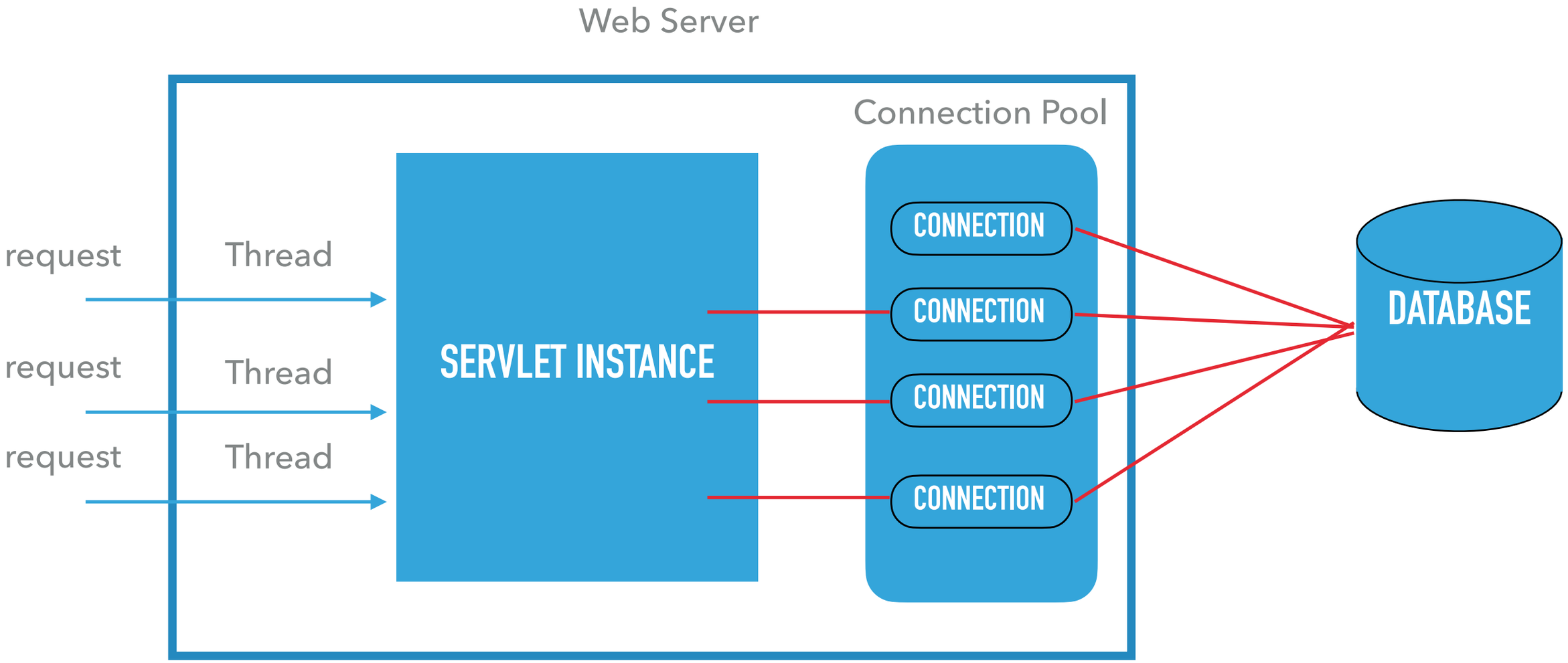
INTRODUCTION TO HIBERNATE

LESSON 9

DATABASE CONNECTION POOL

- Database *connection* pooling is a method used to keep database connections open so they can be reused by others.
- Typically, opening a database connection is an expensive operation, especially if the database is remote. You have to open up network sessions, authenticate, have authorisation checked, and so on. Pooling keeps the connections active so that, when a connection is later requested, one of the active ones is used in preference to having to create another one


Web Server
Connection Pool
request
Thread
request
Thread
request
Thread
SERVLET INSTANCE
CONNECTION
CONNECTION
CONNECTION
CONNECTION
DATABASE

HIBERNATE

OVERVIEW

WHAT IS JDBC?

