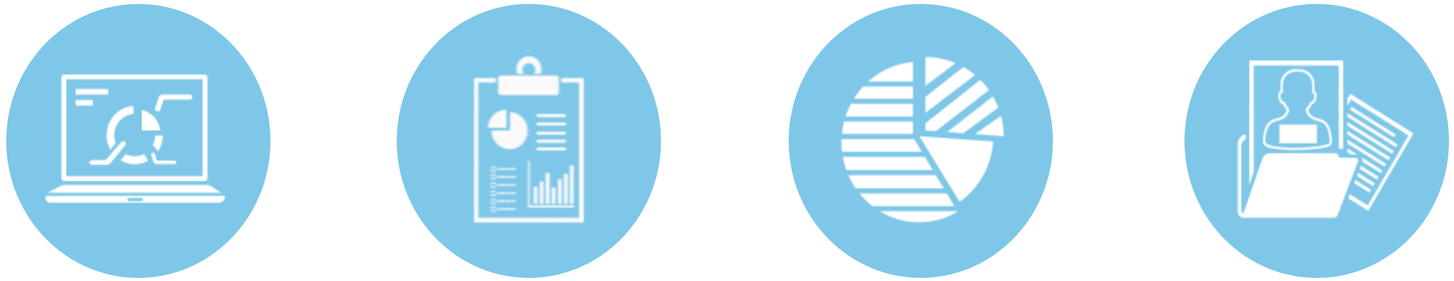
Advanced Java

Lesson 9—Spring JDBC and Transaction Management



Learning Objectives

- Explain Spring JDBC/DAO (Data Access Object)
- Explain how to Implement Spring JDBC in an Application
- Integrate Spring with Hibernate
- Explain how to create Application using Spring Hibernate Template
- Describe Spring JDBC Transaction Management

Advanced Java

Topic 1—Spring JDBC /DAO (Data Access Object)

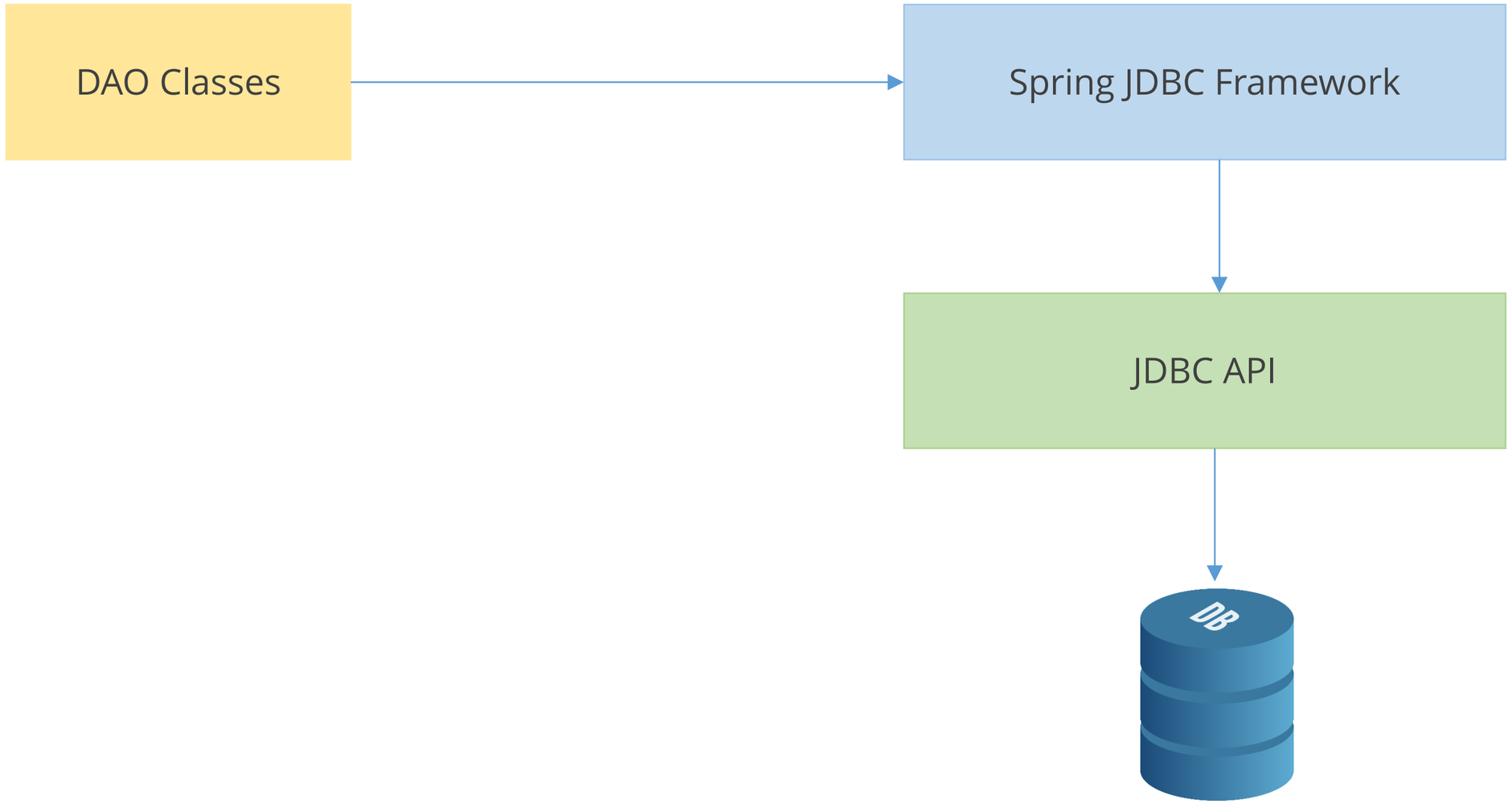
Spring JDBC /DAO (Data Access Object)

