Hibernate and jdbc

Hibernate questions:

Here are the answers to your questions related to Hibernate:

**Q1. Explain Hibernate Architecture?**

Hibernate architecture consists of several components that work together to perform database operations seamlessly. Key components of the architecture include:

* **Configuration**: Reads the configuration files (hibernate.cfg.xml) to establish a connection to the database.
* **SessionFactory**: A factory that creates Sessions. It is a thread-safe object, used for the configuration of Hibernate and is used to create Sessions.
* **Session**: Represents a single unit of work. It is used to retrieve and store data from/to the database and maintains the connection.
* **Transaction**: Used to wrap the database operations to ensure atomicity.
* **Query**: Allows execution of queries like HQL (Hibernate Query Language) and SQL.
* **SessionFactory** and **Session** act as the core components that interact with the database.

**Q2. Explain Differences between Session Vs Session Factory?**

| **Feature** | **Session** | **SessionFactory** |
| --- | --- | --- |
| **Definition** | A single-threaded object used for interacting with the database. | A thread-safe, factory class used to create Session objects. |
| **Creation** | Created by SessionFactory for each interaction. | Created only once, typically at application startup. |
| **Scope** | Works within a single unit of work. | Shared by multiple threads and long-lived in the application. |
| **Caching** | No caching capabilities. | May provide caching functionality. |
| **Lifecycle** | Short-lived, created and destroyed per transaction. | Long-lived, remains throughout the application lifecycle. |

**Q3. What is First Level Cache in Hibernate?**

First Level Cache in Hibernate is a cache associated with a Session object. Each session maintains its own cache that holds data of the entities that have been retrieved or persisted during its lifecycle. The first level cache is mandatory and enabled by default. When an entity is loaded or saved, Hibernate first looks in the session cache before hitting the database. Once the session is closed, this cache is discarded.

**Q4. What is Second Level Cache in Hibernate?**

The Second Level Cache in Hibernate is a session-independent cache shared among multiple sessions. It is used to store entities across different sessions and to reduce the number of database hits. Unlike the first level cache, the second-level cache can be shared between sessions and is configured separately. It is not enabled by default and needs to be configured explicitly using a caching provider like EHCache, Infinispan, or others.

**Q5. What are differences between First and Second Level Cache?**

| **Feature** | **First Level Cache** | **Second Level Cache** |
| --- | --- | --- |
| **Scope** | Session-specific. | SessionFactory-specific and shared across sessions. |
| **Availability** | Available during the lifecycle of a session. | Available throughout the application. |
| **Caching** | Only caches entities within a session. | Caches entities across multiple sessions. |
| **Eviction** | Data is evicted when the session is closed. | Data eviction can be controlled by configuration. |
| **Enabled By Default** | Yes | No, it must be explicitly configured. |

**Q6. What are differences between Hibernate and JDBC?**

| **Feature** | **Hibernate** | **JDBC** |
| --- | --- | --- |
| **Level of Abstraction** | High-level ORM (Object-Relational Mapping) framework. | Low-level API for database interaction. |
| **Configuration** | Requires a configuration file (hibernate.cfg.xml) and mapping files. | Requires manually writing SQL queries for CRUD operations. |
| **Portability** | Database-independent. | Database-dependent (requires changes for different DBs). |
| **Ease of Use** | Provides automatic object mapping and transaction management. | Requires manual mapping between Java objects and database tables. |
| **Caching** | Supports caching (first and second level cache). | No built-in caching mechanism. |
| **SQL Queries** | Supports HQL (Hibernate Query Language), which is database-independent. | Requires writing direct SQL queries. |

**Q7. What are differences between get and load methods?**

| **Feature** | **get()** | **load()** |
| --- | --- | --- |
| **Return Type** | Returns null if the entity is not found. | Throws ObjectNotFoundException if the entity is not found. |
| **Performance** | Immediately hits the database. | May use lazy loading (does not hit the database unless required). |
| **Usage** | Suitable when the entity may or may not exist in the database. | Suitable when the entity is expected to exist in the database. |
| **Null Handling** | Returns null if no data is found. | Throws an exception if the entity is not found. |

**Q8. Explain N+1 Problem in Hibernate?**

The N+1 problem occurs when a query is made to fetch a collection of entities (e.g., a list of parent entities), and then for each of these entities, an additional query is made to fetch related data (e.g., child entities). This results in a total of N+1 SQL queries—one for fetching the parent entities and one for each child entity, leading to inefficient database access and performance issues. This problem can be mitigated using fetch joins or batch fetching.

