01b: ♦ 15+ Hibernate basics Q8 – Q15 interview questions & answers

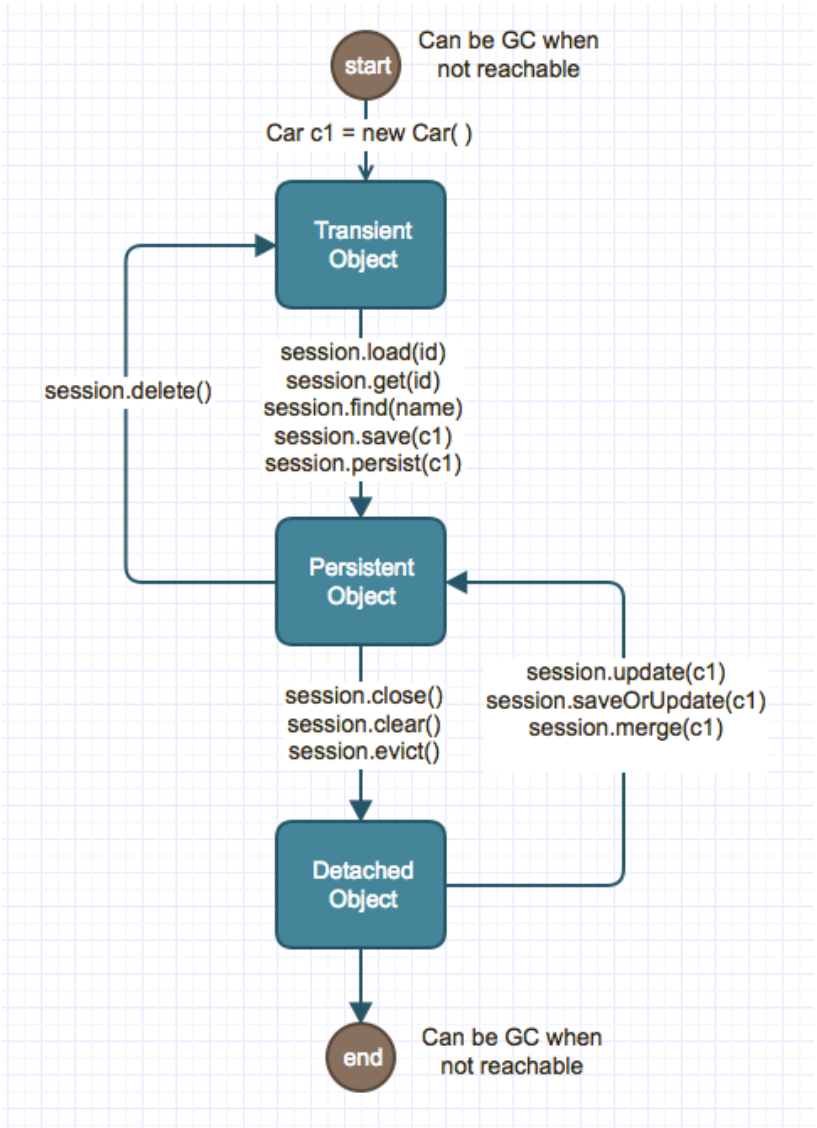
Posted on July 27, 2016 by Arulkumaran Kumaraswamipillai

Extends 15+ Hibernate basics Q1 – Q7 interview questions & answers.

Q9. Explain hibernate object states? Explain hibernate objects life cycle?
A9. There are 3 states.

1. Persistent objects and collections are short lived single threaded objects, which store the persistent state. These objects synchronize their state with the database depending on your flush strategy (i.e. auto-flush where as soon as setXXX() method is called or an item is removed from a Set, List, etc or define your own synchronization points with session.flush(), transaction.commit() calls). If you remove an item from a persistent collection like a Set, it will be removed from the database either immediately or when

flush() or commit() is called depending on your flush strategy. They are Plain Old Java Objects (POJOs) and are currently associated with a session. As soon as the associated session is closed, **persistent objects become detached objects** and are free to be used directly as data transfer objects in any application layers like business layer, presentation layer, etc.



