</class>

<class name=*"com.att.pmoss.cti.GcpSareaCalculatedData"* table=*"OVERBOOKINGINTERFACES"*>

<meta attribute=*"class-description"*>

This class contains the dns, portname ,configuredPortSpeed, authorizedAllocationRatio,adjustedSpeed and speed

</meta>

<id name=*"ptnii"* column=*"DNS"* type=*"string"*/>

<property name=*"portIdentifier"* column=*"PORTIDENTIFIER"* type=*"string"*/>

<property name=*"configuredPortSpeed"* column=*"CONFIGUREDPORTSPEED"* type=*"string"*/>

<property name=*"authorizedAllocationRatio"* column=*"AUTHORIZEDALLOCATIONRATIO"* type=*"string"*/>

<property name=*"adjustedSpeed"* column=*"ADJUSTEDSPEED"* type=*"string"*/>

<property name=*"speed"* column=*"SPEED"* type=*"string"*/>

<property name=*"interfaceName"* column=*"INTERFACENAME"* type=*"string"*/>

<property name=*"reservationInEffectIndicator"* column=*"RESERVATIONINEFFECTINDICATOR"* type=*"string"*/>

<property name=*"weightFactor"* column=*"WEIGHTFACTOR"* type=*"string"*/>

<property name=*"portLock"* column=*"PORTLOCK"* type=*"string"*/>

<property name=*"protectionMode"* column=*"PROTECTIONMODE"* type=*"string"*/>

<property name=*"authorizedDynamicBw"* column=*"AUTHORIZEDDYNAMICBW"* type=*"string"*/>

<property name=*"espEnforcedMaxEvcCount"* column=*"ESPENFORCEDMAXEVCCOUNT"* type=*"string"*/>

</class>

</hibernate-mapping>

1. Using the Hibernate API

* Create a session factory.
* Create a session from the Session factory.
* Use the session to save model object.

Create a session factory.

Configuration configuration=**null**;

StandardServiceRegistry serviceRegistry=**null**;

configuration = **new** Configuration().configure();

configuration.setProperty("hibernate.connection.url",System.getenv("JDBC\_URL"));

configuration.setProperty("hibernate.connection.username",System.getenv("USERNAME"));

configuration.setProperty("hibernate.connection.password",System.getenv("PASSWORD"));

serviceRegistry = **new** StandardServiceRegistryBuilder().applySettings(configuration.getProperties()).build();

factory = configuration.buildSessionFactory(serviceRegistry);



Create a session from the Session factory/ Use the session to save model object.

**public** **static** **void** addGcpSareaCalculatedDataToTable(SessionFactory factory, GcpSareaCalculatedData caldata){

// Create a session from the Session factory

Session session = factory.getCurrentSession();

Transaction tx = **null**;

**try**{

tx = session.beginTransaction();

// Use the session to save model object.

session.save(caldata);

tx.commit();

}**catch** (HibernateException e) {

logger.info("Hibernate Exception while begining the transaction in method addGcpSareaCalculatedDataToTable" + e);

**if** (tx!=**null**)

tx.rollback();

e.printStackTrace();

}

}

To get data from DB

session.save(caldata);

session.persist(caldata);

session.get(Caldata.class, 1);

session.load(Caldata.class, 1); // will first check cache and the create proxy object and then checks the DB

session.update(caldata);

session.delete(caldata);

POJO Annotation: Doing with Annotation approach.