- JDBC stands for **Java Database Connectivity**. It provides a set of Java API for accessing the relational databases from Java program. These Java APIs enables Java programs to execute SQL statements and interact with any SQL compliant database.
- JDBC provides a flexible architecture to write a database independent application that can run on different platforms and interact with different DBMS without any modification.

Pros of JDBC	Cons of JDBC
Clean and simple SQL processing	Complex if it is used in large projects
Good performance with large data	Large programming overhead
Very good for small applications	No encapsulation
Simple syntax so easy to learn	Hard to implement MVC concept
	Query is DBMS specific

WHAT IS OBJECT RELATIONAL MAPPING?

- ORM stands for **O**bject-**R**elational **M**apping (ORM) is a programming technique for converting data between relational databases and object oriented programming languages such as Java, C#, etc.

HIBERNATE

- Hibernate is an **O**bject-**R**elational **M**apping (ORM) solution for JAVA. It is an open source persistent framework created by Gavin King in 2001. It is a powerful, high performance Object-Relational Persistence and Query service for any Java Application.
- Hibernate maps Java classes to database tables and from Java data types to SQL data types and relieves the developer from 95% of common data persistence related programming tasks.
- Hibernate sits between traditional Java objects and database server to handle all the works in persisting those objects based on the appropriate O/R mechanisms and patterns.


JAVA OBJECT
JAVA OBJECT
JAVA OBJECT
JAVA OBJECT
JAVA OBJECT
JAVA OBJECT
ORM/HIBERNATE
DATABASE

HIBERNATE ADVANTAGES

- Hibernate takes care of mapping Java classes to database tables using XML files and without writing any line of code.
- Provides simple APIs for storing and retrieving Java objects directly to and from the database.
- If there is change in the database or in any table, then you need to change the XML file properties only.
- Abstracts away the unfamiliar SQL types and provides a way to work around familiar Java Objects.

- Hibernate does not require an application server to operate.
- Manipulates Complex associations of objects of your database.
- Minimizes database access with smart fetching strategies.
- Provides simple querying of data.

SUPPORTED DATABASES

- Hibernate supports almost all the major RDBMS.
    - HSQL Database Engine
    - DB2/NT
    - MySQL
    - PostgreSQL
    - FrontBase
    - Oracle
    - Microsoft SQL Server Database
    - Sybase SQL Server
    - Informix Dynamic Server

HIBERNATE ARCHITECTURE

- Hibernate has a layered architecture which helps the user to operate without having to know the underlying APIs. Hibernate makes use of the database and configuration data to provide persistence services (and persistent objects) to the application.

JAVA APPLICATION

PERSISTENT OBJECT

HIBERNATE

CONFIGURATION

SESSION FACTORY

SESSION

TRANSACTION

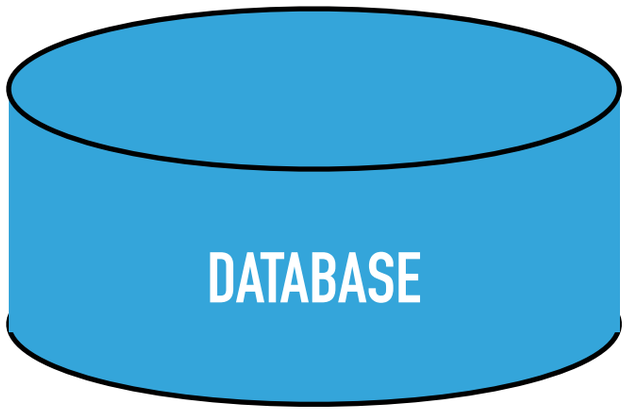
QUERY

CRITERIA

JTA

JDBC

JNDI



- Hibernate uses various existing Java APIs, like JDBC, Java Transaction API(JTA), and Java Naming and Directory Interface (JNDI). JDBC provides a rudimentary level of abstraction of functionality common to relational databases, allowing almost any database with a JDBC driver to be supported by Hibernate. JNDI and JTA allow Hibernate to be integrated with J2EE application servers

