Q. Explain DI or IOC pattern.
A: Dependency injection (DI) is a programming design pattern and architectural model, sometimes also referred to as inversion of control or IOC, although technically speaking, dependency injection specifically refers to an implementation of a particular form of IOC. Dependency Injection describes the situation where one object uses a second object to provide a particular capacity. For example: being passed a database connection as an argument to the constructor instead of creating one internally. The term "Dependency injection" is a misnomer, since it is not a dependency that is injected; rather it is a provider of some capability or resource that is injected. There are three common forms of dependency injection: setter-, constructor- and interface-based injection. Dependency injection is a way to achieve loose coupling. Inversion of control (IOC) relates to the way in which an object obtains references to its dependencies. This is often done by a lookup method. The advantage of inversion of control is that it decouples objects from specific lookup mechanisms and implementations of the objects it depends on. As a result, more flexibility is obtained for production applications as well as for testing.

Q. What are the different IOC containers available?
A. Spring is an IOC container. Other IOC containers are HiveMind, Avalon, and PicoContainer.

Q. What are the different types of dependency injection? Explain with examples.
A: There are two types of dependency injection: setter injection and constructor injection.
Setter Injection: Normally in all the java beans, we will use setter and getter method to set and get the value of property as follows:

```
public class namebean {
        String      name;
        public void setName(String a) {
                name = a;
        }
        public String getName() {
                return name;
        }
}
```

We will create an instance of the bean 'namebean' (say bean1) and set property as bean1.setName("tom"); Here in setter injection, we will set the property 'name' in spring configuration file as shown below:

```
<bean id="bean1"   class="namebean">
        <property   name="name" >
                <value>tom</value>
        </property>
</bean>
```

The subelement <value> sets the 'name' property by calling the set method as setName("tom"); This process is called setter injection. To set properties that reference other beans <ref>, subelement of <property> is used as shown below,

```
<bean id="bean1"   class="bean1impl">
        <property name="game">
                <ref bean="bean2"/>
        </property>
</bean>
<bean id="bean2"   class="bean2impl" />
```

Constructor injection: For constructor injection, we use constructor with parameters as shown below,

```
public class namebean {
        String name;
```

```
        public namebean(String a) {
                name = a;
        }
}
```

We will set the property 'name' while creating an instance of the bean 'namebean' as namebean bean1 = new namebean("tom");

Here we use the <constructor-arg> element to set the property by constructor injection as

```
<bean id="bean1"  class="namebean">
        <constructor-arg>
                <value>My Bean Value</value>
        </constructor-arg>
</bean>
```

Q. What is spring? What are the various parts of spring framework? What are the different persistence frameworks which could be used with spring?
A. Spring is an open source framework created to address the complexity of enterprise application development. One of the chief advantages of the Spring framework is its layered architecture, which allows you to be selective about which of its components you use while also providing a cohesive framework for J2EE application development. The Spring modules are built on top of the core container, which defines how beans are created, configured, and managed, as shown in the following figure. Each of the modules (or components) that comprise the Spring framework can stand on its own or be implemented jointly with one or more of the others. The functionality of each component is as follows:
The core container: The core container provides the essential functionality of the Spring framework. A primary component of the core container is the BeanFactory, an implementation of the Factory pattern. The BeanFactory applies the Inversion of Control (IOC) pattern to separate an application's configuration and dependency specification from the actual application code.
Spring context: The Spring context is a configuration file that provides context information to the Spring framework. The Spring context includes enterprise services such as JNDI, EJB, e-mail, internalization, validation, and scheduling functionality.
Spring AOP: The Spring AOP module integrates aspect-oriented programming functionality directly into the Spring framework, through its configuration management feature. As a result you can easily AOP-enable any object managed by the Spring framework. The Spring AOP module provides transaction management services for objects in any Spring-based application. With Spring AOP you can incorporate declarative transaction management into your applications without relying on EJB components.
Spring DAO: The Spring JDBC DAO abstraction layer offers a meaningful exception hierarchy for managing the exception handling and error messages thrown by different database vendors. The exception hierarchy simplifies error handling and greatly reduces the amount of exception code you need to write, such as opening and closing connections. Spring DAO's JDBC-oriented exceptions comply with its generic DAO exception hierarchy.
Spring ORM: The Spring framework plugs into several ORM frameworks to provide its Object Relational tool, including JDO, Hibernate, and iBATIS SQL Maps. All of these comply with Spring's generic transaction and DAO exception hierarchies.
Spring Web module: The Web context module builds on top of the application context module, providing contexts for Web-based applications. As a result, the Spring framework supports integration with Jakarta Struts. The Web module also eases the tasks of handling multi-part requests and binding request parameters to domain objects.