Hibernate Objects Life Cycle

Note: In JPA 2.0, you use an **EntityManager** instead of a Session. So, you will use entityManager.persist(entity) to create(..), entityManager.merge(entity) to edit(..), entityManager.remove(entity) to remove(..), and entityManager.find(entityClass, primaryKey) to find(..).

16 Technical Key Areas

80+ step by step Java Tutorials

2. Detached objects and collections are instances of persistent objects that were associated with a session but currently not associated with a session. These objects can be freely used as Data Transfer Objects without having any impact on your database. Detached objects can be later on attached to another session by calling methods like session.update(), session.saveOrUpdate() etc. and become persistent objects.

3. Transient objects and collections are instances of persistent objects that were never associated with a session. These objects can be freely used as Data Transfer Objects without having any impact on your database. Transient objects become persistent objects when associated to a session by calling methods like session.save( ), session.persist( ) etc.

Note: The states of transient and detached objects cannot be synchronized with the database as they are not managed by Hibernate.

Q10. What are the benefits of detached objects?
A10.

Pros: When long transactions are required due to user think-time, it is the best practice to break the long transaction up into two or more transactions. You can use detached objects from the first transaction to carry data all the way up to the presentation layer. These detached objects get modified outside a transaction and later on re-attached to a new transaction via another session.

Cons:

– In general, working with detached objects is quite cumbersome, and it is better not to clutter up the session with them if possible. It is better to discard them and re-fetch them on subsequent requests. This approach is not only more portable but also more efficient because the objects hang around in Hibernate's cache anyway.

Akka Tutorial (9)
Core Java Tutorials (2
Hadoop & Spark Tuto
JEE Tutorials (19)
Scala Tutorials (1)
Spring & HIbernate Tu
Tools Tutorials (19)
Other Tutorials (45)

100+ Java pre-interview coding tests

open all | close all
Can you write code? (
♦ Complete the given
Converting from A to I
Designing your classe
Java Data Structures
Passing the unit tests
What is wrong with th
Writing Code Home A
Written Test Core Jav
Written Test JEE (1)

How good are your .....?

open all | close all
Career Making Know-
Job Hunting & Resum

– Also from pure rich domain driven design perspective, it is recommended to use DTOs (DataTransferObjects) and DOs (DomainObjects) to maintain the separation between Service and UI tiers.

Q11. How does Hibernate distinguish between transient (i.e. newly instantiated) and detached objects?
A11.

1. Hibernate uses the "version" property, if there is one.

2. No identifier value means a new object. This does work only for Hibernate managed surrogate keys. Does not work for natural keys and assigned (i.e. not managed by Hibernate) surrogate keys.

3. Write your own strategy with "Interceptor.isUnsaved( )".

Note: When you reattach detached objects, you need to make sure that the dependent objects are reattached as well.

Q12. How does hibernate support lazy loading?
A12. Hibernate uses a **proxy object to support lazy loading**. Basically as soon as you reference a child or lookup object via the accessor/getter methods, if the linked entity is not in the session cache (i.e. the first-level cache), then the proxy code will go off to the database and load the linked object. It uses javassist (or CGLIB) to effectively and dynamically generate sub-classed implementations of your objects.

Let's look at an example. An employee hierarchy table can be represented as Java object hierarchy as shown below:

```
1  public class Employee {
2
3      private Long id;
4      private String name;
5      private String title;
6      private Employee superior;          //parent
7      private Set<Employee> subordinates; //childre
8
9      //getters and setters are omitted
```

```
10
11 }
12
```

In the above example, if you use lazy loading then the "superior" and "subordinates" will be proxied (i.e. not the actual object, but the stub object that knows how to load the actual object) when the main "Employee" object is loaded. So, if you need to get the "subordinates" or "superior" object, you invoke the getter method on the employee like employee.getSuperior( ) and the actual object will be loaded.

