Deloitte.

Spring Basics - II

USI CBO CR LAUNCHPAD TRAINING PROGRAM

Spring Basics – II

Context, Objectives, Agenda

Context

- The Spring framework comprises of many modules such as core, beans, context, expression language, AOP, Aspects, Instrumentation, JDBC, ORM, OXM, JMS, Transaction, Web, Servlet, Struts etc.

Objectives

At the end of the session, you will be able to –

- Identify cross-cutting concerns.
- Identify Pointcut and JoinPoints
- Create Advice
- Understanding practical implementation of Spring Concepts with examples.
- Gain implementation knowledge

Agenda

Topic	Content
AOP	• AOP Solution & Use Cases • Terminology • Spring AOP Example • Types of AOP Advices • Hands on exercises • Pointcut Expressions & Examples • Aspects - Ordering • Examples - Demo
Spring Concepts	• Aliasing • Lazy Initialization (LI) • Spring Bean – Inheritance
Spring: JDBC Template & ORM	• Introduction • Features and Benefits

Aspect Oriented Programming (AOP)

Aspect Oriented Programming (AOP)

Objectives

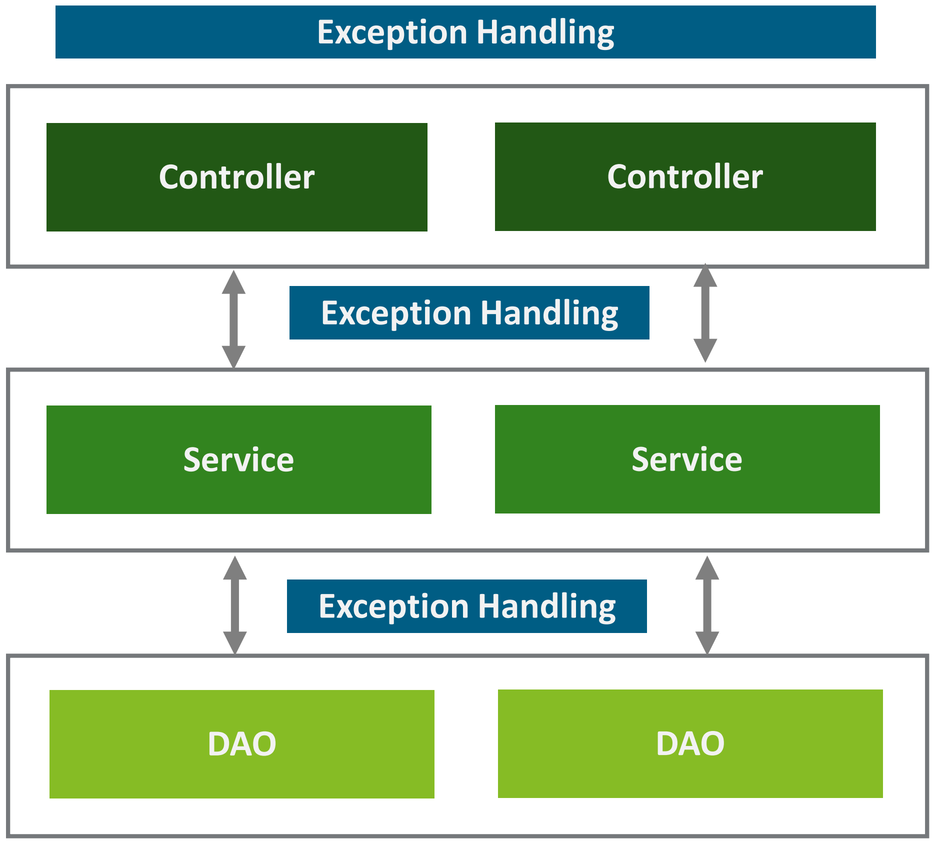
AOP

Why AOP ?

Problem without AOP

- If you have a system that contains several packages and classes, such as tracing, transactions, and exception handling, we have to implement them in every class and every method.
- This results in two problems :
  - **Code tangling** - Each class and method contains tracing, transactions, and exception handling — even business logic. In a tangled code, it is often hard to see what is actually going on in a method.
  - **Code scattering** - Aspects such as transactions are scattered throughout the code and not implemented in a single specific part of the system.
- Using AOP allows you to solve these problems. So, what AOP does is it takes all the transaction code and puts it into a transaction aspect. Then, it takes all the tracing code and puts that into another aspect. Finally, exception handling is also put into a separate aspect.
- After using AOP, there will be a clean separation between the business logic and all additional aspects.
- AOP provides the pluggable way to dynamically add the additional concern before, after or around the actual logic

