Module 1

# 1

# COLLECTION AND GENERICS

**Unit Structure**

## 1.0 Objectives

After going through this chapter, you will be able to:

- Understand the Collections API
- Learn what is generics and how to write Generic Classes and Methods
- Perform basic operations on Collection
- Learn what are wildcard characters and types of wildcards
- Understand List, Set and Maps

## 1.1 Introduction to Generics

The **Java Generics** was added to JDK 1.5 which allowed programming generically. It allows creating classes and methods that work in the same way on different types of objects while providing type-safety right at the compile-time It makes the code stable by detecting the bugs at compile time.

Before generics, we can store any type of objects in the collection, i.e., non-generic. Now generics force the java programmer to store a specific type of objects.

### 1.1.1 ADVANTAGE OF GENERICS

There are mainly 3 advantages of generics. They are as follows:

**1) Type-safety**

We can hold only a single type of objects in generics. It doesn't allow to store other objects. Without Generics, we can store any type of objects.

***Example***:

```
List list = new ArrayList();
list.add(10);     //Storing an int value in list
list.add("10");   //Storing a String value in list
With Generics, it is required to specify the type of object we need to store.
List<Integer> list = new ArrayList<Integer>();
list.add(10);
list.add("10");// compile-time error
```

**2) Type casting is not required**

There is no need to typecast the object. Before Generics, we need to type cast.

***Example***:

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);   //typecasting
After Generics, we don't need to typecast the object.
```

***Example:***

```
List<String> list = new ArrayList<String>();
list.add("hello");
String s = list.get(0);
```

3) **Compile-Time Checking:** Type checking is done at compile time so errors will not occur at runtime.

***Example:***

```
List<String> list = new ArrayList<String>();
list.add("hello");
list.add(32);//Compile Time Error
```

### 1.1.2 GENERIC CLASSES

Generics is not only limited to collection. It is also possible to use Generics with classes. A generic class declaration looks like a non-generic class declaration, except that the class name is followed by a type parameter section.

The type parameter section of a generic class can have one or more type parameters separated by commas. These classes are known as ***parameterized classes or parameterized types*** because they accept one or more parameters.

***Example: Program that demonstrates Generic Classes***

```
public class Shape<T> {
  private T t;
  public void add(T t) {
    this.t = t;
  }
  public T get() {
    return t;
  }
  public static void main(String[] args) {
    Shape<Integer> intShape = new Shape<Integer>();
    Shape<String> strShape = new Box<String>();

    intShape.add(new Integer(25));
    strShape.add(new String("Generic Classes"));
    System.out.printf("Integer Value :%d\n\n", intShape.get());
    System.out.printf("String Value :%s\n", strShape.get());
  }
}
```

***Output***

Integer Value :25

String Value : Generic Classes

In this program, Box is coded as a Generic Class. Technically, Box is a parametric class and T is the type parameter. T serves as a placeholder for holding any type of Object. We can use the above class by substituting any kind of object for the placeholder T.

### 1.1.3 GENERIC METHODS

A generic method declaration can have arguments of different types. Based on the types of the arguments passed to the generic method, the compiler handles each method call appropriately. Unlike a Generic Class, a generic method containing different parameters affects only that method. alone. Hence, a class which is not generic can contain generic and non-generic methods.

***Rules for defining Generic Methods***

- All generic method declarations have a type parameter section delimited by angle brackets (< and >) that precedes the method's return type < T >
- Each type parameter section contains one or more type parameters separated by commas. A type parameter is an identifier that specifies a generic type name.
- The type parameters can be used to declare the return type and act as placeholders for the types of the arguments passed to the generic method, which are known as actual type arguments.
- A generic method's body is declared like that of any other method.
- Note that type parameters can represent only reference types, not primitive types (like int, double and char).

***Example 1: Program that prints the type of data passed using a single Generic method***