@Entity(name=”change pojo name”)

@Table(name=”Mapping the with DB table name”)

@Id // make field as primary key

@Column(name=”column\_name”)

@Basic

@Transient //to filed which ignore to save in DB

@Temporal(TemporalType.DATE) // save date in formatted way

@Lob //to specify column datatype (large object)

@JsonIgnore //to ignore for response json

Primary Key:

@Id

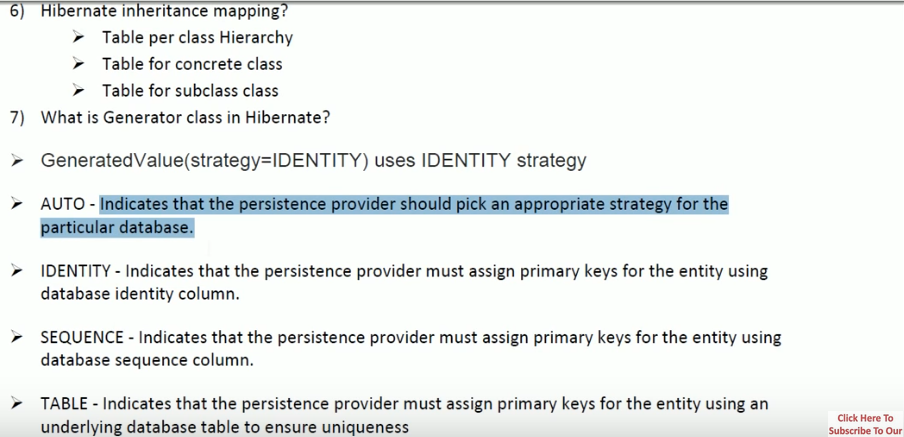
@GeneratedValue(strategy=GenerationType.AUTO) // to generate automatically and used as surrogate primary key.

Types of Strategy

GenerationType.AUTO

GenerationType.SEQUENCE

GenerationType.TABLE



Value object / Entity Object:

Is object which is not having mean by itself. Ex Address:

@Embedded

@AttributeOverrides({ // used for having same table reuse.

@AttributeOverride(name="street", column=@Column(name="office\_street")),

@AttributeOverride(name="city", column=@Column(name="office\_city")),

@AttributeOverride(name="state", column=@Column(name="office\_state"))

})

private Address officeAddress;

@ElementCollection // to store collection in DB as separate table.

**private** Set<Address> addressHistory;

@ElementCollection(fetch=FetchType.***EAGER***) // collection can be used after session closed

//@GenericGenerator(name="hilo-gen",strategy="hilo") // not working-- compile time error

//@GenericGenerator(name="sequence-gen",strategy="sequence") // not working -- runtime error

@GenericGenerator(name="increment-gen",strategy="increment")

@JoinTable(name="addressHistory", joinColumns=@JoinColumn(name="user\_ids")) // to give name to table explicitly

@CollectionId(columns= {@Column(name="add\_id")},generator="increment-gen", type=@Type(type="long")) // to create unique column

**private** Collection<Address> addressHistory;

* @PrimaryKeyJoinColumn should be used for associated entities sharing the same primary key.

@OneToOne(cascade = CascadeType.MERGE)

@PrimaryKeyJoinColumn

private EmployeeDetail employeeDetail;

* @JoinColumn & @OneToOne should be mappedBy attribute when foreign key is held by one of the entities.

@JoinColumn  
@JoinColumn annotation is used for one-to-one or many-to-one associations when foreign key is held by one of the column of entities. Not used for one-to-many association

@ManyToOne

@JoinColumn(name = "statusId")

private EmployeeStatus status;

@Entity

@Table(name = "employee")

public class Employee implements Serializable {

@OneToMany(mappedBy = "employee", fetch = FetchType.EAGER)

@OrderBy("firstName asc")

private Set communications;

}

@JoinTable: @JoinTable and mappedBy should be used for entities linked through an association table.

Table STUDENT and table MAIL is assosatied with third table STIDENT\_MAIL

@OneToMany(cascade = {CascadeType.***ALL***})

//@OneToMany(cascade = {CascadeType.ALL},mappedBy="student")

@JoinTable(name="STUDENT\_MAIL",joinColumns=@JoinColumn(name="STUDENT\_ID"),

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

Collection<Email> email = **new** ArrayList<>();

@MapsId: Two entities with shared key can be persisted using @MapsId annotation.