Exception handling
AOP Applied

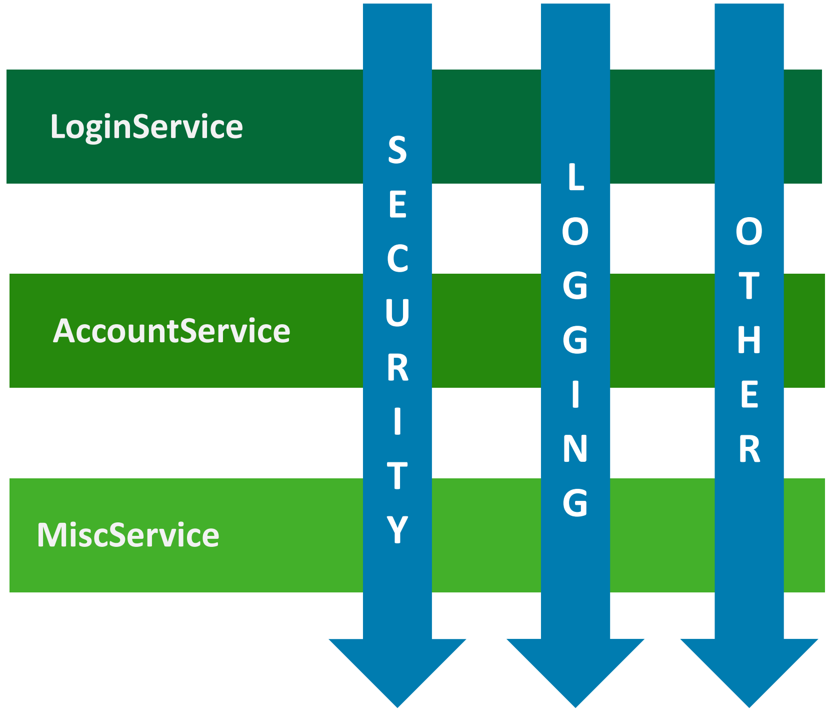


AOP

AOP Solution & Use Cases

AOP Solution

- ➢ Aspect Oriented Programming entails breaking down program logic into distinct parts called concerns. The functions that span multiple points of an application are called cross-cutting concerns. These cross-cutting concerns are used to increase modularity and are conceptually separate from the application's business logic.
- ➢ The key unit of modularity in OOP is the class, whereas in AOP the unit of modularity is the aspect.
- ➢ It Can be used in Cross cutting layer functionality like:
  - Security
  - Transactional
  - Logging
  - Monitoring
  - Cache
- ➢ The **Spring Framework** recommends you to use **Spring AspectJ AOP implementation** because it provides you more control and it is easy to use. There are two ways to use Spring AOP AspectJ implementation -
  - By annotation
  - By xml configuration



AOP

AOP Terminology

Aspect

- A module which has a set of APIs providing cross-cutting requirements. It is a class that contains advices, joinpoints etc.
- An application can have any number of aspects depending on the requirement.
- For example, a logging module would be called AOP aspect for logging.

Join point

- This represents a point in your application where you can plug-in AOP aspect.
- You can also say, it is any point in your program such as method execution, exception handling, field access etc.

Advice

- Advice is the actual action to be taken either before or after the method execution. This is the actual piece of code that is invoked during program execution by Spring AOP framework

PointCut

- This is a set of one or more joinpoints where an advice should be executed. You can specify Pointcuts using expressions or patterns as we will see in our AOP examples

Introduction

- It means introduction of additional method and fields for a type. It allows you to introduce new interface to any advised object.

Target object

- It is the object being advised by one or more aspects. This object will always be a proxied object. Also referred to as the advised object.

Weaving

- Weaving is the process of linking aspects with other application types or objects to create an advised object. This can be done at compile time, load time, or at runtime

AOP

Types of AOP Advices

Types of Advice

01 **before**
Run advice before the method execution

02 **after**
Run advice after the method execution, regardless of its outcome.

03 **after-returning**
Run advice after the method execution, only if the method completes successfully.

04 **after-throwing**
Run advice after the method execution, only if the method exits by throwing an exception.

05 **around**
Run advice before and after the advised method is invoked.

AOP
Example

Step 1

- Import Spring AOP dependencies into your project.

pom.xml

```xml
<dependency>
     <groupId>org.springframework</groupId>
     <artifactId>spring-aop</artifactId>
     <version>5.2.7.RELEASE</version>
</dependency>
<dependency>
     <groupId>org.aspectj</groupId>
     <artifactId>aspectjrt</artifactId>
     <version>1.9.5</version>
</dependency>
<dependency>
     <groupId>org.aspectj</groupId>
     <artifactId>aspectjweaver</artifactId>
     <version>1.9.5</version>
</dependency>
```

Step 2

- Enable AOP configuration in Spring applications (Weaving).

AopConfig.java

```java
@Configuration
@EnableAspectJAutoProxy
public class AopConfig {


}
```

AOP

Example (Contd...)

Step 3

- Write aspect class annotated with @Aspect annotation and write point-cut expressions to match joint-point methods

EmployeeCRUDAspect.java

```
@Aspect
public class EmployeeCRUDAspect {

    @Before("execution(* 
EmployeeManager.getEmployeeById(..))")        //point-cut 
expression
    public void logBefore(JoinPoint joinPoint)
    {
        System.out.println("Employee CRUDAspect log 
BeforeAspect : " + joinPoint.getSignature().getName());
    }
}
```

Step 4

- Write methods on which you want to execute advices and those match with point-cut expressions.

EmployeeManager.java

```
@Component
public class EmployeeManager
{
    public EmployeeDTO getEmployeeById(Integer 
employeeId) {
        System.out.println("Method getEmployeeById() 
called");
        return new EmployeeDTO();
    }
}
```

AOP

Example

(Contd…)

Step 5

- Run the application and watch the console.

TestAOP.java

```
public class TestAOP
{
    @SuppressWarnings("resource")
    public static void main(String[] args) {

        ApplicationContext context = new ClassPathXmlApplicationContext
                    ("com/demo/aop/applicationContext.xml");

        EmployeeManager manager = context.getBean(EmployeeManager.class);

        manager.getEmployeeById(1);
    }
}
```

Program output:

- In the example, logBefore() will be executed before getEmployeeById() method because it matches the join-point expression.