```
public class GenericMethodsEx1
{         // generic method print
public static <T> void print(T t)
{ System.out.println(t.getClass().getName());
}
  public static void main(String args[]) {
GenericMethodsEx1.print(“Hello World”);
GenericMethodsEx1.print(100);
}
}
```

**Output;**

java.lang.String

java.lang.Integer

In this program, the static method print has a return type void and it takes a single parameter called T. T stands for parametric type which can be substituted with any of the Java types by a client application.

***Example 2: Program that prints an array of different type using a single Generic method***

```
public class GenericMethodEx2 {
  // generic method printArray
  public static < T > void printArray( E[] i )
 {
    // Display array elements
    for(T element : i) {
      System.out.printf("%s ", element);
    }
    System.out.println();
  }
  public static void main(String args[]) {
    // Create arrays of Integer, Double and Character
    Integer[] intArray = { 1, 2, 3, 4, 5 };
    Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4 };
    Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };
    System.out.println("Array integerArray contains:");
    printArray(intArray);   // pass an Integer array
    System.out.println("\nArray doubleArray contains:");
    printArray(doubleArray);   // pass a Double array
    System.out.println("\nArray characterArray contains:");
    printArray(charArray);   // pass a Character array
  }
}
```

***Output***

```
Array integerArray contains:
1 2 3 4 5
Array doubleArray contains:
1.1 2.2 3.3 4.4
Array characterArray contains:
H E L L O
```

In this program, the static method printArray has a return type void and it takes a single parameter called T. T stands for parametric type which can be substituted with any of the Java types by a client application. The main method passes arrays with different types and the single generic method is used to print all of the array elements.

### 1.1.4 BOUNDED TYPE PARAMETERS

Sometimes it is required to restrict the kinds of types that are allowed to be passed to a type parameter. Example, a method that operates on numbers might only want to accept instances of Number or its subclasses. In such cases, bounded type parameters can be used.

To declare a bounded type parameter, list the type parameter's name, followed by the extends keyword, followed by its upper bound like below:

class Example <T extends Number>

***Example: Program that calculates the average of numbers either integer or double***

```
package p1;
public class Test<T extends Number> {
   T[] numArr;
   Test(T[] numArr) {
      this.numArr = numArr;
   }
   public double getAvg() {
      double sum = 0.0;
      for (int i = 0; i < numArr.length; i++) {
         sum += numArr[i].doubleValue();
      }
      double avg = sum / numArr.length;
      return avg;
   }
   public static void main(String[] args) {
```

```
    Integer i1[] = {12, 13, 14, 15, 16};
    Double d[] = {1.0, 2.0, 3.0, 4.0};
    Test<Integer> e1 = new Test<Integer>(i1);
    Test<Double> e2 = new Test<Double>(d);
    double ai = e1.getAvg();
    Double ad = e2.getAvg();
    System.out.println("Average of Integers = " + ai);
    System.out.println("Average of Double =" + ad);
  }
```

***Output***

Average of Integers = 14.0

Average of Double =2.5

## 1.2 WILDCARD CHARACTER

- The question mark (?) is known as the wildcard in generic programming . It represents an unknown type.
- The wildcard can be used in a variety of situations such as the type of a parameter, field, or local variable; sometimes as a return type.
- Unlike arrays, different instantiations of a generic type are not compatible with each other, not even explicitly. This incompatibility may be softened by the wildcard if ? is used as an actual type parameter.

### 1.2.1 TYPES OF WILDCARDS

There are three types of wildcards: They are:

**1. UPPER BOUNDED WILDCARDS**

- These wildcards can be used when you want to relax the restrictions on a variable.
- For example, say you want to write a method that works on List < Integer >, List<Double > and List < Number > , you can do this using an upper bounded wildcard.
- To declare an upper-bounded wildcard, use the wildcard character ('?'), followed by the extends keyword, followed by its upper bound

Ex: public static void add(List<? extends Number> list)

***Example: Program that demonstrates Upper Bounded Wildcards***