- It is used to build Data Access Layer of enterprise application.
- It is a mechanism to connect to the database and execute SQL queries.
- The database-related logic has to placed in separate classes, called DAO classes.

JDBC vs. Spring JDBC

Difference based on	JDBC	Spring JDBC
Transaction Management Logic: (enable or disable transactions declaratively)	It doesn't support	Spring JDBC supports
Distribution based Transaction Management	It doesn't support	Spring JDBC supports
Annotation based transaction management	It doesn't support	Spring JDBC supports
Resource Allocating Logic	Connection, Statement, and PreparedStatement have to be allocated explicitly by the developer in JDBC.	Spring provides org.springframework.jdbcTemplate object. It abstracts the Database Resource Allocation Logic.
Resource Releasing Logic	In JDBC, developer explicitly writes the logic to release the database resources.	Spring provides org.springframework.jdbc.core.jdbcTemplate object.
Exception Handling	JDBC throws java.sql.SQLException. There won't be specific exceptions for specific problems.	Spring provides Fine Grind Exception Handling mechanism to deal with Database.

Spring JDBC Architecture



Advantage of Spring JDBC API

- It provides the capability to establish the connection and interact with database
- It simplifies the development of JDBC, Hibernate, JPA ,and JDO
- It provides multiple templates to interact with DB:

1. JdbcTemplate
2. HibernateTemplate
3. JdoTemplate
4. JpaTemplate

Terminologies of Spring JDBC API

CONNECTION POOLING

When a connection is opened and closed (when required) in an application, object utilization becomes difficult and the cost of implementation increases. In such cases, connection pooling is used.

It is a mechanism of pre-creating a group of database connections and keeping them in cache memory for use and reuse. It provides high performance and efficient resource management.

Terminologies of Spring JDBC API

java.sql.DataSource

It is an interface that is an object-oriented representation of connection pooling. It a connection factory for Java application. It is an alternative for DriverManager and provides Database connections to the Java application.

Its implementation classes are provided by the following connection pooling mechanisms:

1. >Apache BasicDatasource
2. >Spring DriverManagerDatasource
3. >C3p0ComboPoolDatasource

Spring JDBC Template

Spring JDBC provides a class called “JdbcTemplate” to handle the common logic.