@OneToOne

@MapsId

@JoinColumn(name = "communicationId")

private Communication communication;

@ManyToMany(cascade= {CascadeType.***ALL***})

//@JoinColumn(name="id", referencedColumnName="id", insertable= false , updatable=false)

Collection<Sports> sportList = **new** ArrayList<>();

### **Hibernate Annotations for inheritance mapping**

Hibernate supports the three basic inheritance mapping strategies:

* table per class hierarchy- single table per Class Hierarchy Strategy. One table Car is created with cartype column is the discriminator.

@DiscriminatorColumn: As the name suggests this column is the discriminator and this annotation specifies the discriminator column for the SINGLE\_TABLE and JOINED Inheritance mapping strategies.

@Entity

@Inheritance(strategy=InheritanceType.SINGLE\_TABLE)

@DiscriminatorColumn(name="cartype",discriminatorType=DiscriminatorType.STRING )

@DiscriminatorValue("Car")

public class Car { }

@Entity

@DiscriminatorValue("BMW")

public class BMW extends Car { }

* table per subclass - joined subclass Strategy. Two table ship and Titanic are created with

@Entity

@Inheritance(strategy=InheritanceType.JOINED)

public class Ship implements Serializable {}

@Entity

@PrimaryKeyJoinColumn

public class Titanic extends Ship {}

* table per concrete class

@Entity

@Inheritance(strategy = InheritanceType.TABLE\_PER\_CLASS)

public class Aeroplane implements Serializable {}

IMP

Fetching type:

Lazy(default) // it will return first level of attribute, gives next level if it is called.

Eager // if we want to access the collection data after closing the session is called eager fetch

@Entity

@Table(name = "employee")

public class Employee implements Serializable {

@OneToMany(mappedBy = "employee", fetch = FetchType.EAGER)

@OrderBy("firstName asc")

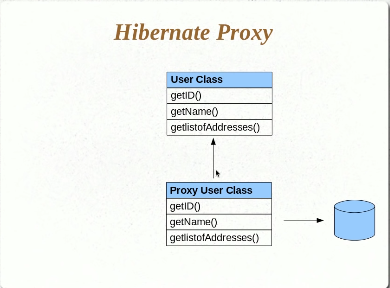
private Set communications;

}

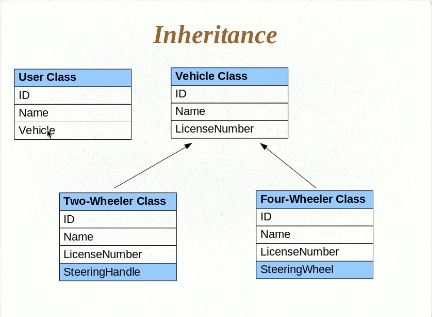
# Hibernate Session get() vs load() difference with examples

Based on the above explanations we have following differences between **get() vs load()**:

1. get() loads the data as soon as it’s called whereas load() returns a proxy object and loads data only when it’s actually required, so load() is better because it support lazy loading.
2. Since load() throws exception when data is not found, we should use it only when we know data exists.
3. We should use get() when we want to make sure data exists in the database.







**Transient, Persistent and Detached Objects**

UserDetails user = new UserDetails() // this the Transient object before hand overring to hibernate

User.setName(“Sachin”);

Session.save(user) // now user object is persistent object of hibernate. Any change to state of the object will go as update to table.

User.setName(“Sachin again”);

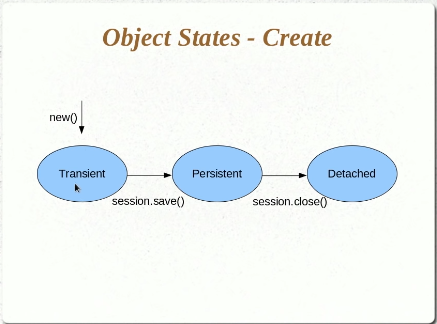
User.setName(“Sachin again again”); // this stored in DB as update query.