CONFIGURATION OBJECT

- The Configuration object is the first Hibernate object you create in any Hibernate application. It is usually created only once during application initialization. It represents a configuration or properties file required by the Hibernate

SESSION FACTORY OBJECT

- Configuration object is used to create a SessionFactory object which in turn configures Hibernate for the application using the supplied configuration file and allows for a Session object to be instantiated. The SessionFactory is a thread safe object and used by all the threads of an application.
- The SessionFactory is a heavyweight object; it is usually created during application start up and kept for later use. You would need one SessionFactory object per database using a separate configuration file. So, if you are using multiple databases, then you would have to create multiple SessionFactory objects.

```java
@Bean
public Properties hibernateProperties(
        @Value("org.hibernate.dialect.MySQLDialect") String dialect,
        @Value("true") boolean showSql,
        @Value("true") boolean formatSql,
        @Value("validate") String hbm2ddl) {

    Properties properties = new Properties();
    properties.put("hibernate.dialect", dialect);
    properties.put("hibernate.show_sql", showSql);
    properties.put("hibernate.format_sql", formatSql);
    properties.put("hibernate.hbm2ddl.auto", hbm2ddl);

    return properties;
}

@Bean
public SessionFactory sessionFactory(DataSource dataSource,
                                     @Value("com.javaguru.todolist") String packagesToScan,
                                     Properties hibernateProperties) throws Exception {

    LocalSessionFactoryBean sessionFactoryBean = new LocalSessionFactoryBean();
    sessionFactoryBean.setDataSource(dataSource);
    sessionFactoryBean.setPackagesToScan(packagesToScan);
    sessionFactoryBean.setHibernateProperties(hibernateProperties);
    sessionFactoryBean.afterPropertiesSet();
    return sessionFactoryBean.getObject();
}

@Bean
public PlatformTransactionManager transactionManager(SessionFactory sessionFactory) {
    return new HibernateTransactionManager(sessionFactory);
}
```

```java
@Bean
public LocalSessionFactoryBean sessionFactory() {
    LocalSessionFactoryBean sessionFactory = new LocalSessionFactoryBean();
    sessionFactory.setDataSource(dataSource());
    sessionFactory.setPackagesToScan("com.javaguru");
    sessionFactory.setHibernateProperties(hibernateProperties());

    return sessionFactory;
}

@Bean
public PlatformTransactionManager hibernateTransactionManager() {
    HibernateTransactionManager transactionManager
            = new HibernateTransactionManager();
    transactionManager.setSessionFactory(sessionFactory().getObject());
    return transactionManager;
}

private final Properties hibernateProperties() {
    Properties hibernateProperties = new Properties();
    hibernateProperties.setProperty(
            "hibernate.hbm2ddl.auto", "create-drop");
    hibernateProperties.setProperty(
            "hibernate.dialect", "org.hibernate.dialect.MySQL8Dialect");

    return hibernateProperties;
}
```

SESSION OBJECT

- A Session is used to get a physical connection with a database. The Session object is lightweight and designed to be instantiated each time an interaction is needed with the database. Persistent objects are saved and retrieved through a Session object.
- The session objects should not be kept open for a long time because they are not usually thread safe and they should be created and destroyed them as needed. The main function of the Session is to offer, create, read, and delete operations for instances of mapped entity classes.

SESSION

- The session opens a single database connection when it is created, and holds onto it until the session is closed. Every object that is loaded by Hibernate from the database is associated with the session, allowing Hibernate to automatically persist objects that are modified, and allowing Hibernate to implement functionality such as lazy-loading.

CRITERIA OBJECT

- Criteria objects are used to create and execute object oriented criteria queries to retrieve objects.