JdbcTemplate is the abstraction of JDBC technology.

Methods provided by JdbcTemplate

1. For insert(), update(), and delete() methods of JDBC, Spring JdbcTemplate provides update() method

2. For findByAccno(), findByBalance(), findAll(), rowCount() methods of JDBC, Spring Template provides query(), queryForInt(), queryObject(), queryForMap(), and queryForList()

3. To handle batch updates, JdbcTemplate has convenient methods

Spring JDBC Template Methods

`public int update (String query)`	public int update (String query)
`public int update (String query)`	inserts, updates, and deletes records using PreparedStatement using given arguments
`public void execute (String query)`	executes DDL query
`public T execute (String sql, PreparedStatementCallback action)`	executes the query by using PreparedStatement callback
`public T query (String sql, ResultSetExtractorrse)`	fetches records using ResultSetExtractor
`public List query (String sql, RowMapper rse)`	fetches records using RowMapper

Advanced Java

Topic 2—Spring JDBC Implementation in an Application

Implementing Spring JDBC in an Application

1. Create database and table in the database
2. Create Java project in eclipse
3. Create Spring Bean class (POJO class)
4. Create a DAO class
5. Configure DataSource and Spring Bean spring configuration XML file
6. Create Spring container either by XmlBeanFactory or ApplicationContext

Implementing Spring JDBC in an Application

- Create database and table in the database
- Create Java project in eclipse
- Create Spring Bean class (POJO class)
- Create a DAO class
- Configure DataSource and Spring Bean spring configuration XML file
- Create Spring container either by XmlBeanFactory or ApplicationContext

CREATE DATABASE AND TABLE IN THE DATABASE

Syntax to create database: `create database databasename;`

Syntax to create table:

```
CREATE TABLE table_name (
    column1 datatype,
    column2 datatype,
    column3 datatype,
   ....
);
```

Implementing Spring JDBC in an Application

- Create database and table in the database
- Create Java project in eclipse
- Create Spring Bean class (POJO class)
- Create a DAO class
- Configure DataSource and Spring Bean spring configuration XML file
- Create Spring container either by XmlBeanFactory or ApplicationContext

CREATE JAVA PROJECT IN ECLIPSE

We have already discussed the steps to create Java project in eclipse in previous lessons. Please refer to them.

Implementing Spring JDBC in an Application

- Create database and table in the database
- Create Java project in eclipse
- Create Spring Bean class (POJO class)
- Create a DAO class
- Configure DataSource and Spring Bean spring configuration XML file
- Create Spring container either by XmlBeanFactory or ApplicationContext

CREATE SPRING BEAN CLASS (POJO CLASS)

Consider a user class that has three member variables: id, name, and salary.

```
public class User {
private int id;
private String name;
private float salary;
//no-arg and parameterized constructors
//getters and setters
}
```

Implementing Spring JDBC in an Application

- Create database and table in the database
- Create Java project in eclipse
- Create Spring Bean class (POJO class)
- Create a DAO class
- Configure DataSource and Spring Bean spring configuration XML file
- Create Spring container either by XmlBeanFactory or ApplicationContext

CREATE A DAO CLASS

```
public class UserDao {
private JdbcTemplate jdbcTemplate;

public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
    this.jdbcTemplate = jdbcTemplate;
}

public int saveUser(User e){
    String query="insert into user values(
    '"+e.getId()+"','"+e.getName()+"','"+e.getSalary()+"')";
    return jdbcTemplate.update(query);
}
public int updateUser(User e){
    String query="update user set
    name='"+e.getName()+"',salary='"+e.getSalary()+"' where id='"+e.getId()+"' ";
    return jdbcTemplate.update(query);
}
public int deleteUser(User e){
    String query="delete from user where id='"+e.getId()+"' ";
    return jdbcTemplate.update(query);
}
```

Implementing Spring JDBC in an Application

- Create database and table in the database
- Create Java project in eclipse
- Create Spring Bean class (POJO class)
- Create a DAO class
- Configure DataSource and Spring Bean spring configuration XML file
- Create Spring container either by XmlBeanFactory or ApplicationContext

CONFIGURE DATASOURCE AND SPRING BEAN SPRING CONFIGURATION XML FILE

JDBC driver can be used to establish connectivity to the database during development or basic unit or integration testing.