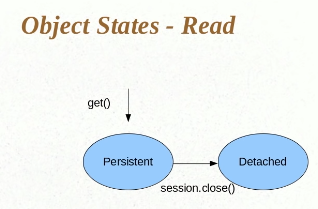
Session.close() // now user object is detached object any change to object state will not be related in DB.

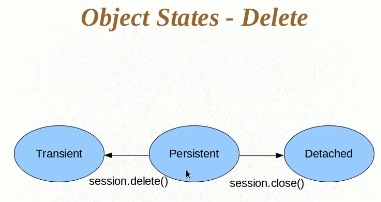
User.setName(“Sachin again again again”); // not stored in DB because object is detached from hibernate.

# Understanding State Changes

# 







# Persisting Detached Objects

# 

# Introducing HQL and the Query Object

HQL: Hibernate Query Language.

* Similar to SQL but we use class name instead of table name.
* No need of select \*
* Where clause uses class variable for the applying condition.

**public** **static** List<Object[]> readingDataFromTmpTable(SessionFactory factory){

Session session = factory.getCurrentSession();

Transaction tx = **null**;

List<Object[]> gcpTmpDatalist = **null**;

**try**{

tx = session.beginTransaction();

org.hibernate.Query query = session.createSQLQuery("FROM GCPSAREATMP");

gcpTmpDatalist = query.list();

tx.commit();

}**catch** (HibernateException e) {

logger.info("Hibernate Exception while begining the transaction in method readingDataFromTmpTable" + e);

**if** (tx!=**null**)

tx.rollback();

e.printStackTrace();

}

**return** gcpTmpDatalist;

}

import org.springframework.data.jpa.repository.Query;

import org.springframework.data.repository.CrudRepository;

import org.springframework.stereotype.Repository;

@Repository

public interface CityRepository extends CrudRepository<City, Long> {

@Query("select c from City c where c.name like %?1")

List<City> findByNameEndsWith(String chars);

}

# Select and Pagination in HQL

Pagination allows us to return just a subset of a whole result in a Page. This is useful, for example, when navigating through several pages of data on a web page.

select \* from students limit 5 offset 5;

org.hibernate.Query query = session.createSQLQuery(" Select \* FROM GCPSAREATMP");

query.setFirstResult(5); //offset

query.setMaxResults(5); //limit

@Query(value = "SELECT u FROM User u ORDER BY id")

Page<User> findAllUsersWithPagination(Pageable pageable);

# Understanding Parameter Binding and SQL Injection

There are two possible ways that we can pass method parameters to our query.

Concatenating the query make SQL injection.

tx = session.beginTransaction();

org.hibernate.Query query = session.createSQLQuery("select a.dns, regexp\_replace(ifname, '.\*:','') ifname, ifspeed, regexp\_replace(b.DESCR, '.\* ','') DESCR, a.VENDOR, c.IFINDEX from ctidevices a,ctiinterfaces b , ctiinterfaceindexes c where a.devkey = b.devkey and b.intkey = c.intkey and a.validto is null and a.netid='SAR'and a.dns = :dns");

query.setParameter("dns",dns);

gcpTmpDatalist = query.list();

tx.commit();

Other ways:

query.setInteger(position, val)

query.setInteger(name, val)

**pass method parameters to the query in the same order they appear in the method declaration:**

@Query("SELECT u FROM User u WHERE u.status = ?1")

User findUserByStatus(Integer status);

@Query("SELECT u FROM User u WHERE u.status = ?1 and u.name = ?2")

User findUserByStatusAndName(Integer status, String name);

**pass method parameters to the query using named parameters:**

@Query("SELECT u FROM User u WHERE u.status = :status and u.name = :name")

User findUserByUserStatusAndUserName(@Param("status") Integer userStatus,

  @Param("name") String userName);

## **Collection Parameter**

the case when the where clause of our JPQL or SQL query contains the IN (or NOT IN) keyword:

In this case we can define a query method which takes *Collection*as a parameter:

|  |  |
| --- | --- |
|  | @Query(value = "SELECT u FROM User u WHERE u.name IN :names")  List<User> findUserByNameList(@Param("names") List<String> names); |

## **Define Order in a Query**

We can pass an additional parameter of type Sort to a Spring Data method declaration that has the @Query annotation. It'll be translated into the ORDER BY clause that gets passed to the database.