CREATE CRITERIA

- The Hibernate **Session** interface provides **createCriteria()** method, which can be used to create a **Criteria** object that returns instances of the persistence object's class when your application executes a criteria query.

```
public List<Task> findAll() {
    return sessionFactory.getCurrentSession().createCriteria(Task.class)
            .list();
}
```

RESTRICTIONS WITH CRITERIA

- You can use **add()** method available for **Criteria** object to add restriction for a criteria query.
- SELECT * FROM tasks WHERE id = ?

```
public Optional<Task> findTaskById(Long id) {
    Task task = (Task) sessionFactory.getCurrentSession().createCriteria(Task.class)
            .add(Restrictions.eq("id", id))
            .uniqueResult();
    return Optional.ofNullable(task);
}
```

QUERY OBJECT

- Query objects use SQL or Hibernate Query Language (HQL) string to retrieve data from the database and create objects. A Query instance is used to bind query parameters, limit the number of results returned by the query, and finally to execute the query.

QUERY

- You can use native SQL to express database queries if you want to utilize database-specific features such as query hints or the CONNECT keyword in Oracle. Hibernate 3.x allows you to specify handwritten SQL, including stored procedures, for all create, update, delete, and load operations.

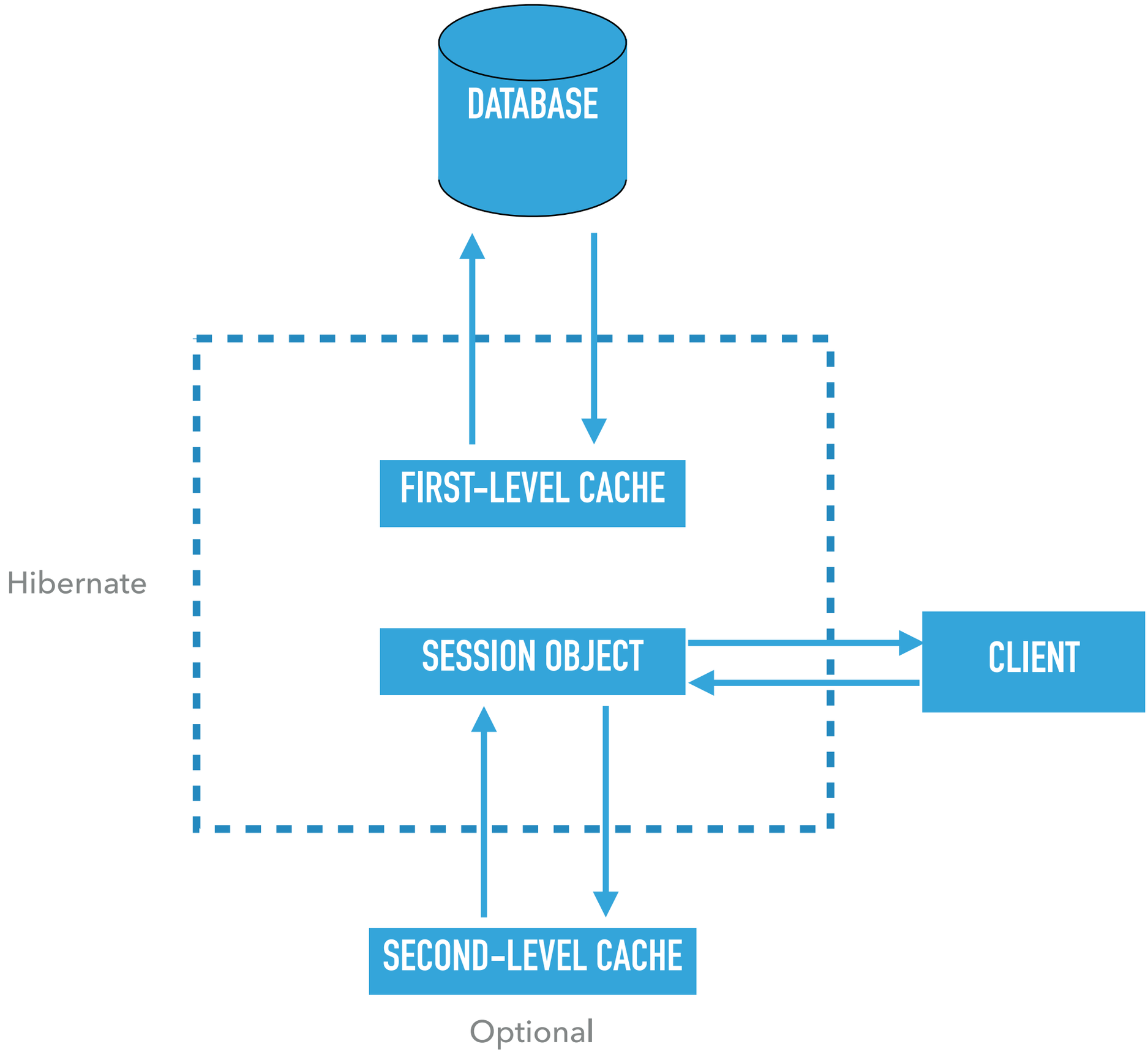
```
public boolean existsByName(String name) {
    String query = "select case when count(*)> 0 " +
            "then true else false end " +
            "from Task where name=“ + name;
    return (boolean) sessionFactory.getCurrentSession().createQuery(query)
            .setMaxResults(1)
            .uniqueResult();
}
```