Console Output

```
EmployeeCRUDAspect.logBefore() : getEmployeeById
Method getEmployeeById() called
```

Knowledge Check

Check what will be the output when you use other Advice instead of @before advice.

Break – 15 min.

AOP

Pointcut Expressions

- Pointcut is an **expression language** of Spring AOP. The most typical pointcut expressions are used to match a number of methods by their **signatures**.
- The **@Pointcut** annotation is used to define the pointcut.
- Lets see some **examples** which will show how to write pointcut expressions to match any kind of method joint points into your spring application.

Pointcut usage	Pointcut Expression
✓ Match all methods within a class in another package	• execution(* com.java.AttendanceManager.*(..))
✓ Match all methods within a class within same package	• execution(* AttendanceManager.*(..))
✓ Match all public methods in class AttendanceManager	• execution(public * AttendanceManager.*(..))
✓ Match all public methods in AttendanceManager with return type AttendanceDTO	• execution(public AttendanceDTO AttendanceManager.*(..))
✓ Match all public methods in AttendanceManager with return type AttendanceDTO and first parameter as AttendanceDTO	• execution(publicAttendanceDTO AttendanceManager.* (AttendanceDTO, ..))
✓ Match all public methods in AttendanceManager with return type AttendanceDTO and definite parameters	• execution(public AttendanceDTO AttendanceManager.*(AttendanceDTO, Integer))

AOP

Pointcut Expressions

- Spring AOP only supports method execution join points for Spring beans.
- Some of the listed pointcut expressions only those, are matching the execution of methods on Spring beans.
- Spring supports operations on pointcuts – Union and Intersection. Union is usually more useful.

- A pointcut declaration has four parts as below:
  - Matching Method Signature Patterns
  - Matching Type Signature Patterns
  - Matching Bean Name Patterns
  - Combining Pointcut Expressions

Designator	Comments
execution	• pointcut expression for matching method execution join points
within	• pointcut expression for matching to join points within certain types
this	• pointcut expression for matching to join points where the bean reference is an instance of the given type
target	• pointcut expression for matching to join points where the target object is an instance of the given type
args	• pointcut expression for matching to join points where the arguments are instances of the given types

Designator	Comments
@target	• pointcut expression for matching to join points where the class of the executing object has an annotation of the given type
@args	• pointcut expression for matching to join points where the runtime type of the actual arguments passed have annotations of the given type
@within	• pointcut expression for matching to join points within types that have the given annotation
@annotation	• pointcut expression for matching to join points where the subject of the join point has the given annotation

AOP

Aspects Ordering

- Suppose you have multiple aspects (Logging, Security and Transaction Aspect) in your application and they are can be applied on a certain method.
- When there's more than one aspect applied to the same join point, the precedence/order of the aspects will not be determined unless you have explicitly specified it using either **@Order** annotation or **org.springframework.core.Ordered** interface.

Specifying aspects ordering using @Order annotation

Example :

```
@Aspect
@Component
@Order(0)
public class SecurityAspect {
 @Before("execution(*
com.aopapp.service.*.transfer(*,*,*))")
 public void beforeAdviceForTransferMethods(JoinPoint jp)
throws Throwable {
           System.out.println("****SecurityAspect.beforeAd
viceForTransferMethods() " + jp.getSignature().getName());
    }
}
```

Example :

```
@Aspect
@Component
@Order(1)
public class TransactionAspect {
 @Before("execution(* com.aopapp.service.*.transfer(*,*,*))")
 public void beforeAdviceForTransferMethods(JoinPoint jp) throws
Throwable {
           System.out.println("****TransactionAspect.beforeAdvice
ForTransferMethods() " + jp.getSignature().getName());
    }
}
```

AOP

Aspects Ordering

Specifying aspects ordering by using Ordered interface

Example :

```
  @Aspect
@Component
public class SecurityAspect implements Ordered{
 @Before("execution(* com.aopapp.service.*.transfer(*,*,*))")
 public void beforeAdviceForTransferMethods(JoinPoint jp) throws
Throwable {

System.out.println("****SecurityAspect.beforeAdviceForTransferMet
hods() " + jp.getSignature().getName());
    }

 @Override
 public int getOrder() {
  return 0;
 }
```

Example :

```
@Aspect
@Component
public class TransactionAspect implements Ordered{
 @Before("execution(*
com.aopapp.service.*.transfer(*,*,*))")
 public void beforeAdviceForTransferMethods(JoinPoint jp)
throws Throwable {

System.out.println("****TransactionAspect.beforeAdviceForTra
nsferMethods() " + jp.getSignature().getName());
    }

 @Override
 public int getOrder() {
  return 1;
 }
}
```

AOP

Demo

Operation.java

```
package com.java;
public  class
Operation{

public void msg(){
System.out.println("ms
g method invoked");}

public int
m(){System.out.println
("m method
invoked");return 2;}

public int
k(){System.out.println
("k method
invoked");return 3;}
}
```