Spring has two data source classes:

1. DriverManagerDataSource: Returns a new connection every time a connection is required
2. SingleConnectionDataSource: Returns the same connection every time a connection is required

Implementing Spring JDBC in an Application

- Create database and table in the database
- Create Java project in eclipse
- Create Spring Bean class (POJO class)
- Create a DAO class
- Configure DataSource and Spring Bean spring configuration XML file
- Create Spring container either by XmlBeanFactory or ApplicationContext

CONFIGURE DATASOURCE AND SPRING BEAN SPRING CONFIGURATION XML FILE

```
<bean id="ds" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
<property name="driverClassName" value="com.mysql.jdbc.Driver" />
<property name="url" value="jdbc:mysql:3306//localhost/databasename" />
<property name="username" value="username" />
<property name="password" value="password" />
</bean>

<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
<property name="dataSource" ref="ds"></property>
</bean>

<bean id="edao" class="UserDao">
<property name="jdbcTemplate" ref="jdbcTemplate"></property>
</bean>
```

Configure DriverManagerDataSource, JdbcTemplate, and UserDAO class in applicationContext.xml file.

Implementing Spring JDBC in an Application

- Create database and table in the database
- Create Java project in eclipse
- Create Spring Bean class (POJO class)
- Create a DAO class
- Configure DataSource and Spring Bean spring configuration XML file
- Create Spring container either by XmlBeanFactory or ApplicationContext

CREATE SPRING CONTAINER EITHER BY XmlBeanFactory OR ApplicationContext

To run the application, a Spring container based on ApplicationContext is used:

```
public class Test {

public static void main(String[] args) {
    ApplicationContext ctx=new
ClassPathXmlApplicationContext("applicationContext.xml");

    UserDao dao=(UserDao)ctx.getBean("edao");
    int status=dao.saveUser(new User(101,"John",350000));
    System.out.println(status);



}
```

Advanced Java

Topic 3—Integrating Spring with Hibernate

Why Integrate Spring with Hibernate?

Spring Framework provides integration with Hibernate, Toplink, Ibatis, and JPA.

Reasons to integrate Spring with Hibernate:

- In large applications, this integration makes the maintenance very easy.
- When integrated, caching and other performance tuning is handled automatically.
- Database can be changed easily without affecting the application code.
- Integrating hibernate application with spring avoids creating the hibernate.cfg.xml file. All information can be contained in the applicationContext.xml file.

Integrating Spring with Hibernate

1. Configure application.xml.
2. Configure session factory using the class AnnotationSessionFactoryBean.
3. Define HibernateTemplate by passing session factory.
4. For the database transaction, define HibernateTransactionManager. Using the transaction manager, configure AOP.

HibernateTemplate Class

HibernateTemplate is a helper class that simplifies Hibernate data access code.

It automatically converts HibernateExceptions into DataAccessExceptions by following the org.springframework.dao exception hierarchy.

Structure of HibernateTemplate class:

```
org.springframework.orm.hibernate3
public class HibernateTemplate
extends HibernateAccessor
implements HibernateOperations
```

Methods of HibernateTemplate Class

Method	Description
`void persist(Object entity)`	Persists the given object.
`Serializable save(Object entity)`	Persists the given object and returns id.
`void saveOrUpdate(Object entity)`	Persists or updates the given object. If id is found, it updates the record; otherwise, it saves the record.
`void update(Object entity)`	Updates the given object
`void delete(Object entity)`	Deletes the given object on the basis of id.
`Object get(Class entityClass, Serializable id)`	Returns the persistent object on the basis of given id.
`Object load(Class entityClass, Serializable id)`	Returns the persistent object on the basis of given id.
`List loadAll(Class entityClass)`	Returns all the persistent objects.