TRANSACTION OBJECT

- Transaction represents a unit of work with the database and most of the RDBMS supports transaction functionality. Transactions in Hibernate are handled by an underlying transaction manager and transaction (from JDBC or JTA).

HIBERNATE CACHING

- Caching is a mechanism to enhance the performance of a system. It is a buffer memory that lies between the application and the database. Cache memory stores recently used data items in order to reduce the number of database hits as much as possible.


DATABASE
FIRST-LEVEL CACHE
Hibernate
SESSION OBJECT
CLIENT
SECOND-LEVEL CACHE
Optional

FIRST-LEVEL CACHE

- The first-level cache is the Session cache and is a mandatory cache through which all requests must pass. The Session object keeps an object under its own power before committing it to the database.

- If you issue multiple updates to an object, Hibernate tries to delay doing the update as long as possible to reduce the number of update SQL statements issued. If you close the session, all the objects being cached are lost and either persisted or updated in the database.

HIBERNATE ANNOTATIONS

HIBERNATE ANNOTATIONS

- Hibernate Annotations is the powerful way to provide the metadata for the Object and Relational Table mapping. All the metadata is clubbed into the POJO java file along with the code, this helps the user to understand the table structure and POJO simultaneously during the development.

```java
@Entity
@Table(name = "tasks")
public class Task {

    @Id
    @Column(name = "id")
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    @Column(name = "name")
    private String name;
    @Column(name = "description")
    private String description;

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }

}
```

- Hibernate detects that the @Id annotation is on a field and assumes that it should access properties of an object directly through fields at runtime. If you placed the @Id annotation on the getId() method, you would enable access to properties through getter and setter methods by default. Hence, all other annotations are also placed on either fields or getter methods, following the selected strategy.

@Entity Annotation

- The EJB 3 standard annotations are contained in the **javax.persistence** package, so we import this package as the first step. Second, we used the **@Entity** annotation to the Task class, which marks this class as an entity bean, so it must have a no-argument constructor that is visible with at least protected scope.

@Table Annotation

- The @Table annotation allows you to specify the details of the table that will be used to persist the entity in the database.
- The @Table annotation provides four attributes, allowing you to override the name of the table, its catalogue, and its schema, and enforce unique constraints on columns in the table.

@Id and @GeneratedValue Annotations

- Each entity bean will have a primary key, which you annotate on the class with the **@Id** annotation. The primary key can be a single field or a combination of multiple fields depending on your table structure.

- By default, the @Id annotation will automatically determine the most appropriate primary key generation strategy to be used but you can override this by applying the **@GeneratedValue** annotation, which takes two parameters **strategy** and **generator** that I'm not going to discuss here, so let us use only the default key generation strategy. Letting Hibernate determine which generator type to use makes your code portable between different databases.

@Column Annotation

- The @Column annotation is used to specify the details of the column to which a field or property will be mapped. You can use column annotation with the following most commonly used attributes:
    - **name** attribute permits the name of the column to be explicitly specified.
    - **length** attribute permits the size of the column used to map a value particularly for a String value.
    - **nullable** attribute permits the column to be marked NOT NULL when the schema is generated.
    - **unique** attribute permits the column to be marked as containing only unique values.