Hibernate does require the same EntityManager to be available in order to lazily load objects. If you have no EntityManager, then you have no knowledge of the datastore. Once the transaction is committed the objects become detached, and you can't lazy load detached objects. So, you need to **lazily load your objects within the same transaction** in your service layer.

Q13. What do you understand by automatic dirty checking in Hibernate?
A13. Dirty checking is a feature of hibernate that saves time and effort to update the database when states of objects are modified inside a transaction. All persistent objects are monitored by hibernate. It detects which objects have been modified and then calls update statements on all updated objects.

Hibernate Session contains a PersistenceContext object that maintains a cache of all the objects read from the database as a Map. So, when you modify an object within the same session, Hibernate compares the objects and triggers the updates when the session is flushed. The objects that are in the PersistenceContext are pesistent objects.

Q14. What do you understand by the terms optimistic locking versus pessimistic locking?
A14. **Optimistic locking** means a specific record in the database table is open for all users/sessions. Optimistic locking uses a strategy where you read a record, make a note of the version number and check that the version number

hasn't changed before you write the record back. When you write the record back, you filter the update on the version to make sure that it hasn't been updated between when you check the version and write the record to the disk. If the record is dirty (i.e. different version to yours) you abort the transaction and the user can re-start it.

You could also use other strategies like checking the timestamp or all the modified fields (this is useful for legacy tables that don't have version number or timestamp column). Note: The strategy to compare version numbers and timestamp will work well with detached hibernate objects as well. Hibernate will automatically manage the version numbers.

In Hibernate, you can use either long number or Date for versioning

```
1 @Version
2 private long id;
3
```

or

```
1 @Version
2 private Date version;
3
```

Pessimistic locking means a specific record in the database table is open for read/write only for that current session. The other session users can not edit the same because you lock the record for your exclusive use until you have finished with it. It has much better integrity than optimistic locking, but requires you to be careful with your application design to avoid deadlocks. In pessimistic locking, appropriate transaction isolation levels need to be set, so that the records can be locked at different levels. The general isolation levels are

— Read uncommitted isolation
— Read committed isolation

— Repeatable read isolation
— Serializable isolation

It can be dangerous to use “read uncommitted isolation” as it uses one transaction’s uncommitted changes in a different transaction. The “Serializable isolation” is used to protect phantom reads, phantom reads are not usually problematic, and this isolation level tends to scale very poorly. So, if you are using pessimistic locking, then read committed and repeatable reads are the most common ones.

Q15. What is First and Second Level caching in Hibernate?
A15 **First-level cache** is always associated with the Session object. Hibernate uses this cache by default. Though we can not disable the first level cache in hibernate, we can remove some of objects from it when needed with session.evict() to remove a particular object and session.clear() to remove the whole cache.

Second-level cache is always associated with the Session Factory object. The second-level cache is called ‘second-level’ because there is already a cache operating for you in Hibernate for the duration you have a session open. While running the transactions, in between it loads the objects at the Session Factory level, so that those objects will be available to the entire application, not bound to single user. The ‘second-level’ cache exists as long as the session factory is alive.

Q16. What are the general steps involved in creating Hibernate related artifacts?
A16.The general steps involved in creating Hibernate related artifacts involve the following steps:

Step #1. Define the domain (aka entity) objects like Employee, Address, etc to represent relevant tables in the underlying database with the appropriate annotations or using the *.hbm.xml mapping files.

Step #2. Define the Repository (aka DAO — Data Access Objects) interfaces and implementations classes that use the

domain objects and the hibernate session to perform data base CRUD (Create, Read, Update and Delete) operations the hibernate way.

Step #3. Define the service interfaces and the classes that make use of one or more repositories (aka DAOs) in a transactional context.A transaction manager will be used to coordinate transactions (i.e. commit or rollback) between a number of repositories.

Step #4. Finally, use an IoC container like Spring framework to wire up the Hibernate classes like SessionFactory, Session, transaction manager, etc and the user defined repositories, and the service classes. A number of interceptors can be wired up as well for deadlock retry, logging, auditing, etc using Spring.

Step #5. Favor using JPA and CrudRepository from Spring.