```
import java.util.Arrays;
import java.util.List;
class UpperBoundedWildcardDemo
{
	public static void main(String[] args)
	{
		//Upper Bounded Integer List
		List<Integer> list1= Arrays.asList(4,5,6,7);

		//printing the sum of elements in list1
		System.out.println("Total sum is:"+sum(list1));
		//Upper Bounded Double list
		List<Double> list2=Arrays.asList(4.1,5.1,6.1);

		//printing the sum of elements in list2
		System.out.print("Total sum is:"+sum(list2));
	}
	private static double sum(List<? extends Number> list)
	{
		double sum=0.0;
		for (Number i: list)
		{
			sum+=i.doubleValue();
		}
		return sum;
	}
}
```

***Output***

Total sum is:22.0

Total sum is:15.299999999999999

Here, the sum method is used to calculate the sum of both Integer list and Double list elements as it accepts list as a parameter whse upper bound is Number.

## 2. LOWER BOUNDED WILDCARDS

- If we use the lower-bounded wildcards you can restrict the type of the "?" to a particular type or a super type of it.
- It is expressed using the wildcard character ('?'), followed by the super keyword, followed by its lower bound: <? super A>

***Syntax***: Collectiontype <? super A>

***Example: Program that demonstrates Lower Bounded Wildcards***

```
import java.util.Arrays;
import java.util.List;
class LowerBoundedWildcardDemo {
   public static void main(String[] args) {
      //Lower Bounded Integer List
      List<Integer> list1 = Arrays.asList(1,2,3,4);
      //Integer list object is being passed
      print(list1);
      //Lower Bounded Number list
      List<Number> list2 = Arrays.asList(1,2,3,4);
      //Integer list object is being passed
      print(list2);
   }

   public static void print(List<? super Integer> list) {
      System.out.println(list);
   }
}
```

***Output***

```
[1, 2, 3, 4]
[1, 2, 3, 4]
```

Here, the print method is used to calculate the print both Integer list and Number list elements as it accepts list as a parameter whse lower bound is Integer.

## 3. UNBOUNDED WILDCARDS

The unbounded wildcard type is specified using the wildcard character (?), for example, List<?>. This is called a ***list of unknown type.***

Consider the following method, printList:

```
public static void printList(List<Object> list) {
   for (Object elem : list)
      System.out.println(elem + " ");
   System.out.println();
}
```

The goal of printList is to print a list of any type, but it fails to achieve that goal — it prints only a list of Object instances; it cannot print List<Integer>, List<String>, List<Double>, and so on, because they are not subtypes of List<Object>.

To write a generic printList method, use List<?>:

***Example: Program that demonstrates UnBounded Wildcards***

```
import java.util.Arrays;
import java.util.List;
public class UnboundedWildcardDemo {
   public static void printList(List<?> list) {
      for (Object elem : list) {
         System.out.println(elem + " ");
      }
      System.out.println();
   }
   public static void main(String args[]) {
      List<Integer> li = Arrays.asList(1, 2, 3);
      List<String> ls = Arrays.asList("one", "two", "three");
      printList(li);
      printList(ls);
   }
}
```

***Output***