### **3.1. Sorting for JPA Provided and Derived Methods**

For the methods we get out-of-the-box like findAll(Sort) or the ones that are generated by parsing method signatures, **we can only use object properties to define our sort**:

|  |  |
| --- | --- |
| 1 | userRepository.findAll(new Sort(Sort.Direction.ASC, "name")); |

**Now imagine that we want to sort by the length of a name property:**

userRepository.findAllUsers(JpaSort.unsafe("LENGTH(name)"));

@Query(value = "SELECT u FROM User u")

List<User> findAllUsers(Sort sort);

**When the @Query annotation uses native SQL, then it's not possible to define a Sort.**

## **Update Queries With** @Modifying

**@Query annotation to modify the state of the database by also adding the @Modifying annotation**

@Modifying

@Query("update User u set u.status = :status where u.name = :name")

int updateUserSetStatusForName(@Param("status") Integer status,

  @Param("name") String name);

@Modifying

@Query(

  value =

    "insert into Users (name, age, email, status) values (:name, :age, :email, :status)",

  nativeQuery = true)

void insertUser(@Param("name") String name, @Param("age") Integer age,

  @Param("status") Integer status, @Param("email") String email);

## **Dynamic Query**

where we need to select all the users whose email is LIKE one from a set defined at runtime — email1, email2, …, emailn:

Instead, by implementing a custom composite repository, we can extend the base JpaRepository functionality and provide our own logic for building a dynamic query. Let's take a look at how to do this.

**Spring provides a way for extending the base repository through the use of custom fragment interfaces.**

We'll start by creating a custom fragment interface:

public interface UserRepositoryCustom {

    List<User> findUserByEmails(Set<String> emails);

}

And then, we'll implement it:

public class UserRepositoryCustomImpl implements UserRepositoryCustom {

@PersistenceContext

private EntityManager entityManager;

@Override

public List<User> findUserByEmails(Set<String> emails) {

CriteriaBuilder cb = entityManager.getCriteriaBuilder();

CriteriaQuery<User> query = cb.createQuery(User.class);

Root<User> user = query.from(User.class);

Path<String> emailPath = user.get("email");

List<Predicate> predicates = new ArrayList<>();

for (String email : emails) {

predicates.add(cb.like(emailPath, email));

}

query.select(user)

.where(cb.or(predicates.toArray(new Predicate[predicates.size()])));

return entityManager.createQuery(query)

.getResultList();

}

}

we'll integrate our fragment by extending the new interface in the UserRepository:

public interface UserRepository extends JpaRepository<User, Integer>, UserRepositoryCustom {

    //  query methods from section 2 - section 7

}

# Named Queries

@NamedQuery annotation is a predefined query that we create and associate with a container-managed entity. @Query annotation is a similar annotation, which declares finder queries directly on repository methods. While @NamedQuery is used on domain classes, Spring Data JPA @Query annotation is used on Repository interface. This frees the domain classes from persistence specific information, which is a good thing.

@Entity

@NamedQuery(name="GcpSareaTmpData.byPtnii",query="from GcpSareaTmpData where ptnii=?")