Spring MVC framework: The Model-View-Controller (MVC) framework is a full-featured MVC implementation for building Web applications. The MVC framework is highly configurable via strategy interfaces and accommodates numerous view technologies including JSP, Velocity, Tiles, iText, and POI.

Q. What is AOP? How does it relate with IOC? What are different tools to utilize AOP?
A: Aspect-oriented programming, or AOP, is a programming technique that allows programmers to modularize crosscutting concerns, or behavior that cuts across the typical divisions of responsibility, such as logging and transaction management. The core construct of AOP is the aspect, which encapsulates behaviours' affecting multiple classes into reusable modules. AOP and IOC are complementary technologies in that both apply a modular approach to complex problems in enterprise application development. In a typical object-oriented development approach you might implement logging functionality by putting logger statements in all your methods and Java classes. In an AOP approach you would instead modularize the logging services and apply them declaratively to the components that required logging. The advantage, of course, is that the Java class doesn't need to know about the existence of the logging service or concern itself with any related code. As a result, application code written using Spring AOP is loosely coupled. The best tool to utilize AOP to its capability is AspectJ. However AspectJ works at he byte code level and you need to use AspectJ compiler to get the AOP features built into your compiled code. Nevertheless AOP functionality is fully integrated into the Spring context for transaction management, logging, and various other features. In general any AOP framework control aspects in three possible ways:

Joinpoints: Points in a program's execution. For example, joinpoints could define calls to specific methods in a class
Pointcuts: Program constructs to designate joinpoints and collect specific context at those points
Advices: Code that runs upon meeting certain conditions. For example, an advice could log a message before executing a joinpoint

Q. What are the advantages of spring framework?
A.

- Spring has layered architecture. Use what you need and leave you don't need now.
- Spring Enables POJO Programming. There is no behind the scene magic here. POJO programming enables continuous integration and testability.
- Dependency Injection and Inversion of Control Simplifies JDBC (Read the first question.)
- Open source and no vendor lock-in.

Q. Explain BeanFactory in spring.
A: Bean factory is an implementation of the factory design pattern and its function is to create and dispense beans. As the bean factory knows about many objects within an application, it is able to create association between collaborating objects as they are instantiated. This removes the burden of configuration from the bean and the client. There are several implementation of BeanFactory. The most useful one is "org.springframework.beans.factory.xml.XmlBeanFactory" It loads its beans based on the definition contained in an XML file. To create an XmlBeanFactory, pass an InputStream to the constructor. The resource will provide the XML to the factory. BeanFactory factory = new XmlBeanFactory(new FileInputStream("myBean.xml"));
This line tells the bean factory to read the bean definition from the XML file. The

bean definition includes the description of beans and their properties. But the bean factory doesn't instantiate the bean yet. To retrieve a bean from a 'BeanFactory', the getBean() method is called. When getBean() method is called, factory will instantiate the bean and begin setting the bean's properties using dependency injection. myBean bean1 = (myBean)factory.getBean("myBean");

Q. Explain the role of ApplicationContext in spring.
A. While Bean Factory is used for simple applications, the Application Context is spring's more advanced container. Like 'BeanFactory' it can be used to load bean definitions, wire beans together and dispense beans upon request. It also provide

1) a means for resolving text messages, including support for internationalization.
2) a generic way to load file resources.
3) events to beans that are registered as listeners.