Advanced Java

Topic 4—Creating Application using Spring Hibernate Template

Creating Application using Spring Hibernate Template

1. Create database.
2. Create Java project in eclipse.
3. Create User.java file. It is the persistent class.
4. Create UserDao.java file. It is the DAO class that uses HibernateTemplate.
5. Create applicationContext.xml file. It contains information of DataSource, SessionFactory, etc.
6. Create Test.Java file. It calls methods of UserDao class.

Creating Application using Spring Hibernate Template

CREATE DATABASE

- Create database
- Create Java project in eclipse
- Create User.java file
- Create UserDao.java file
- Create applicationContext.xml file
- Create Test.Java file

Syntax to create database: `create database databasename;`

Creating Application using Spring Hibernate Template

CREATE JAVA PROJECT IN ECLIPSE

- Create database
- Create Java project in eclipse
- Create User.java file
- Create UserDao.java file
- Create applicationContext.xml file
- Create Test.Java file

We have already discussed the steps to create Java project in eclipse in previous lessons. Please refer to them.

Creating Application using Spring Hibernate Template

CREATE User.java FILE

- Create database
- Create Java project in eclipse
- Create User.java file
- Create UserDao.java file
- Create applicationContext.xml file
- Create Test.Java file

User.java is an annotated POJO that represents User instance.

In the User class, there is an ID and Name of user. @Id makes column the primarykey,

```
@Entity
public class User implements Serializable {
  private static final long serialVersionUID = 1L;
  @Id
  private int id;
  private String name;
// constructor
// public setter and getter methods
}
```

Creating Application using Spring Hibernate Template

CREATE UserDao.java FILE

- Create database
- Create Java project in eclipse
- Create User.java file
- Create UserDao.java file
- Create applicationContext.xml file
- Create Test.Java file

This method performs the task of saving data in Database.

```
public class PageDao {
      private HibernateTemplate hibernateTemplate;

      public void setHibernateTemplate(HibernateTemplate hibernateTemplate) {
            this.hibernateTemplate = hibernateTemplate;
      }

      public void persist(){
        User u1= new User(1,"Ankita");
        hibernateTemplate.save(u1);

        User u2= new User(2,"Renu");
        hibernateTemplate.save(u2);
    }
}
```

In this example, MySQl database is used. The instance of HibernateTemplate is fetched by spring injection and saved to database using HibernateTemplate.
Inside persist method, there are two User Objects.

Creating Application using Spring Hibernate Template

CREATE applicationContext.xml FILE

- Create database
- Create Java project in eclipse
- Create User.java file
- Create UserDao.java file
- Create applicationContext.xml file
- Create Test.Java file

application.xml contains all the declarations of DAOs and other transactional configurations. To inject HibernateTemplate into DAO classes, a bean org.springframework.orm.hibernate3.HibernateTemplate has been created.

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
        <property name="driverClassName"  value="com.mysql.jdbc.Driver"></property>
        <property name="url" value="jdbc:mysql://localhost:3306/database"></property>
        <property name="username" value="username"></property>
        <property name="password" value="password"></property>
    </bean>
    <bean id="mysessionFactory"
class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
        <property name="dataSource" ref="dataSource"></property>
          <property name="mappingResources">
        <list>
        <value>employee.hbm.xml</value>
        </list>
        </property>
            <property name="hibernateProperties">
            <props>
                <prop key="hibernate.dialect">org.hibernate.dialect.MySQLDialect</prop>
                <prop key="hibernate.hbm2ddl.auto">update</prop>
                <prop key="hibernate.show_sql">true</prop>
                 </props>
```

Creating Application using Spring Hibernate Template

CREATE applicationContext.xml FILE

- Create database
- Create Java project in eclipse
- Create User.java file
- Create UserDao.java file
- Create applicationContext.xml file
- Create Test.Java file

Code continuation