TrackOperation.java

```
@Aspect
public class
TrackOperation{
    @Pointcut("execution(*
Operation.*(..))")
    public void
k(){}//pointcut name


@Before("k()")//applying
pointcut on before advice
    public void
myadvice(JoinPoint jp)//it
is advice (before advice)
    {

System.out.println("additi
onal concern");

//System.out.println("Meth
od Signature: "  +
jp.getSignature());
    }
}
```

applicationContext.xml

```
<bean id="opBean"
class="com.java.Operat
ion">   </bean>

 <bean
id="trackMyBean"
class="com.java.TrackO
peration"></bean>

<bean
class="org.springframe
work.aop.aspectj.annot
ation.AnnotationAwareA
spectJAutoProxyCreator
"></bean>
```

Test.java

```
public class Test{
    public static void
main(String[] args){
        ApplicationContext
context = new
ClassPathXmlApplicationCon
text("applicationContext.x
ml");
        Operation e =
(Operation)
context.getBean("opBean");

System.out.println("callin
g msg...");
        e.msg();

System.out.println("callin
g m...");
        e.m();

System.out.println("callin
g k...");
        e.k();
    }
}
```

Output

```
calling msg...
additional concern
msg() method invoked
calling m...
additional concern
m() method invoked
calling k...
additional concern
k() method invoked
```

additional concern is printed before msg(), m() and k() method is invoked

Knowledge Check

Which advice do you have to use if you would like to try and catch exceptions?

- ○ After
- ○ AfterThrowing
- ○ AfterReturning
- ○ Around

A module that encapsulates pointcuts and advice

- ○ Aspect
- ○ Join Point
- ○ Weaving
- ○ Pointcut

Identify the Joinpoint visibility ? @Pointcut("execution(* *(..))")

- ○ All methods, except private method
- ○ All Public methods
- ○ All Private methods
- ○ All method within same package

Identify the Joinpoint visibility ? execution(* Controller.*(..))

- ○ All methods, except private method
- ○ All Public methods
- ○ All Private methods
- ○ All method within same package

Recap

AOP

Glimpse of Important points

AOP compliments OOPs in the sense that it also provides modularity.

Advice Type

- before
- after
- after-returning
- after-throwing
- around

JoinPoint represents a point in your application where you can plug-in AOP aspect..

PointCut is a set of one or more JoinPoints where an advice should be executed. You can specify PointCuts using expressions or patterns.

When there's more than one aspect applied to the same join point, the precedence/order of the aspects will not be determined unless you have explicitly specified it .

Lunch Break – 45 min.

Spring Concepts

Spring Concepts

Objectives

01 **Aliasing**

02 **Lazy Initialization (LI)**

03 **Spring Bean – Inheritance**

Spring Concepts

Aliasing

- In a bean definition itself, you can supply more than one name for the bean, by using a combination of up to one name specified by the id attribute, and any number of other names in the name attribute.
- These names can be equivalent aliases to the same bean, and are useful for some situations, such as allowing each component in an application to refer to a common dependency by using a bean name that is specific to that component itself.
- In short, Spring also provides an alias to bean. Hence we can also access a bean by bean alias name
- To define a bean alias name, spring provides a **<alias>** tag :
- Syntax:

<alias name = “bean-name” alias = “bean-alias-name”/>

- **Use Case** : This is commonly the case in large systems where configuration is split amongst each subsystem, each subsystem having its own set of object definitions.

Aliasing Bean

Example :

For example, the configuration metadata for subsystem A may refer to a DataSource via the name **'subsystemA-dataSource**. The configuration metadata for subsystem B may refer to a DataSource via the name **'subsystemB-dataSource**'. When composing the main application that uses both these subsystems the main application refers to the DataSource via the name 'myApp-dataSource'. To have all three names refer to the same object you add to the MyApp configuration metadata the following aliases definitions:

Code :

```
<alias name="subsystemA-dataSource"
       alias="subsystemB-dataSource"/>
<alias name="subsystemA-dataSource"
       alias="myApp-dataSource" />
```

- Now each component and the main application can refer to the dataSource through a name that is unique and guaranteed not to clash with any other definition (effectively creating a namespace), yet they refer to the same bean

Spring Concepts

Lazy Initialization

- By default, Spring “application context” eagerly creates and initializes all ‘singleton scoped‘ beans during application startup itself. It helps in detecting the bean configuration issues at early stage, in most of the cases. But sometimes, you may need to mark some or all beans to be lazy initialized due to different project requirements.
- Spring provides two easy ways to configure lazy initialization of beans based on which kind of configuration you are employing i.e. XML based configuration or java based configuration

Lazy initialized beans in XML configuration

Lazy load specific beans only

```
<beans>
<bean id="employeeManager"
class="com.java.spring.service.impl.EmployeeManagerImpl"
    lazy-init="true"/>
<beans>
```

Lazy load all beans globally

```
<beans default-lazy-init="true">
<bean id="employeeManager"
class="com.java.spring.service.impl.EmployeeManagerImpl"
/>
<beans>
```

Spring Concepts

Lazy Initialization

(Contd…)

Lazy initialized beans in XML configuration

Lazy load specific bean

```
import
org.springframework.context.annotation.Lazy;

@Configuration
public class AppConfig {

    @Lazy
    @Bean
    public EmployeeManager employeeManager()
{
        return new EmployeeManagerImpl();
    }

}
```

Lazy load all beans with @Lazy annotation

```
import
org.springframework.context.annotation.Lazy;

@Lazy
@Configuration
public class AppConfig {

    @Bean
    public EmployeeManager employeeManager() {
        return new EmployeeManagerImpl();
    }

}
```

@Autowired lazy beans

```
@Lazy
@Service
public class EmployeeManagerImpl
implements EmployeeManager {
  //
}

@Controller
public class EmployeeController {

    @Lazy
    @Autowired
    EmployeeManager employeeManager;
}
```

Note : Without using @Lazy annotation at both places, it will not work.