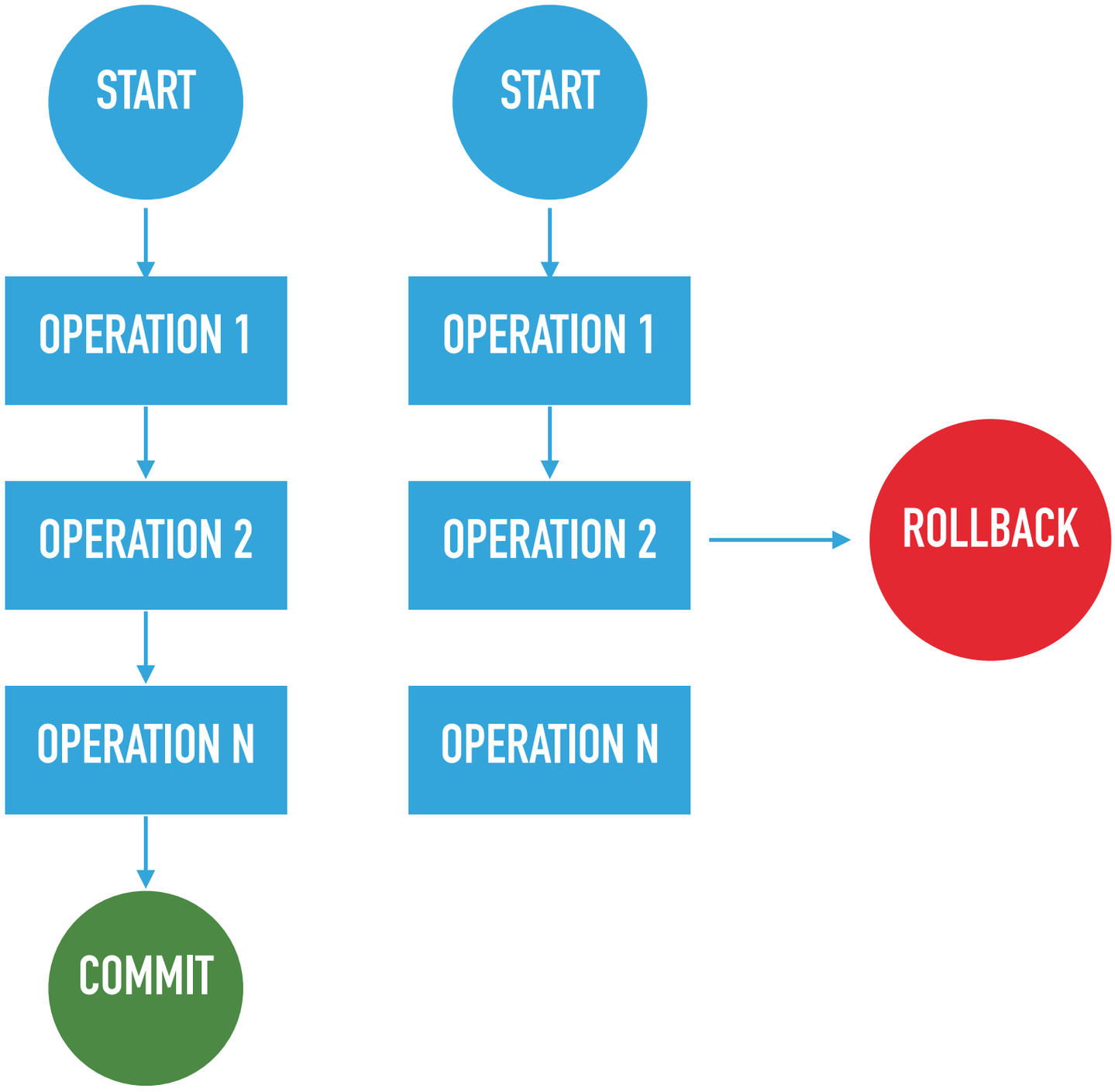
TRANSACTIONS

- A transaction, in the context of a database, is a logical unit that is independently executed for data retrieval or updates. In relational databases, database transactions must be atomic, consistent, isolated and durable -summarized as the ACID acronym.
- Transactions are completed by COMMIT or ROLLBACK SQL statements, which indicate a transaction's beginning or end.
- The ACID acronym defines the properties of a database transaction.