**Q9. Difference between JPA Vs Hibernate?**

| **Feature** | **JPA** | **Hibernate** |
| --- | --- | --- |
| **Definition** | JPA (Java Persistence API) is a specification for ORM in Java. | Hibernate is an implementation of JPA and provides additional features. |
| **Standardization** | A specification that provides guidelines for ORM frameworks. | An actual framework that implements the JPA specification. |
| **Features** | Focuses on object-relational mapping and persistence management. | Provides additional features beyond JPA like second-level caching, and custom query languages. |
| **Portability** | JPA implementations can be swapped easily (e.g., Hibernate, EclipseLink). | Hibernate is a specific implementation, not interchangeable with others easily. |

**Q10. When to use @version and @Audited in Hibernate?**

* **@Version**: The @Version annotation is used for optimistic locking. It ensures that no other transaction has modified the entity before you update it. It helps avoid data inconsistencies and conflicts in concurrent transaction scenarios by adding a version column in the database.
* **@Audited**: The @Audited annotation is used to track changes made to an entity. It is commonly used with Hibernate Envers to automatically audit changes made to entity data, including additions, updates, and deletions. It keeps track of the historical versions of the entity data.

Let me know if you'd like more details or have any further questions!

Jdbc interview questions:

Here are the **JDBC interview questions** categorized based on **levels**: **Beginner**, **Intermediate**, and **Advanced**.

**Beginner Level**

**Q1. What is JDBC?**

**Answer**:  
JDBC (Java Database Connectivity) is an API in Java that allows Java programs to interact with databases. It provides methods for querying, updating, and managing databases.

**Q2. What is a Connection object in JDBC?**

**Answer**:  
The Connection object in JDBC is used to establish a connection with the database. It provides methods to create Statement, PreparedStatement, CallableStatement, and manages transactions.

**Q3. What is the purpose of the Statement interface in JDBC?**

**Answer**:  
The Statement interface in JDBC is used to execute SQL queries against the database. It is typically used for static SQL queries that are not precompiled.

**Q4. What is a ResultSet in JDBC?**

**Answer**:  
A ResultSet is a container for the data retrieved from a database query. It allows you to iterate over the rows of the result set and retrieve column values.

**Q5. What is a SQLException in JDBC?**

**Answer**:  
SQLException is an exception class in JDBC that provides detailed information about a database error. It includes the error code, SQL state, and message to help identify issues during database operations.

**Q6. How do you close JDBC resources?**

**Answer**:  
JDBC resources like Connection, Statement, and ResultSet should be closed explicitly to avoid memory leaks. This can be done in the finally block or using the **try-with-resources** statement.

Example using **try-with-resources**:

try (Connection conn = DriverManager.getConnection(url, username, password);

Statement stmt = conn.createStatement();

ResultSet rs = stmt.executeQuery("SELECT \* FROM users")) {

// process the result set

} catch (SQLException e) {

e.printStackTrace();

}

**Intermediate Level**

**Q7. What is the difference between Statement and PreparedStatement?**

**Answer**:

* **Statement** is used for executing simple SQL queries. It does not support parameterized queries and can be prone to SQL injection.
* **PreparedStatement** is used for executing precompiled SQL queries and supports parameterized queries, improving performance and security by preventing SQL injection.

**Q8. What is the difference between execute() and executeQuery() methods in JDBC?**

**Answer**:

* **execute()**: It can execute any SQL query and returns a boolean indicating whether the result is a ResultSet or an update count.
* **executeQuery()**: It is specifically used for executing SELECT queries and returns a ResultSet containing the results.

**Q9. What is the setAutoCommit() method in JDBC?**

**Answer**:  
setAutoCommit() is a method in JDBC that controls whether each SQL statement is committed automatically after execution. When set to false, you can explicitly control when to commit or rollback transactions.

**Q10. What are the types of JDBC drivers?**

**Answer**:

1. **Type-1 (JDBC-ODBC Bridge Driver)**: Uses ODBC to connect to the database. It is slow and not recommended for production.
2. **Type-2 (Native-API Driver)**: Uses database-specific APIs to connect and requires a native library.
3. **Type-3 (Network Protocol Driver)**: Uses a middle-tier server to connect to the database. The client does not need database-specific drivers.
4. **Type-4 (Thin Driver)**: A pure Java driver that communicates directly with the database.

**Q11. How do you handle transactions in JDBC?**

**Answer**:  
In JDBC, transactions are managed using the Connection object:

1. **Disable auto-commit** using connection.setAutoCommit(false).
2. Perform SQL operations.
3. **Commit** changes using connection.commit() or **rollback** using connection.rollback() in case of failure.
4. Re-enable **auto-commit** using connection.setAutoCommit(true) after the transaction.