Spring Concepts

Spring Bean – Inheritance

- In Spring, the inheritance is supported in bean configuration for a bean to share common values, properties or configurations.

A child bean or inherited bean can inherit its parent bean configurations, properties and some attributes. In additional, the child beans are allow to override the inherited value. The Settings that will always be taken from the child definition are depends on autowire mode, dependency check, singleton, scope, lazy init.

Customer Model Class

```java
package com.java.common;
public class Customer
{
private int type;
private String action;
private String Country;
//...
}
```

SpringBeans.xml

```xml
<bean id="BaseCustomerTest"
class="com.java.common.Customer"> <property
name="country" value="India" /> </bean>
<bean id="CustomerBean"
parent="BaseCustomerTest">
<property name="action" value="buy" />
<property name="type" value="1" /> </bean>
```

App.java

```java
package com.java.common;

public class App
{
public static void main( String[] args )
{
  ApplicationContext context = new
ClassPathXmlApplicationContext("SpringBeans.xml
");
Customer cust =
(Customer)context.getBean("CustomerBean");
System.out.println(cust);
             }
 }
```

Output : Customer [type=1, action=buy, Country=India]

The ‘**CustomerBean**’ bean just inherited the country property from its parent (‘**BaseCustomerTest**’)

Spring Concepts

Spring Bean – Inheritance

(Contd…)

1. Inheritance with abstract

If you want to make this base bean as a template and not allow others to instantiate it, you can add an '**abstract**' attribute in the <bean> element

SpringBeans.xml

```
<bean id="BaseCustomerTest"
class="com.java.common.Customer"
abstract="true" >
<property name="country" value="India" />
</bean>
<bean id="CustomerBean"
parent="BaseCustomerTest">
<property name="action" value="buy" />
<property name="type" value="1" /> </bean>
```

Output

```
Customer cust =
(Customer) context.getBean("BaseCustomerTest");
```

Explanation :
The 'BaseCustomerTest' bean is a pure template, for bean to inherit it only, if you try to instantiate it, you will encounter error `org.springframework.beans.factory.BeanIsAbstractException`

2. Overrride

We can override the inherited value by specify the new value in the child bean.

SpringBeans.xml

```
<bean id="BaseCustomerTest"
class="com.java.common.Customer"
abstract="true" >
<property name="country" value="India" />
</bean>
<bean id="CustomerBean"
parent="BaseCustomerTest">
<property name="country" value="USA" />
<property name="action" value="buy" />
<property name="type" value="1" /> </bean>
```

Output

```
Customer [Country=USA, action=buy, type=1]
```

Explanation :
The 'CustomerBean' bean just overrides the parent ('BaseCustomerTest') country property, from 'India' to 'USA'.

Knowledge Check

What help Spring to create bean and inject its dependencies when needed ?

- ○ Lazy Initialization
- ○ Aliasing
- ○ Advice
- ○ Inheritance

What will a child bean definition inherit from Parent definition ?

- ○ constructor argument values
- ○ property values
- ○ method overrides
- ○ All of the Above

What allows us to override already configured beans and to substitute them with different object definition?

- ○ Lazy Initialization
- ○ Aliasing
- ○ Advice
- ○ Inheritance

You cannot override the inherited value in the child bean.

- ○ True
- ○ False
- ○ Maybe

Recap

Spring Concepts

Glimpse of Important points

The bean alias name gives us one more way to access our defined bean in the spring application.

The Spring bean configuration inheritance is very useful to avoid the repeated common value or configurations for multiple beans.

A child bean definition will inherit constructor argument values, property values, and method overrides from the parent. The child bean definition also can add new values.

The Settings that will always be taken from the child definition are depends on, autowire mode, dependency check, singleton, scope, lazy init.

When we configure a bean with lazy initialization, the bean will only be created, and its dependencies injected, once they're needed

Spring: JDBC Template & ORM

Spring: JDBC Template & ORM

Objectives

Spring: JDBC Template & ORM

Introduction

Problems of JDBC API

❑ JDBC produces a lot of boiler plate code, such as opening/closing a connection to a database, handling sql exceptions etc,.

❑ We need to perform exception handling code on the database logic.

❑ Repetition of all these codes from one to another database logic is a time consuming task

❑ We need to handle transaction.

Spring JDBC to the rescue

Spring JDBC eliminates all the above mentioned problems of JDBC API. It provides you methods to write the queries directly, so it saves a lot of work and time.