```
1
2
3
one
two
three
```

Here, li and ls are Integer and String lists created from Arrays and both lists are printed using the generic method printList.

## 1.3 Introduction To Java Collections

Before Collections, the standard way used for grouping data was using the array, Vector and HashTable. But all of these have different methods and syntax for accessing data and performing operations on it. For example, Arrays uses the square brackets symbol [] to access individual data members whereas Vector uses the method elementAt(). These differences led to a lot of discrepancies. Thus, the "***Collection Framework***" was introduced in JDK 1.2 to bring a unified mechanism to store and manipulate a group of objects.

### 1.3.1 JAVA COLLECTION FRAMEWORK

Any group of objects which are represented as a single unit is known as the collection of the objects. In Java Collections, individual objects are called as ***elements***.

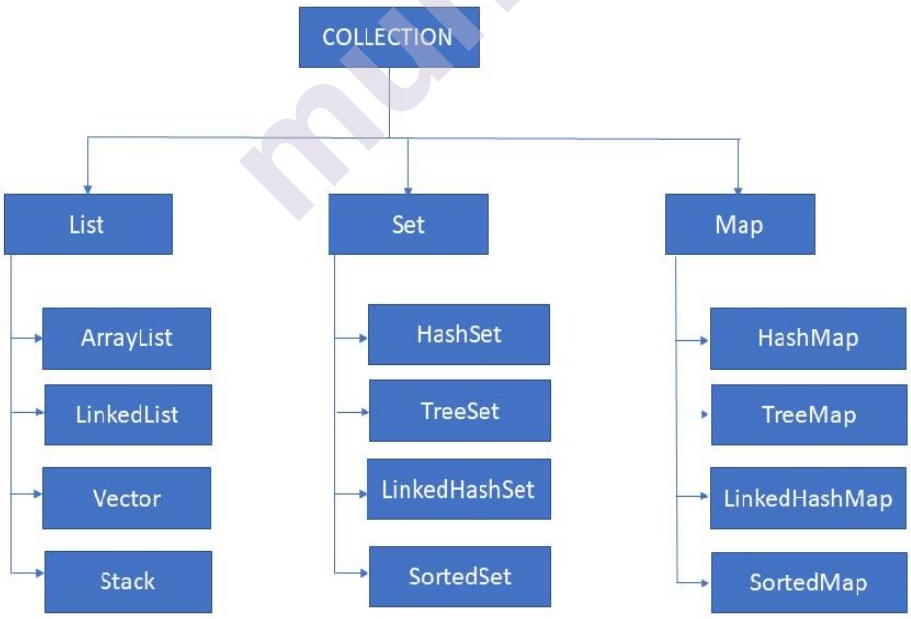
The ***"Collection Framework"*** holds all the collection classes and interface in it. These classes and interfaces define all operations that you can perform uniformly on different types of data such as searching, sorting, insertion, manipulation, and deletion.

The Java Collection framework has

1. Interfaces and its implementations, i.e., classes
2. Algorithm

### 1.3.2 JAVA COLLECTION HIERARCHY

The **java.util** package contains all the classes and interfaces for the Collection framework.




The Collection interface is the root of the collection hierarchy and is implemented by all the classes in the collection framework.

The Collections framework has a set of core interfaces. They are:

- List
- Set
- Map
- Queue

The Collection classes implement these interfaces and provide plenty of methods for adding, removing and manipulating data.

### 1.3.3 ADVANTAGES OF COLLECTION FRAMEWORK

1. **Consistent API:** The API has a basic set of interfaces like *Collection*, *Set*, *List*, or *Map*, all the classes (ArrayList , LinkedList, Vector, etc) that implement these interfaces have some common set of methods.
2. **Reduces programming effort:** A programmer doesn't have to worry about the design of the Collection but rather he can focus on its best use in his program. Therefore, the basic concept of Object-oriented programming (i.e.) abstraction has been successfully implemented.
3. **Increases program speed and quality:** Increases performance by providing high-performance implementations of useful data structures and algorithms.
4. **Clean code –** These APIs have been written with all good coding practices and documented very well. They follow a certain standard across whole Java collection framework. It makes the programmer code look good and clean.

### 1.3.4 BASIC AND BULK OPERATIONS ON JAVA COLLECTION

The Collection Interface provides methods that performs basic operations on data such as:

| Methods | Description |
|---|---|
| boolean add(E element) | Used to add an element in the collection |
| boolean remove(Object element) | Used to delete an element from the collection |
| int size() | Returns the total number of elements in the collection |
| Iterator<E> iterator() | Returns an iterator over the elements in the collection |
| boolean contains(Object element) | Returns true if the collection contains the specified element |
| boolean isEmpty() | Returns true if the collection is empty |

The Collection Interface also contains methods that performs operations on entire collections which are also called as ***bulk operations***.

| Methods | Description |
|---|---|
| boolean containsAll(Collection<?> c) | Used to check if this collection contains all the elements in the invoked collection. |
| boolean addAll(Collection<? extends E> c) | Used to insert the specified collection elements in the invoking collection. |
| boolean removeAll(Collection<?> c) | Used to delete all the elements of the specified collection from the invoking collection. |