Because of additional functionality, 'Application Context' is preferred over a BeanFactory. Only when the resource is scarce like mobile devices, 'BeanFactory' is used. The three commonly used implementation of 'Application Context' are

1. ClassPathXmlApplicationContext : It Loads context definition from an XML file located in the classpath, treating context definitions as classpath resources. The application context is loaded from the application's classpath by using the code

```
ApplicationContext    context = new ClassPathXmlApplicationContext("bean.xml");
```

2. FileSystemXmlApplicationContext : It loads context definition from an XML file in the filesystem. The application context is loaded from the file system by using the code

```
ApplicationContext    context = new FileSystemXmlApplicationContext("bean.xml");
```

3. XmlWebApplicationContext : It loads context definition from an XML file contained within a web application.

Q. How does Spring supports DAO in hibernate?
A. Spring’s HibernateDaoSupport class is a convenient super class for Hibernate DAOs. It has handy methods you can call to get a Hibernate Session, or a SessionFactory. The most convenient method is getHibernateTemplate(), which returns a HibernateTemplate. This template wraps Hibernate checked exceptions with runtime exceptions, allowing your DAO interfaces to be Hibernate exception-free.
Example:

```
public class UserDAOHibernate extends HibernateDaoSupport {

        public User getUser(Long id) {
                return (User) getHibernateTemplate().get(User.class, id);
        }
        public void saveUser(User user) {
                getHibernateTemplate().saveOrUpdate(user);
                if (log.isDebugEnabled()) {
                        log.debug("userId set to: " + user.getID());
                }
        }
        public void removeUser(Long id) {
                Object user = getHibernateTemplate().load(User.class, id);
                getHibernateTemplate().delete(user);
        }
}
```

Q. What are the id generator classes in hibernate?
A: increment: It generates identifiers of type long, short or int that are unique only when no other process is inserting data into the same table. It should not the used in the clustered environment.

identity: It supports identity columns in DB2, MySQL, MS SQL Server, Sybase and HypersonicSQL. The returned identifier is of type long, short or int.
sequence: The sequence generator uses a sequence in DB2, PostgreSQL, Oracle, SAP DB, McKoi or a generator in Interbase. The returned identifier is of type long, short or int
hilo: The hilo generator uses a hi/lo algorithm to efficiently generate identifiers of type long, short or int, given a table and column (by default hibernate_unique_key and next_hi respectively) as a source of hi values. The hi/lo algorithm generates identifiers that are unique only for a particular database. Do not use this generator with connections enlisted with JTA or with a user-supplied connection.
seqhilo: The seqhilo generator uses a hi/lo algorithm to efficiently generate identifiers of type long, short or int, given a named database sequence.
uuid: The uuid generator uses a 128-bit UUID algorithm to generate identifiers of type string, unique within a network (the IP address is used). The UUID is encoded as a string of hexadecimal digits of length 32.
guid: It uses a database-generated GUID string on MS SQL Server and MySQL.
native: It picks identity, sequence or hilo depending upon the capabilities of the underlying database.
assigned: lets the application to assign an identifier to the object before save() is called. This is the default strategy if no <generator> element is specified.
select: retrieves a primary key assigned by a database trigger by selecting the row by some unique key and retrieving the primary key value.
foreign: uses the identifier of another associated object. Usually used in conjunction with a <one-to-one> primary key association.

Q. How is a typical spring implementation look like?
A. For a typical Spring Application we need the following files
1. An interface that defines the functions.
2. An Implementation that contains properties, its setter and getter methods, functions etc.,
3. A XML file called Spring configuration file.
4. Client program that uses the function.