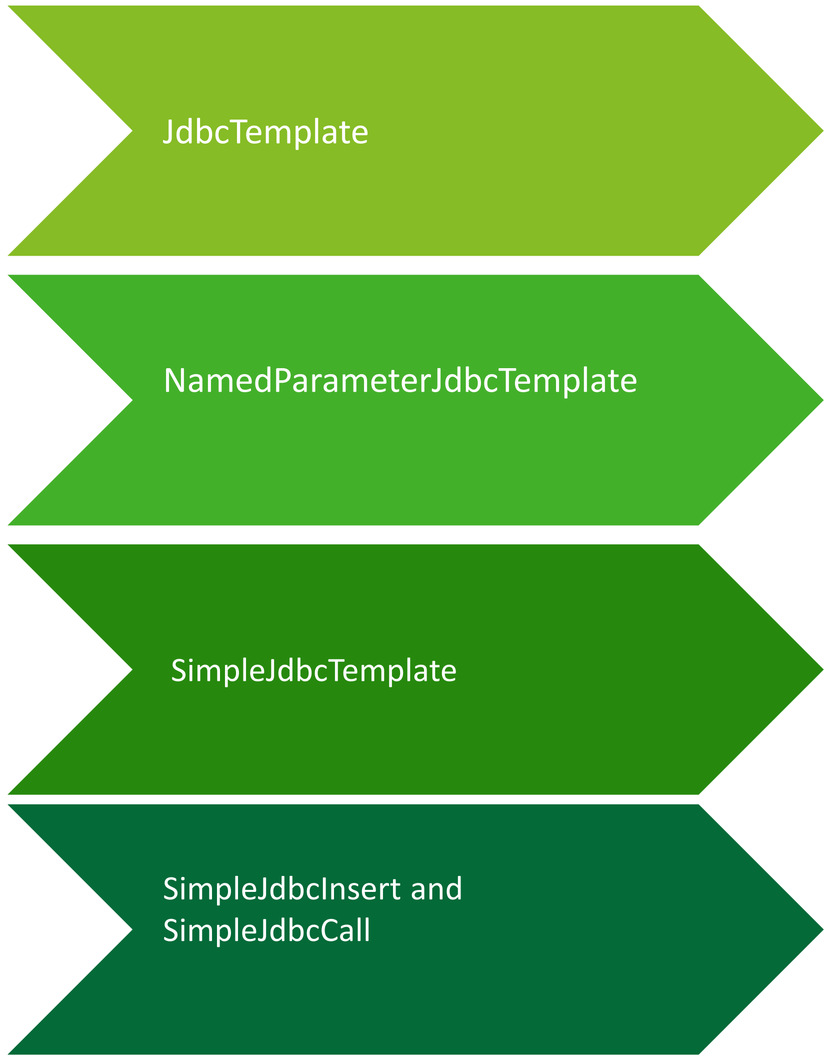
Spring framework provides following approaches for JDBC database access:

❑ JdbcTemplate

❑ NamedParameterJdbcTemplate

❑ SimpleJdbcTemplate

❑ SimpleJdbcInsert and SimpleJdbcCall


JdbcTemplate
NamedParameterJdbcTemplate
SimpleJdbcTemplate
SimpleJdbcInsert and SimpleJdbcCall

Spring: JDBC Template & ORM

JDBC Template Types

JdbcTemplate class

- ➢ It is the central class in the Spring JDBC support classes. It takes care of creation and release of resources such as creating and closing of connection object etc. So it will not lead to any problem if you forget to close the connection.
- ➢ It handles the exception and provides the informative exception messages by the help of excepion classes defined in the org.springframework.dao package..
- ➢ CRUD is data-oriented and the standardized use of HTTP action verbs. HTTP has a few important verbs.
- ➢ Within a database, each of these operations maps directly to a series of commands. However, their relationship with a RESTful API is slightly more complex.

Methods of JdbcTemplate class	Description
• public int update(String query)	➢ It performs the INSERT statement to create a new record.
• public int update(String query,Object... args)	➢ is used to insert, update and delete records using PreparedStatement with given arguments.
• public void execute(String query)	➢ is used to execute DDL query.
• public T execute(String sql, PreparedStatementCallback action)	➢ executes the query by using PreparedStatement callback.
• public T query(String sql, ResultSetExtractor rse)	➢ is used to fetch records using ResultSetExtractor.
• public List query(String sql, RowMapper rse)	➢ is used to fetch records using RowMapper.

Spring: JDBC Template & ORM

JDBC Template Types

PreparedStatement in JdbcTemplate

- We can execute parameterized query using Spring JdbcTemplate by the help of **execute()** method of JdbcTemplate class. To use parameterized query, we pass the instance of **PreparedStatementCallback** in the execute method.

- Syntax of execute method to use parameterized query -

```
public T execute(String sql,PreparedStatementCallback<T>);
```

- Method of PreparedStatementCallback interface

```
public T doInPreparedStatement(PreparedStatement ps)
throws SQLException, DataAccessException
```

Example :

```
String query = "insert into employee values(?,?,?)";

public Boolean doInPreparedStatement(PreparedStatement ps)
        throws SQLException, DataAccessException {

    ps.setInt(1,Id);
    ps.setString(2,Name);
    ps.setFloat(3,Salary);

    return ps.execute();
}
```

```
dao.saveEmployeeByPreparedStatement(new Employee(111,“User",35000));
```

Spring: JDBC Template & ORM

JDBC Template Types

ResultSetExtractor

- The **org.springframework.jdbc.core.ResultSetExtractor** interface is a callback interface used by JdbcTemplate's query methods. Implementations of this interface perform the actual work of extracting results from a ResultSet, but don't need to worry about exception handling.
- SQLExceptions will be caught and handled by the calling JdbcTemplate. This interface is mainly used within the JDBC framework itself.
- We can easily fetch the records from the database using query() method of JdbcTemplate class where we need to pass the instance of ResultSetExtractor.
- Syntax of query method using ResultSetExtractor -

```
public T query(String sql,ResultSetExtractor<T> rse)
```