Q17. How would you define a hibernate domain object with table mappings, native named queries, and custom data conversion using annotations?
A17. Firstly, define a parent domain object class for any common method implementations.

```
1  public class MyAppDomainObject {
2
3      //for example
4      protected boolean isPropertyEqual(Object com
5          if (comparee == null) {
6              if (compareToo != null) {
7                  return false;
8              }
9          } else if (!comparee.equals(compareToo))
10             return false;
11         }
12         return true;
13     }
14 }
15
```

The entity class.

```
1  @Entity
2  @org.hibernate.annotations.Entity(selectBeforeUp
```

```
@Table(name = "tbl_employee")
@TypeDefs(value = { @TypeDef(name = "dec", typeC

@NamedNativeQueries({
    @NamedNativeQuery(name = "HighSalary", query
 @NamedNativeQuery(name = "LowSalary", query = "
})

public class Employee extends MyAppDomainObject

    @Id
    @GeneratedValue(strategy = GenerationType.AU
    @Column(name = "employee_id")
    private Long id;

    @Column(name = "emp_code")
    private String accountCode;

    @Column(name = "manager_code")
    private String adviserCode;

    @Column(name = "type")
    @Enumerated(EnumType.STRING)
    private EmployeeType type = EmployeeType.PER

    @Type(type = "dec")
    @Column(name = "base_salary")
    private Decimal salary = Decimal.ZERO;

    @Transient
    private Decimal salaryWithBonus; //not persi

    @Formula("base_salary*2")
    private Decimal doubleSalary;   //derived or

    @Formula("(select base_salary where type = '
    private Decimal permanantLeaveLoading;   //d

    @OneToOne(cascade = { CascadeType.REFRESH })
    @JoinColumn(name = "emp_code", insertable =
    private EmployeeExtrInfo extraInfo;

    @ManyToOne(cascade = { CascadeType.REFRESH }
    @JoinColumn(name = "manager_code", insertabl
    private Manager manager;

    @OneToMany(cascade = { ALL, MERGE, PERSIST,
    @JoinColumn(name = "emp_code", nullable = fa
    @Cascade({ org.hibernate.annotations.Cascade
    private List<PaymentDetail> paymentDetails =

    //getters and setters omitted for brevity
    //equals and hashcode methods
}

```

Hibernate repository class that makes use of the Employee domain object. Firstly define the interface.

```
import java.util.List;

public interface EmployeeTableRepository {
```

```
4    Employee saveEmployee(Employee employee) throws
5    Employee loadEmployee(Long employeeId) throws R
6    List<Employee> findAllEmployeesWithHighSalary(B
7    List<Employee> findAllEmployeesWithLowSalary(Bi
8 }
9
```

Define the implementation

```
1  @SuppressWarnings("unchecked")
2  public class EmployeeTableHibernateRepository ex
3
4      public EmployeeTableHibernateRepository (Hib
5          setHibernateTemplate(hibernateTemplate);
6      }
7
8      //The employee objects gets constructed and
9      public Employee saveEmployee(Employee employ
10         Session session = getHibernateTemplate()
11         session.saveOrUpdate(employee);
12         session.flush();
13         session.evict(employee);
14         return this.loadEmployee(employee.getId(
15     }
16
17     public Employee loadEmployee(Long employeeId
18         Session session = getHibernateTemplate()
19         Criteria crit = session.createCriteria(E
20         crit.add(Restrictions.eq("id",employeeId
21         List<Employee> employees = crit.list();
22         if (employees.size() == 1) {
23             return employees.get(0);
24         }
25
26         //this is a custom exception class
27         throw new RepositoryException("Found mor
28     }
29
30
31     public List<Employee> findAllEmployeesWithHi
32         Session session = getHibernateTemplate()
33         Query query = session.getNamedQuery("Hig
34         query.setBigDecimal(":median_salary", me
35         return (List<Employee>) query.list();
36     }
37
38     public List<Employee> findAllEmployeesWithLo
39         Session session = getHibernateTemplate()
40         Query query = session.getNamedQuery("Low
41         query.setBigDecimal(":median_salary", me
42         return (List<Employee>) query.list();
43     }
44
45    //other methods can be defined here
46 }
47
```