@NamedNativeQuery(name="GcpSareaTmpData.byPortIdentifier", query="select \* from GcpSareaTmpData where PortIdentifier= ?", resultClass=GcpSareaTmpData.**class**)

**public** **class** GcpSareaTmpData {

**private** String ptnii;

}

tx = session.beginTransaction();

org.hibernate.Query query session.getNamedQuery("GcpSareaTmpData.byPtnii");

gcpTmpDatalist = query.list();

tx.commit();

@Entity

@Table(name = "cities")

@NamedQuery(name = "City.findAllOrderedByNameDescending",

query = "SELECT c FROM City c ORDER BY c.name DESC")

public class City { }

@Repository

public interface CityRepository extends CrudRepository<City, Long> {

List<City> findAllOrderedByNameDescending();

}

### **Native**

We can use also native SQL to define our query. All we have to do is to set the value of the nativeQuery attribute to true and define the native SQL query in the value attribute of the annotation:

@Query(

  value = "SELECT \* FROM USERS u WHERE u.status = 1",

  nativeQuery = true)

Collection<User> findAllActiveUsersNative();

**Following is named native query in spring Boot.**

**public** **interface** StudentRepository **extends** CrudRepository<Student, Integer>{

@Query(value="select st.\* from students st where st.id=1", nativeQuery = **true**)

Iterable<Student> queryBy();

@Query(value="select s.name as name , st.name as sport from students s\r\n" +

"left join student\_sprots ss on s.id = ss.studentid\r\n" +

"left join sports st on ss.sports=st.id", nativeQuery = **true**)

Iterable<StudentJoinDetails> queryByJoin();

}

# Introduction to Criteria API

Pull data from DB have following way.

1. Making class as entity and pulling data using that.(basic way and less controller)
2. Hibernate query language have more controller can specify where clause.(if the query is big then its difficult to maintain)
3. Third way is this criteria API.

Criteria criteria = session.createCriteria(GcpSareaTmpData.**class**);

criteria.add(Restrictions.*or*(Restrictions.*eq*("ptnii", "3"), Restrictions.*between*("ptnii", 1, 10)))

.add(Restrictions.*ge*("ptnii", 10));

criteria.list();

we can chain the restriction.

**Criteria API offers a programmatic way to create typed queries**,

we saw three ways to use criteria queries in our Spring application:

* creating a DAO class is the most straightforward and most flexible way
* extending a *@Repository* interface to seamless integration with automatic queries
* using predicates in *Specification* instances to make the simple cases cleaner and less verbose

@Entity

class Book {

    @Id

    Long id;

    String title;

    String author;

    // getters and setters

}

@Override

List<Book> findBooksByAuthorNameAndTitle(String authorName, String title) {

    CriteriaBuilder cb = em.getCriteriaBuilder();

    CriteriaQuery<Book> cq = cb.createQuery(Book.class);

    Root<Book> book = cq.from(Book.class);

    List<Predicate> predicates = new ArrayList<>();

    if (authorName != null) {

        predicates.add(cb.equal(book.get("author"), authorName));

    }

    if (title != null) {

        predicates.add(cb.like(book.get("title"), "%" + title + "%"));

    }

    cq.where(predicates.toArray(new Predicate[0]));

    return em.createQuery(cq).getResultList();

}

The code above follows a standard Criteria API workflow:

* First, we get a *CriteriaBuilder* reference, which we can use to create different parts of the query
* Using the *CriteriaBuilder*, we create a *CriteriaQuery<Book>*, which describes what we want to do in the query. Also, it declares the type of a row in the result
* With *CriteriaQuery<Book>* we declare the starting point of the query (*Book* entity), and we store it in the *book* variable for later use
* Next, with *CriteriaBuilder* we create predicates against our *Book* entity. Note, that these predicates don't have any effect yet
* We apply both predicates to our *CriteriaQuery.* *CriteriaQuery.where(Predicate…)* combines its arguments in a logical *and*. This is the point when we tie these predicates to the query
* After that, we create a *TypedQuery<Book>* instance from our *CriteriaQuery*
* Finally, we return all matching *Book* entities

# Projections and Query By Example

Using Projections we can do aggregation and select particular parameters

Criteria criteria = session.createCriteria(GcpSareaTmpData.**class**)

.setProjection(Projections.*count*(dns))

.setProjection(Projections.*max*(dns));

Query with Example:

create object and set value for object based on the set value it query and added example as criteria.