- It defines only one method **extractData** that accepts ResultSet instance as a parameter.

```
public T extractData(ResultSet rs)throws SQLException, DataAccessException
```

Example :

```
public List<Student> listStudents() {
   String SQL = "select * from Student";
      List <Student> students = jdbcTemplateObject.query(SQL,
         new ResultSetExtractor<List<Student>>(){

         public List<Student> extractData(
            ResultSet rs) throws SQLException, DataAccessException {

            List<Student> list = new ArrayList<Student>();
            while(rs.next()){
               Student student = new Student();
               student.setId(rs.getInt("id"));
               student.setName(rs.getString("name"));
               student.setAge(rs.getInt("age"));
               student.setDescription(rs.getString("description"));
               student.setImage(rs.getBytes("image"));
               list.add(student);
            }
            return list;
         }
      });
   return students;
}
```

- SQL – Select query to read students.
- jdbcTemplateObject – StudentJDBCTemplate object to read student object from database.
- ResultSetExtractor – ResultSetExtractor object to parse resultset object.

Spring: JDBC Template & ORM

JDBC Template Types

RowMapper

- The **org.springframework.jdbc.core.RowMapper<T>** interface is used by JdbcTemplate for mapping rows of a ResultSet on a per-row basis. Implementations of this interface perform the actual work of mapping each row to a result object.
- SQLExceptions if any thrown will be caught and handled by the calling JdbcTemplate.
- Syntax of query method using RowMapper -

```
public T query(String sql, RowMapper<T> rm)
```

- It defines only one method **mapRow** that accepts ResultSet instance as a parameter.

```
public T (ResultSet rs)throws SQLException,DataAccess
Exception
```

- **Advantages :** RowMapper interface allows to map a row of the relations with the instance of user-defined class. It iterates the ResultSet internally and adds it into the collection. So we don't need to write a lot of code to fetch the records as ResultSetExtractor.

Example :

```
String SQL = "select * from Student";
List <Student> students=jdbcTemplateObject
     .querySQL, new StudentMapper());
```

- SQL – Read query to read all student records.
- jdbcTemplateObject – StudentJDBCTemplate object to read student records from database.
- StudentMapper – StudentMapper object to map student records to student objects.

Spring: JDBC Template & ORM

JDBC Template Types

(Contd…)

NamedParameterJdbcTemplate

- org.springframework.jdbc.core.NamedParameterJdbcTemplate class is a template class with a basic set of JDBC operations, allowing the use of named parameters rather than traditional '?' placeholders.
- It also allows to expand a list of values to the appropriate number of placeholders.
- Interface Declaration :

```
public class NamedParameterJdbcTemplate
          extends Object
          implements NamedParameterJdbcOperations
```

- It is fast in comparison to SOAP because there is no strict specification like SOAP.
- These are reusable and language neutral.

Example :

```
MapSqlParameterSource in = new MapSqlParameterSource();
in.addValue("id", id);
in.addValue("description", new SqlLobValue(description,
new DefaultLobHandler()), Types.CLOB);
String SQL = "update Student set description =
:description where id = :id";
NamedParameterJdbcTemplate jdbcTemplateObject = new
NamedParameterJdbcTemplate(dataSource);
jdbcTemplateObject.update(SQL, in);
```

- in – SqlParameterSource object to pass a parameter to update a query.
- SqlLobValue – Object to represent an SQL BLOB/CLOB value parameter.
- jdbcTemplateObject – NamedParameterJdbcTemplate object to update student object in the database.

Spring: JDBC Template & ORM

JDBC Template Types

(Contd…)

SimpleJdbcTemplate - Update

- Spring JDBC supports the Java 5+ feature var-args (variable argument) and autoboxing by the help of SimpleJdbcTemplate class.
- SimpleJdbcTemplate class wraps the JdbcTemplate class and provides the update method where we can pass arbitrary number of arguments.
- Syntax of update method of SimpleJdbcTemplate class -

```
int update(String sql,Object... parameters)
```

- We should pass the parameter values in the update method in the order they are defined in the parameterized query.

Example :

```
String SQL = "update Student set age = ? where id = ?";

SqlUpdate sqlUpdate = new SqlUpdate(dataSource,SQL);
sqlUpdate.declareParameter(new SqlParameter("age",
Types.INTEGER)); sqlUpdate.declareParameter(new
SqlParameter("id", Types.INTEGER)); sqlUpdate.compile();

sqlUpdate.update(age.intValue(),id.intValue());
```

- SQL - Update query to update student records.
- jdbcTemplateObject – StudentJDBCTemplate object to read student records the from database.
- StudentMapper – StudentMapper object to map student records to student objects.
- sqlUpdate – SqlUpdate object to update student records.

Spring: JDBC Template & ORM

JDBC Template Types

(Contd…)

SimpleJdbcInsert

- The **org.springframework.jdbc.core.SimpleJdbcInsert** class is a multi-threaded, reusable object providing easy insert capabilities for a table.
- It provides meta data processing to simplify the code needed to construct a basic insert statement.
- The actual insert is being handled using Spring's JdbcTemplate
- Class Declaration -

```
public class SimpleJdbcInsert
            extends AbstractJdbcInsert
            implements SimpleJdbcInsertOperations
```

Example :

```
jdbcInsert = new SimpleJdbcInsert(dataSource).withTableName("Student");
          Map<String,Object> parameters = new HashMap<String,Object>();

          parameters.put("name", name);
          parameters.put("age", age);
          jdbcInsert.execute(parameters);
```

- jdbcInsert – SimpleJdbcInsert object to insert record in student table.
- jdbcTemplateObject – StudentJDBCTemplate object to read student object in database.