```
</property>
    </bean>
       <bean id="template"
class="org.springframework.orm.hibernate3.HibernateTemplate">
    <property name="sessionFactory" ref="mysessionFactory"></property>
    </bean>
      <bean id="d" class="com.javatpoint.EmployeeDao">
    <property name="template" ref="template"></property>
    </bean>

    </beans>
```

Creating Application using Spring Hibernate Template

CREATE Test.Java FILE

- Create database
- Create Java project in eclipse
- Create User.java file
- Create UserDao.java file
- Create applicationContext.xml file
- Create Test.Java file

Find the class SpringDemo.java. SpringDemo calls the persist method from the PageDao to save the data in database. Run the command.

```
public class SpringDemo {
    public static void main(String... args) {
            ApplicationContext context = new
ClassPathXmlApplicationContext("application.xml");
            PageDao pd=(PageDao) context.getBean("pageDao");
            pd.persist();
    }
}
```

Output:

```
Hibernate: insert into User (name, id) values (?, ?)
Hibernate: insert into User (name, id) values (?, ?)
```

Advanced Java

Topic 5—Spring JDBC Transaction Management

Spring JDBC Transaction Management

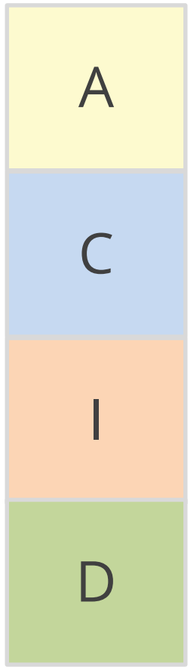
Spring provides extensive support for transaction management. This helps developers to focus on business logic rather than worrying about the integrity of data in case of any system failures.

Spring provides an abstract layer on top of different transaction management APIs.

It helps application developers to focus on the business problem, without having to know much about the underlying transaction management APIs.

ACID

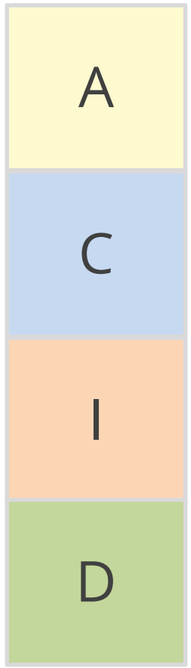
The concept of transactions can be described using the following four key properties, known as ACID.



Atomicity requires that each transaction be ‘all or nothing’. If one part of the transaction fails, then the entire transaction fails, and the database state is left unchanged.

ACID

The concept of transactions can be described using the following four key properties, known as ACID.



Atomicity requires that each transaction be ‘all or nothing’. If one part of the transaction fails, then the entire transaction fails, and the database state is left unchanged.

Consistency property ensures that any transaction will bring the database from one valid state to another.

ACID

The concept of transactions can be described using the following four key properties, known as ACID.

A	**Atomicity** requires that each transaction be ‘all or nothing’. If one part of the transaction fails, then the entire transaction fails, and the database state is left unchanged.
C	**Consistency** property ensures that any transaction will bring the database from one valid state to another.
I	**Isolation** property ensures that the concurrent execution of transactions results in a system state that would be obtained if transactions were executed serially, that is, one after the other.
D	

ACID

The concept of transactions can be described using the following four key properties, known as ACID.

A	**Atomicity** requires that each transaction be ‘all or nothing’. If one part of the transaction fails, then the entire transaction fails, and the database state is left unchanged.
C	**Consistency** property ensures that any transaction will bring the database from one valid state to another.
I	**Isolation** property ensures that the concurrent execution of transactions result in a system state that would be obtained if transactions were executed serially, that is, one after the other.
D	**Durability** property ensures that once a transaction has been committed, it will remain so, even in the event of power loss, crashes, or errors.

Transaction Management Interface

PlatformTransactionManager is a general interface for all Spring transaction managers:

```
public interface PlatformTransactionManager {
   TransactionStatus getTransaction(TransactionDefinition definition);
   throws TransactionException;

   void commit(TransactionStatus status) throws TransactionException;
   void rollback(TransactionStatus status) throws TransactionException;
}
```

Spring has several built-in implementations of this interface that can be used with different transaction management APIs like DataSourceTransactionManager, HibernateTransactionManager, and JpaTransactionManager

Methods of PlatformTransactionManage

Method	Description
`TransactionStatus getTransaction(TransactionDefinition definition)`	This method returns a currently active transaction or creates a new one, according to the specified propagation behavior.
`void commit(TransactionStatus status)`	This method commits the given transaction, with regard to its status.
`void rollback(TransactionStatus status)`	This method performs a rollback of the given transaction.

TransactionDefinition Interface

TransactionDefinition is the core interface of the transaction support in Spring, and it is defined as follows:

```
public interface TransactionDefinition
{
   int getPropagationBehavior();
   int getIsolationLevel();
   String getName();
   int getTimeout();
   boolean isReadOnly();
}
```

Methods of TransactionDefinition

Method	Description
`int getPropagationBehavior()`	This method returns the propagation behavior. Spring offers all of the transaction propagation options offered by EJB CMT.
`int getIsolationLevel()`	This method returns the degree to which this transaction is isolated from the work of other transactions.
`String getName()`	This method returns the name of the transaction.
`int getTimeout()`	This method returns the time (in seconds) in which the transaction must complete.
`boolean isReadOnly()`	This method queries whether the transaction is read only.

TransactionStatus Interface

TransactionStatus interface provides a simple way for transactional code to control transaction execution and query transaction status.

```
public interface TransactionStatus extends SavepointManager {
   boolean isNewTransaction();
   boolean hasSavepoint();
   void setRollbackOnly();
   boolean isRollbackOnly();
   boolean isCompleted();
}
```

Methods of TransactionStatus

Method	Description
`boolean hasSavepoint()`	This method returns whether this transaction internally carries a savepoint, i.e., has been created as nested transaction based on a savepoint.
`boolean isCompleted()`	This method returns whether this transaction is completed, i.e., whether it has already been committed or rolled back.
`boolean isNewTransaction()`	This method returns true in case the present transaction is new.
`boolean isRollbackOnly()`	This method returns whether the transaction has been marked as rollback-only.
`void setRollbackOnly()`	This method sets the transaction as rollback-only.

Types of Transaction Management

Spring supports two types of transaction management:

1. Programmatic transaction management
2. Declarative transaction management

Types of Transaction Management

PROGRAMMATIC TRANSACTION MANAGEMENT

Programmatic transaction management

Declarative transaction management

- It manages transactions with the help of programming.
- It uses PlatformTransactionManager directly to implement the programmatic approach, to implement transactions, and to create instance of TransactionDefinition with the appropriate transaction attributes.
- Once the TransactionDefinition is created, you can start your transaction by calling getTransaction() method, which returns an instance of TransactionStatus.
- The TransactionStatus objects helps in tracking the current status of the transaction.
- Programmatic Transaction Management uses commit() method of PlatformTransactionManager to commit the transaction.

Types of Transaction Management

DECLARATIVE TRANSACTION MANAGEMENT

Programmatic transaction management
Declarative transaction management

- Separates transaction management from the business code.
- Allows you to manage the transaction with the help of configuration instead of hard coding in your source code.
- Uses annotations or XML-based configurations to manage the transactions.
- Uses <tx:advice /> tag. This creates a transaction-handling advice and defines a Pointcut that matches all methods needed for the transaction at the same time.

Key Takeaways

- Spring JDBC is used to build Data Access Layer of enterprise application. It is a mechanism to connect to the database and execute SQL queries. The database-related logic has to placed in separate classes called DAO classes.
- Connection pooling is a mechanism of pre-creating a group of database connections and keeping them in cache memory for use and reuse. It provides high performance and efficient resource management.
- Spring supports two types of transaction management: Programmatic transaction management and Declarative transaction management

Quiz

Thank You