Q. How do you define hibernate mapping file in spring?
A. Add the hibernate mapping file entry in mapping resource inside Spring's applicationContext.xml file in the web/WEB-INF directory.

```
<property name="mappingResources">
    <list>
        <value>org/appfuse/model/User.hbm.xml</value>
    </list>
</property>
```

Q. How do you configure spring in a web application?
A. It is very easy to configure any J2EE-based web application to use Spring. At the very least, you can simply add Spring's ContextLoaderListener to your web.xml file:

```
<listener>
      <listener-class>org.springframework.web.context.ContextLoaderListener</listener-
class>
</listener>
```

Q. Can you have xyz.xml file instead of applicationcontext.xml?
A. ContextLoaderListener is a ServletContextListener that initializes when your webapp starts up. By default, it looks for Spring's configuration file at WEB-INF/applicationContext.xml. You can change this default value by specifying a <context-param> element named "contextConfigLocation." Example:

```
<listener>
                <listener-class>org.springframework.web.context.ContextLoaderListener
                                                                        <context-param>
                                        <param-name>contextConfigLocation</param-name>
                                           <param-value>/WEB-INF/xyz.xml</param-value>
</context-param>
</listener-class>
</listener>
```

Q. How do you configure your database driver in spring?
A. Using datasource "org.springframework.jdbc.datasource.DriverManagerDataSource". Example:

```
<bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName">
        <value>org.hsqldb.jdbcDriver</value>
    </property>
    <property name="url">
        <value>jdbc:hsqldb:db/appfuse</value>
    </property>
    <property name="username"><value>sa</value></property>
    <property name="password"><value></value></property>
</bean>
```

Q. How can you configure JNDI instead of datasource in spring applicationcontext.xml?
A. Using "org.springframework.jndi.JndiObjectFactoryBean". Example:

```
<bean id="dataSource"
class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName">
        <value>java:comp/env/jdbc/appfuse</value>
    </property>
</bean>
```

Q. What are the key benefits of Hibernate?
A: These are the key benefits of Hibernate:

- Transparent persistence based on POJOs without byte code processing
- Powerful object-oriented hibernate query language
- Descriptive O/R Mapping through mapping file.
- Automatic primary key generation
- Hibernate cache : Session Level, Query and Second level cache.
- Performance: Lazy initialization, Outer join fetching, Batch fetching

Q. What is hibernate session and session factory? How do you configure sessionfactory in spring configuration file?
A. Hibernate Session is the main runtime interface between a Java application and Hibernate. SessionFactory allows applications to create hibernate session by reading hibernate configurations file hibernate.cfg.xml.

```
// Initialize the Hibernate environment
Configuration cfg = new Configuration().configure();
// Create the session factory
SessionFactory factory = cfg.buildSessionFactory();
// Obtain the new session object
Session session = factory.openSession();
```

The call to Configuration().configure() loads the hibernate.cfg.xml configuration file and initializes the Hibernate environment. Once the configuration is initialized, you can make any additional modifications you desire programmatically. However,

you must make these modifications prior to creating the SessionFactory instance. An instance of SessionFactory is typically created once and used to create all sessions related to a given context.
The main function of the Session is to offer create, read and delete operations for instances of mapped entity classes. Instances may exist in one of three states:

transient: never persistent, not associated with any Session
persistent: associated with a unique Session
detached: previously persistent, not associated with any Session

A Hibernate Session object represents a single unit-of-work for a given data store and is opened by a SessionFactory instance. You must close Sessions when all work for a transaction is completed. The following illustrates a typical Hibernate session:
Session session = null;
UserInfo user = null;
Transaction tx = null;

```
try {
   session = factory.openSession();
   tx = session.beginTransaction();
   user = (UserInfo)session.load(UserInfo.class, id);
   tx.commit();
} catch(Exception e) {
   if (tx != null) {
      try {
         tx.rollback();
      } catch (HibernateException e1) {
         throw new DAOException(e1.toString()); }
   } throw new DAOException(e.toString());
} finally {
   if (session != null) {
      try {
         session.close();
      } catch (HibernateException e) { }
   }
}
```