The service layer that uses the repository layer

```
import org.springframework.transaction.PlatformT
import org.springframework.transaction.Transacti
import org.springframework.transaction.Transacti
import org.springframework.transaction.support.D
//....other imports

public class EmployeeServiceImpl implements Empl

    private final EmployeeTableRepository employ
    private PlatformTransactionManager transacti

    public EmployeeServiceImpl (EmployeeTableRep
        this.employeeRepository = employeeReposi
        this.transactionManager = transactionMan
    }

    public Employee loadEmployee(Long employeeId
        return this.employeeRepository.loadEmplo
    }
}

```

More Hibernate Interview Questions & Answers

1. Hibernate mistakes

2. Hibernate cache First & second level interview questions and answers

3. Hibernate automatic dirty checking

4. Hibernate entities with auditable, soft delete & optimistic locking fields

5. Understanding Hibernate proxy objects

6. Hibernate custom data type

7. 8 JPA interview questions and answers

Popular Member Posts

♦ 11 Spring boot interview questions & answers

904 views

♦ Q11-Q23: Top 50+ Core on Java OOP Interview Questions & Answers

816 views

001A: ♦ 7+ Java integration styles & patterns interview questions & answers

427 views

18 Java scenarios based interview Questions and Answers

408 views

♦ 7 Java debugging interview questions & answers

323 views

01b: ♦ 13 Spring basics Q8 – Q13 interview questions & answers

311 views

01: ♦ 15 Ice breaker questions asked 90% of the time in Java job interviews with hints

303 views

♦ 10 ERD (Entity-Relationship Diagrams) Interview Questions and Answers

301 views

♦ Q24-Q36: Top 50+ Core on Java classes, interfaces and generics interview questions & answers

251 views

001B: ♦ Java architecture & design concepts interview questions & answers

209 views

Bio | Latest Posts

Arulkumaran Kumaraswamipillai

Mechanical Eng to freelance Java developer in 3 yrs. Contracting since 2003, and attended 150+ Java job interviews, and often got 4 - 7 job offers to choose from. It pays to prepare. So, published Java interview Q&A books via Amazon.com in 2005, and sold 35,000+ copies. Books are outdated and replaced with this subscription

based site.**945+** paid members. join my LinkedIn Group. **Reviews**

About Arulkumaran Kumaraswamipillai

Mechanical Eng to freelance Java developer in 3 yrs. Contracting since 2003, and attended 150+ Java job interviews, and often got 4 - 7 job offers to choose from. It pays to prepare. So, published Java interview Q&A books via Amazon.com in 2005, and sold 35,000+ copies. Books are outdated and replaced with this subscription based site.**945+** paid members. join my LinkedIn Group. **Reviews**

‹ 01b: ♦ 13 Spring basics Q8 – Q13 interview questions & answers

03: Q13 – Q18 Scala FP combinators interview questions & answers ›

Posted in Hibernate, Hibernate Job Interview Essentials, member-paid

Empowers you to open more doors, and fast-track

Technical Know Hows

☀ Java generics in no time ☀ Top 6 tips to transforming your thinking from OOP to FP ☀ How does a HashMap internally work? What is a hashing function?
☀ 10+ Java String class interview Q&As ☀ Java auto un/boxing benefits & caveats ☀ Top 11 slacknesses that can come back and bite you as an experienced Java developer or architect

Non-Technical Know Hows

☀ 6 Aspects that can motivate you to fast-track your career & go places ☀ Are you reinventing yourself as a Java developer? ☀ 8 tips to safeguard your Java career against offshoring ☀ My top 5 career mistakes

Prepare to succeed

☀ Turn readers of your Java CV go from “Blah blah” to “Wow”? ☀ How to prepare for Java job interviews? ☀ 16 Technical Key Areas ☀ How to choose from multiple Java job offers?

Select Category

© Disclaimer

 ↑