ACID

- Atomicity: A transaction must be fully complete, saved (committed) or completely undone (rolled back). A sale in a retail store database illustrates a scenario which explains atomicity, e.g., the sale consists of an inventory reduction and a record of incoming cash. Both either happen together or do not happen - it's all or nothing.
- Consistency: The transaction must be fully compliant with the state of the database as it was prior to the transaction. In other words, the transaction cannot break the database's constraints. For example, if a database table's Phone Number column can only contain numerals, then consistency dictates that any transaction attempting to enter an alphabetical letter may not commit.

- Isolation: Transaction data must not be available to other transactions until the original transaction is committed or rolled back.
- Durability: Successful transaction must permanently change the state of a system, and before ending it, the state changes are recorded in a persisted transaction log. If our system is suddenly affected by a system crash or a power outage, then all unfinished committed transactions may be replayed.


START
OPERATION 1
OPERATION 2
OPERATION N
COMMIT
START
OPERATION 1
OPERATION 2
OPERATION N
ROLLBACK

JPA AND TRANSACTION MANAGEMENT

- It's important to notice that JPA on itself does not provide any type of declarative transaction management. When using JPA outside of a dependency injection container, transactions need to be handled programatically by the developer

```
UserTransaction utx = entityManager.getTransaction();

try {
    utx.begin();
    businessLogic();
    utx.commit();
} catch(Exception ex) {
    utx.rollback();
    throw ex;
}
```

- This way of managing transactions makes the scope of the transaction very clear in the code, but it has several disavantages:
  - it's repetitive and error prone
  - any error can have a very high impact
  - errors are hard to debug and reproduce
  - this decreases the readability of the code base
  - what if this method calls another transactional method?

Using Spring @Transactional Annotation

- With Spring @Transactional, the above code gets reduced to simply this:

```
@Transactional
public void businessLogic() {
... use entity manager inside a transaction ...
}
```

- This is much more convenient and readable, and is currently the recommended way to handle transactions in Spring.
- By using @Transactional, many important aspects such as transaction propagation are handled automatically. In this case if another transactional method is called by businessLogic(), that method will have the option of joining the ongoing transaction.
- One potential downside is that this powerful mechanism hides what is going on under the hood, making it hard to debug when things don't work.

- One of the key points about @Transactional is that there are two separate concepts to consider, each with it's own scope and life cycle:
  - the persistence context
  - the database transaction
- The transactional annotation itself defines the scope of a single database transaction. The database transaction happens inside the scope of a *persistence context*.

- The persistence context is in JPA the EntityManager, implemented internally using a Hibernate Session (when using Hibernate as the persistence provider).

The Transactional Aspect

- The Transactional Aspect is an 'around' aspect that gets called both before and after the annotated business method. The concrete class for implementing the aspect is TransactionInterceptor.
- The Transactional Aspect has two main responsibilities:
  - At the 'before' moment, the aspect provides a hook point for determining if the business method about to be called should run in the scope of an ongoing database transaction, or if a new separate transaction should be started.
  - At the 'after' moment, the aspect needs to decide if the transaction should be committed, rolled back or left running.
- At the 'before' moment the Transactional Aspect itself does not contain any decision logic, the decision to start a new transaction if needed is delegated to the Transaction Manager.

@EnableTransactionManagement Annotation

- `@EnableTransactionManagement is` responsible for registering the necessary Spring components that power annotation-driven transaction management, such as the TransactionInterceptor and the proxy- or AspectJ-based advice that weave the interceptor into the call stack when `JdbcFooRepository's @Transactional` methods are invoked.

REFERENCES

- http://hibernate.org/
- http://hibernate.org/orm/documentation/
- https://www.tutorialspoint.com/hibernate/
- https://dzone.com/articles/how-does-spring-transactional