Q. What is the difference between hibernate get and load methods?
A. The load() method is older; get() was added to Hibernate's API due to user request. The difference is trivial:
The following Hibernate code snippet retrieves a User object from the database:
User user = (User) session.get(User.class, userID);
The get() method is special because the identifier uniquely identifies a single instance of a class. Hence it's common for applications to use the identifier as a convenient handle to a persistent object. Retrieval by identifier can use the cache when retrieving an object, avoiding a database hit if the object is already cached.
Hibernate also provides a load() method: User user = (User) session.load(User.class, userID);
If load() can't find the object in the cache or database, an exception is thrown. The load() method never returns null. The get() method returns null if the object can't be found. The load() method may return a proxy instead of a real persistent instance. A proxy is a placeholder instance of a runtime-generated subclass (through cglib or Javassist) of a mapped persistent class, it can initialize itself if any method is called that is not the mapped database identifier getter-method. On the other hand, get() never returns a proxy. Choosing between get() and load() is easy: If you're certain the persistent object exists, and nonexistence would be considered exceptional, load() is a good option. If you aren't certain there is a persistent instance with the given

identifier, use get() and test the return value to see if it's null. Using load() has a further implication: The application may retrieve a valid reference (a proxy) to a persistent instance without hitting the database to retrieve its persistent state. So load() might not throw an exception when it doesn't find the persistent object in the cache or database; the exception would be thrown later, when the proxy is accessed.

Q. What type of transaction management is supported in hibernate?
A. Hibernate communicates with the database via a JDBC Connection; hence it must support both managed and non-managed transactions.

non-managed in web containers:

```
<bean                                                    id="transactionManager"
class="org.springframework.orm.hibernate.HibernateTransactionManager">
    <property name="sessionFactory">
        <ref local="sessionFactory"/>
    </property>
</bean>
```

managed in application server using JTA:

```
<bean                                                    id="transactionManager"
class="org.springframework.transaction.jta.JtaTransactionManager">
    <property name="sessionFactory">
        <ref local="sessionFactory"/>
    </property>
</bean>
```

Q. What is lazy loading and how do you achieve that in hibernate?
A. Lazy setting decides whether to load child objects while loading the Parent Object. You need to specify parent class.Lazy = true in hibernate mapping file. By default the lazy loading of the child objects is true. This make sure that the child objects are not loaded unless they are explicitly invoked in the application by calling getChild() method on parent. In this case hibernate issues a fresh database call to load the child when getChild() is actully called on the Parent object. But in some cases you do need to load the child objects when parent is loaded. Just make the lazy=false and hibernate will load the child when parent is loaded from the database. Examples: Address child of User class can be made lazy if it is not required frequently. But you may need to load the Author object for Book parent whenever you deal with the book for online bookshop. Hibernate does not support lazy initialization for detached objects. Access to a lazy association outside of the context of an open Hibernate session will result in an exception.

Q. What are the different fetching strategies in Hibernate?
A. Hibernate3 defines the following fetching strategies:

Join fetching - Hibernate retrieves the associated instance or collection in the same SELECT, using an OUTER JOIN.
Select fetching - a second SELECT is used to retrieve the associated entity or collection. Unless you explicitly disable lazy fetching by specifying lazy="false", this second select will only be executed when you actually access the association.
Subselect fetching - a second SELECT is used to retrieve the associated collections for all entities retrieved in a previous query or fetch. Unless you explicitly disable lazy fetching by specifying lazy="false", this second select will only be executed when you actually access the association.Batch fetching - an optimization strategy for select fetching - Hibernate retrieves a batch of entity instances or collections in a single SELECT, by specifying a list of primary keys or foreign keys.