**Q12. What is the difference between executeUpdate() and execute()?**

**Answer**:

* **executeUpdate()**: Used for executing SQL statements that modify the database (like INSERT, UPDATE, DELETE). It returns an integer representing the number of affected rows.
* **execute()**: Can execute any SQL statement, including queries. It returns a boolean indicating whether the result is a ResultSet.

**Advanced Level**

**Q13. What is Batch Processing in JDBC?**

**Answer**:  
Batch processing in JDBC allows executing multiple SQL statements in a single batch to improve performance. By using addBatch() and executeBatch(), you can group several SQL operations together and execute them at once.

Example:

Statement stmt = connection.createStatement();

stmt.addBatch("INSERT INTO employees (id, name) VALUES (1, 'John')");

stmt.addBatch("INSERT INTO employees (id, name) VALUES (2, 'Jane')");

int[] results = stmt.executeBatch();

**Q14. What is SQL Injection and how can it be prevented in JDBC?**

**Answer**:  
SQL injection occurs when user input is directly included in SQL queries, allowing malicious users to manipulate the database.  
It can be prevented by using **PreparedStatement**, which automatically escapes input values and prevents malicious input from altering the query.

Example using **PreparedStatement**:

String sql = "SELECT \* FROM users WHERE username = ? AND password = ?";

PreparedStatement pstmt = connection.prepareStatement(sql);

pstmt.setString(1, username);

pstmt.setString(2, password);

ResultSet rs = pstmt.executeQuery();

**Q15. What is the difference between get() and load() methods in Hibernate and how does it relate to JDBC?**

**Answer**:

* In **Hibernate**, get() fetches the object from the database immediately and returns null if not found. load() retrieves the object lazily, throwing an exception if the entity is not found.
* In **JDBC**, these behaviors need to be manually handled with ResultSet to check for null values and handle lazy fetching manually (for performance reasons, using PreparedStatement or CallableStatement).

**Q16. How do you perform Transaction Management in JDBC?**

**Answer**:  
Transaction management in JDBC involves:

1. Starting a transaction by disabling auto-commit.
2. Executing multiple SQL queries.
3. Committing the transaction using connection.commit().
4. Rolling back if there is an error using connection.rollback().
5. Enabling auto-commit again after the transaction.

Example:

Connection conn = DriverManager.getConnection(url, user, pass);

conn.setAutoCommit(false);

try {

Statement stmt = conn.createStatement();

stmt.executeUpdate("UPDATE users SET balance = balance - 100 WHERE id = 1");

stmt.executeUpdate("UPDATE users SET balance = balance + 100 WHERE id = 2");

conn.commit();

} catch (SQLException e) {

conn.rollback();

} finally {

conn.setAutoCommit(true);

conn.close();

}

**Q17. How can you prevent resource leaks in JDBC?**

**Answer**:  
To prevent resource leaks, always close the JDBC resources (Connection, Statement, ResultSet) properly. Use **try-with-resources** (Java 7 and above) or a finally block to close resources:

try (Connection conn = DriverManager.getConnection(url, user, pass);

Statement stmt = conn.createStatement();

ResultSet rs = stmt.executeQuery("SELECT \* FROM users")) {

while (rs.next()) {

// Process data

}

} catch (SQLException e) {

e.printStackTrace();

}

**Q18. What are the common performance issues in JDBC, and how can you resolve them?**

**Answer**:

* **Problem**: Too many database connections.
  + **Solution**: Use connection pooling to reduce the overhead of establishing connections.
* **Problem**: Too many SQL queries (N+1 problem).
  + **Solution**: Use batch processing or fetch joins to minimize the number of queries.
* **Problem**: Inefficient queries.
  + **Solution**: Use prepared statements and ensure proper indexing on the database side.

These questions cover a range of beginner, intermediate, and advanced topics related to JDBC and will help you prepare for your interview at different levels of expertise. Let me know if you need more detailed explanations for any question!

Application properties interview questions ?

Sure! Here are **level-wise questions** on **application.properties** for Spring Boot, categorized as **Beginner**, **Intermediate**, and **Advanced**.

**Beginner Level**

**Q1. What is the purpose of the application.properties file in Spring Boot?**

**Answer**:  
The application.properties file is used to configure various properties for a Spring Boot application. It contains key-value pairs that control the behavior of the application, such as database connections, server configurations, logging settings, etc.

**Q2. How do you specify the server port in application.properties?**

**Answer**:  
You can specify the port for the Spring Boot application using the server.port property:

server.port=8080

This sets the server port to 8080.