GcpSareaTmpData gcp = **new** GcpSareaTmpData();

gcp.setPtnii("12");

gcp.setPortIdentifier("SAREA");

Example example = Example.create(gcp).enableLike().ignoreCase().excludeProperty("Ptnii"); Criteria criteria = session.createCriteria(GcpSareaTmpData.**class**)

.add(example);

## **Interface-Based Projections**

Looking back at the Address class, we can see **it has many properties, yet not all of them are helpful.**

public interface AddressView {

    String getZipCode();

}

public interface AddressRepository extends Repository<Address, Long> {

    List<AddressView> getAddressByState(String state);

}

public interface PersonView {

    String getFirstName();

    String getLastName();

}

|  |
| --- |
| public interface AddressView {      // ...      PersonView getPerson();  } |

**Notice the method that returns the nested projection must have the same name as the method in the root class that returns the related entity.**

another sort of interface-based projections: open projections. **These projections enable us to define interface methods with unmatched names and with return values computed at runtime.**

public interface PersonView {

    // ...

    @Value("#{target.firstName + ' ' + target.lastName}")

    String getFullName();

}

public interface PersonRepository extends Repository<Person, Long> {

    PersonView findByLastName(String lastName);

}

## **Class-Based Projections**

Instead of using proxies Spring Data creates for us from projection interfaces, **we can define our own projection classes.**

public class PersonDto {

    private String firstName;

    private String lastName;

    public PersonDto(String firstName, String lastName) {

        this.firstName = firstName;

        this.lastName = lastName;

    }

    // getters, equals and hashCode

}

**For a projection class to work in tandem with a repository interface, the parameter names of its constructor must match properties of the root entity class.**

public interface PersonRepository extends Repository<Person, Long> {

    // ...

    PersonDto findByFirstName(String firstName);

}

## **Dynamic Projections**

**We can apply dynamic projections just by declaring a repository method with a Class parameter:**

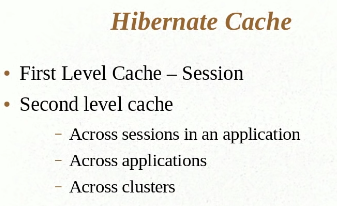
public interface PersonRepository extends Repository<Person, Long> {

    // ...

    <T> T findByLastName(String lastName, Class<T> type);

}

# Caching in Hibernate



First Level cache: hibernate have inbuild caching when it in the session.

If we make two time get, hibernate make only one DB query. When session is closed then cache is also closed.

As most other fully-equipped ORM frameworks, Hibernate has the concept of first-level cache. It is a session scoped cache which ensures that each entity instance is loaded only once in the persistent context.

Once the session is closed, first-level cache is terminated as well. This is actually desirable, as it allows for concurrent sessions to work with entity instances in isolation from each other.

# Configuring Second Level Cache

second-level cache is *SessionFactory*-scoped, meaning it is shared by all sessions created with the same session factory. When an entity instance is looked up by its id (either by application logic or by Hibernate internally, *e.g.* when it loads associations to that entity from other entities), and if second-level caching is enabled for that entity, the following happens:

* If an instance is already present in the first-level cache, it is returned from there
* If an instance is not found in the first-level cache, and the corresponding instance state is cached in the second-level cache, then the data is fetched from there and an instance is assembled and returned
* Otherwise, the necessary data are loaded from the database and an instance is assembled and returned

## **Region Factory**

Hibernate second-level caching is designed to be unaware of the actual cache provider used. Hibernate only needs to be provided with an implementation of the org.hibernate.cache.spi.RegionFactory interface which encapsulates all details specific to actual cache providers. Basically, it acts as a bridge between Hibernate and cache providers.

**we use Ehcache as a cache provider**

**Classes** must implement **Serializable** if you **want** to **serialize** them. ... **Serializable** is simply **required** for transfering data via IIOP or JRMP(Java Remote Method Protocol) (RMI) between JVM-instances. In case of a pure web application the domain objects are sometimes stored in HTTPSession for **caching** / optimization purposes.