Q. What are different types of cache hibernate supports ?

A. Caching is widely used for optimizing database applications. Hibernate uses two different caches for objects: first-level cache and second-level cache. First-level cache is associated with the Session object, while second-level cache is associated with the Session Factory object. By default, Hibernate uses first-level cache on a per-transaction basis. Hibernate uses this cache mainly to reduce the number of SQL queries it needs to generate within a given transaction. For example, if an object is modified several times within the same transaction, Hibernate will generate only one SQL UPDATE statement at the end of the transaction, containing all the modifications. To reduce database traffic, second-level cache keeps loaded objects at the Session Factory level between transactions. These objects are available to the whole application, not just to the user running the query. This way, each time a query returns an object that is already loaded in the cache, one or more database transactions potentially are avoided. In addition, you can use a query-level cache if you need to cache actual query results, rather than just persistent objects. The query cache should always be used in conjunction with the second-level cache. Hibernate supports the following open-source cache implementations out-of-the-box:

EHCache is a fast, lightweight, and easy-to-use in-process cache. It supports read-only and read/write caching, and memory- and disk-based caching. However, it does not support clustering.
OSCache is another open-source caching solution. It is part of a larger package, which also provides caching functionalities for JSP pages or arbitrary objects. It is a powerful and flexible package, which, like EHCache, supports read-only and read/write caching, and memory- and disk-based caching. It also provides basic support for clustering via either JavaGroups or JMS.
SwarmCache is a simple cluster-based caching solution based on JavaGroups. It supports read-only or nonstrict read/write caching (the next section explains this term). This type of cache is appropriate for applications that typically have many more read operations than write operations.
JBoss TreeCache is a powerful replicated (synchronous or asynchronous) and transactional cache. Use this solution if you really need a true transaction-capable caching architecture.
Commercial Tangosol Coherence cache.

Q. What are the different caching strategies?
A. The following four caching strategies are available:
Read-only: This strategy is useful for data that is read frequently but never updated. This is by far the simplest and best-performing cache strategy.
Read/write: Read/write caches may be appropriate if your data needs to be updated. They carry more overhead than read-only caches. In non-JTA environments, each transaction should be completed when Session.close() or Session.disconnect() is called.
Nonstrict read/write: This strategy does not guarantee that two transactions won't simultaneously modify the same data. Therefore, it may be most appropriate for data that is read often but only occasionally modified.
Transactional: This is a fully transactional cache that may be used only in a JTA environment.

Q. How do you configure 2nd level cach in hibernate?
A. To activate second-level caching, you need to define the hibernate.cache.provider_class property in the hibernate.cfg.xml file as follows:

```
<hibernate-configuration>
                                                    <session-factory>
                                                        <property
name="hibernate.cache.provider_class">org.hibernate.cache.EHCacheProvider</property>
```

```
</session-factory>
</hibernate-configuration>
```

By default, the second-level cache is activated and uses the EHCache provider.
To use the query cache you must first enable it by setting the property hibernate.cache.use_query_cache to true in hibernate.properties.

Q. What is the difference between sorted and ordered collection in hibernate?
A. A sorted collection is sorted in-memory using java comparator, while order collection is ordered at the database level using order by clause.

Q. What are the types of inheritence models and describe how they work like vertical inheritence and horizontal?
A. There are three types of inheritance mapping in hibernate:

Example: Let us take the simple example of 3 java classes. Class Manager and Worker are inherited from Employee Abstract class.
1. Table per concrete class with unions : In this case there will be 2 tables. Tables: Manager, Worker [all common attributes will be duplicated]
2. Table per class hierarchy: Single Table can be mapped to a class hierarchy. There will be only one table in database called 'Employee' that will represent all the attributes required for all 3 classes. But it needs some discriminating column to differentiate between Manager and worker;
3. Table per subclass: In this case there will be 3 tables represent Employee, Manager and Worker