**Q3. How do you configure a database connection in application.properties?**

**Answer**:  
To configure a database connection, you typically set the following properties:

spring.datasource.url=jdbc:mysql://localhost:3306/mydb

spring.datasource.username=root

spring.datasource.password=secret

spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

This configures the database connection to a MySQL instance.

**Q4. How do you set the logging level in application.properties?**

**Answer**:  
You can set the logging level for different packages using the logging.level property:

logging.level.org.springframework=INFO

logging.level.com.myapp=DEBUG

This sets the logging level for Spring Framework to INFO and for the custom package com.myapp to DEBUG.

**Q5. How do you disable the Spring Boot banner?**

**Answer**:  
You can disable the Spring Boot startup banner with the following property:

spring.main.banner-mode=off

This will prevent the banner from displaying when the application starts.

**Intermediate Level**

**Q6. How do you configure Spring Boot to use a custom application properties file?**

**Answer**:  
You can specify a custom properties file using the spring.config.location property:

spring.config.location=classpath:/config/custom-application.properties

This allows you to load properties from a custom file.

**Q7. How do you change the context path of a Spring Boot application?**

**Answer**:  
You can change the context path using the server.servlet.context-path property:

server.servlet.context-path=/myapp

This will set the context path to /myapp.

**Q8. How do you configure Spring Security in application.properties?**

**Answer**:  
You can configure basic security settings, such as username and password, using the following properties:

spring.security.user.name=admin

spring.security.user.password=adminpass

This sets a default username (admin) and password (adminpass) for the application.

**Q9. How do you configure the session timeout in Spring Boot?**

**Answer**:  
You can configure the session timeout using the server.servlet.session.timeout property:

server.servlet.session.timeout=15m

This sets the session timeout to 15 minutes.

**Q10. How do you configure Spring Boot to connect to a PostgreSQL database?**

**Answer**:  
To configure PostgreSQL, you would typically use the following properties:

spring.datasource.url=jdbc:postgresql://localhost:5432/mydb

spring.datasource.username=postgres

spring.datasource.password=password

spring.datasource.driver-class-name=org.postgresql.Driver

These properties set the connection details for PostgreSQL.

**Advanced Level**

**Q11. How do you enable profiles in Spring Boot using application.properties?**

**Answer**:  
You can enable specific profiles by using the spring.profiles.active property:

spring.profiles.active=dev

This activates the dev profile, loading the properties from application-dev.properties.

**Q12. How do you configure multiple data sources in application.properties?**

**Answer**:  
To configure multiple data sources, you can provide properties for each data source with a unique prefix:

spring.datasource.primary.url=jdbc:mysql://localhost:3306/db1

spring.datasource.primary.username=root

spring.datasource.primary.password=secret

spring.datasource.secondary.url=jdbc:mysql://localhost:3306/db2

spring.datasource.secondary.username=root

spring.datasource.secondary.password=secret

This sets up two data sources, and you will need to configure them in your Java configuration using @Primary and @Qualifier annotations.

**Q13. How do you set a custom location for the application properties in Spring Boot?**

**Answer**:  
You can set a custom location for the application.properties file using the spring.config.location property:

spring.config.location=file:/path/to/external/config/

This loads properties from the specified directory.

**Q14. How do you configure external property files in Spring Boot?**

**Answer**:  
You can use the spring.config.additional-location property to specify additional external property files:

spring.config.additional-location=file:/path/to/external/config/

This adds external configuration to the application's configuration.

**Q15. How do you configure a custom health check in Spring Boot using application.properties?**

**Answer**:  
You can enable or configure custom health checks using the management.health properties:

management.health.db.enabled=true

management.health.diskspace.enabled=false

This configures the database and disables the disk space health check.

**Q16. How do you configure a custom message source for i18n (internationalization) in Spring Boot?**

**Answer**:  
You can specify the basename of your message properties file using:

spring.messages.basename=i18n/messages

This tells Spring Boot to look for message files (e.g., messages.properties) in the i18n folder.

**Q17. How do you handle HTTP error pages in Spring Boot using application.properties?**

**Answer**:  
You can configure custom error paths using the server.error properties:

server.error.path=/error

server.error.whitelabel.enabled=false

This sets up custom error handling by disabling the default white-label error page and specifying a custom error path.

**Q18. How can you configure Jackson JSON properties in application.properties?**

**Answer**:  
Jackson properties are used to configure JSON serialization and deserialization:

spring.jackson.date-format=yyyy-MM-dd

spring.jackson.serialization.INDENT\_OUTPUT=true

This configures Jackson to use a specific date format and enables pretty-printing for JSON output.

These **beginner**, **intermediate**, and **advanced** questions about the application.properties file in Spring Boot will help you understand how to configure and customize various aspects of a Spring Boot application. Let me know if you need more details!