Spring: JDBC Template & ORM

JDBC Template Types

(Contd…)

SimpleJdbcCall

- The **org.springframework.jdbc.core.SimpleJdbcCall** class is a multi-threaded, reusable object representing a call to a stored procedure or a stored function.
- It provides meta data processing to simplify the code needed to access basic stored procedures/functions.
- Passing the name of the procedure/function and a map containing the parameters when you execute the call. The names of the supplied parameters will be matched up with in and out parameters declared when the stored procedure was created.
- Class Declaration -

```
public class SimpleJdbcCall
            extends AbstractJdbcCall
            implements SimpleJdbcCallOperations
```

Example :

```
SimpleJdbcCall jdbcCall = new
SimpleJdbcCall(dataSource).withProcedureName("getRecord");
SqlParameterSource in = new
      MapSqlParameterSource().addValue("in_id", id);
Map<String, Object> out = jdbcCall.execute(in);
Student student = new Student();
student.setId(id);
student.setName((String) out.get("out_name"));
student.setAge((Integer) out.get("out_age"));
```

- jdbcCall – SimpleJdbcCall object to represent a stored procedure.
- in – SqlParameterSource object to pass a parameter to a stored procedure.
- student – Student object.
- out – Map object to represent output of stored procedure call result.

Spring: JDBC Template & ORM

ORM

Advantage of ORM Frameworks with Spring

Less coding is required

By the help of Spring framework, you don't need to write extra codes before and after the actual database logic such as getting the connection, starting transaction, commiting transaction, closing connection etc.

Easy to test

Spring's IoC approach makes it easy to test the application.

Better exception handling

Spring framework provides its own API for exception handling with ORM framework.

Integrated transaction management

By the help of Spring framework, we can wrap our mapping code with an explicit template wrapper class or AOP style method interceptor.

Spring: JDBC Template & ORM

ORM

(Contd…)

- ➢ Spring Data JPA API provides JpaTemplate class to integrate spring application with JPA.
- ➢ The implementation of JPA specification are provided by –
  - Hibernate - Ibatis - OpenJPA etc.
- ➢ **Advantages** include not writing the before and after code for persisting, updating, deleting or searching object such as creating Persistence instance, creating EntityManagerFactory instance, creating EntityTransaction instance, creating EntityManager instance, commiting EntityTransaction instance and closing EntityManager.

Example of Spring and JPA Integration

Step 1
- create Account.java file

Step 2
- create Account.xml file

Step 3
- create AccountDao.java file

Step 4
- create persistence.xml file

Step 5
- create applicationContext.xml file

Step 5
- create AccountsClient.java file

Spring: JDBC Template & ORM

ORM

(Contd…)

- The Spring framework provides HibernateTemplate class, so you don't need to follow so many steps like create Configuration, BuildSessionFactory, Session, beginning and committing transaction etc.
- This saves a lot of code.
- If we integrate the hibernate application with spring, we don't need to create the hibernate.cfg.xml file. We can provide all the information in the applicationContext.xml file.

Steps for Hibernate and Spring Integration

Step 1

- create table in the database
  - Optional

Step 2

- create applicationContext.xml
  - It contains information of DataSource, SessionFactory etc

Step 3

- create Employee.java file
  - It is the persistent class

Step 4

- create employee.hbm.xml file
  - It is the mapping file

Step 5

- create EmployeeDao.java file
  - It is the dao class that uses HibernateTemplate

Step 5

- create InsertTest.java file
  - It calls methods of EmployeeDao class

Knowledge Check

Name the Incorrect approach for Spring Framework JDBC database access

- ○ JdbcTemplate
- ○ NamedParameterJdbcTemplate
- ○ SimpleJdbcTemplate
- ○ SimpleJdbcDelete

What is not provided by the JdbcTemplate?

- ○ Open/Close Connection
- ○ JDBC Exception Wrapping
- ○ Data Source access
- ○ JDBC Statement Execution

Which data access technology is not supported by the Spring framework?

- ○ JDBC
- ○ Hibernate
- ○ NoSQL
- ○ JPA

Identify the JDBC access approach: SELECT COUNT(*) FROM EMPLOYEE WHERE FIRST_NAME = :firstName";

- ○ JdbcTemplate
- ○ NamedParameterJdbcTemplate
- ○ SimpleJdbcTemplate
- ○ SimpleJdbcInsert

Recap

Glimpse of Important points

Spring JDBC framework offers a convenient class, org.springframework.jdbc.core.support.JdbcDaoSupport, to simplify your DAO implementation

All the database operations can be performed by the help of JdbcTemplate class such as insertion, updation, deletion and retrieval of the data from the database.

We can easily fetch the records from the database using query() method of JdbcTemplate class where we need to pass the instance of ResultSetExtractor.

Spring provides API to easily integrate Spring with ORM frameworks such as Hibernate, JPA, JDO and iBATIS.

Spring framework provides its own API for exception handling with ORM framework.

Any Questions ?

Thank You

Deloitte.