| boolean retainAll(Collection<?> c) | Used to delete all the elements of invoking collection except the specified collection. |
| void clear() | Removes all the elements from the collection. |

***Iterator interface***

It provides the facility of iterating the elements in a collection.

| Method | Description |
|---|---|
| public boolean hasNext() | It returns true if the iterator has more elements otherwise it returns false. |
| public Object next() | It returns the element and moves the cursor pointer to the next element. |
| public void remove() | It removes the last elements returned by the iterator. It is less used. |

## LIST

**List** in Java provides the facility to maintain the *ordered collection*. It contains the index-based methods to insert, update, delete and search the elements. It can have the duplicate elements also. We can also store the null elements in the list.

The List interface is found in the java.util package and inherits the Collection interface. It is a factory of ListIterator interface. Through the ListIterator, we can iterate the list in forward and backward directions.

The List interface is declared as:

**public interface List<E> extends Collection<E>**

### 1.4.1 METHODS OF LIST INTERFACE

| Method | Description |
|---|---|
| void add(int index, E element) | It is used to insert the specified element at the specified position in a list. |
| boolean add(E e) | It is used to append the specified element at the end of a list. |
| void clear() | It is used to remove all of the elements from this list. |
| boolean equals(Object o) | It is used to compare the specified object with the elements of a list. |
| int hashcode() | It is used to return the hash code value for a list. |
| boolean isEmpty() | It returns true if the list is empty, otherwise false. |
| int lastIndexOf(Object o) | It is used to return the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element. |

### 1.4.2 HOW TO CREATE LIST?

Since List is an interface, a concrete implementation of the interface needs to be instantiated, in order to use it.

The implementation classes of List interface are ArrayList, LinkedList, Stack and Vector.

- //Creating a List of type String using java.util.ArrayList
  List<String> list=new ArrayList<String>();
- //Creating a List of type String using LinkedList
  List<String> list=new LinkedList<String>();
- java.util.Vector
  List list1=new Vector();
- java.util.Stack
  List list1=new Stack();

The ArrayList and LinkedList are widely used in Java programming.

**ArrayList**

ArrayList is a variable length array of object references.It can dynamically increase or decrease in size. It can be created with an initial size. When this size is exceeded,the collection is automatically enlarged. When objects are removed, the array may be shrunk.

***Example: Program that demonstrates List using ArrayList class***

```
import java.util.*;
public class ArrayListDemo {
   public static void main(String[] args) {
      // Creating a list
      List<Integer> l1= new ArrayList<Integer>();
      // Adds 1 at 0 index
      l1.add(0, 1);
      // Adds 2 at 1 index
      l1.add(1, 2);
      System.out.println(l1);
      // Creating another list
      List<Integer> l2= new ArrayList<Integer>();
      l2.add(1);
      l2.add(2);
      l2.add(3);
      // Will add list l2 from 1 index
      l1.addAll(1, l2);
      System.out.println(l1);
      // Removes element from index 1
      l1.remove(1);
      System.out.println(l1);
      // Prints element at index 3
      System.out.println(l1.get(3));
      // Replace 0th element with 5
      l1.set(0, 5);
      System.out.println(l1);
   }
}
```

**Output:**

```
[1, 2]
[1, 1, 2, 3, 2]
[1, 2, 3, 2]
2
[5, 2, 3, 2]
```

## LinkedList

LinkedList is a class which is implemented in the collection framework which inherently implements the linked list data structure. It is a linear data structure where the elements are not stored in contiguous locations and every element is a separate object with a data part and address part. The elements are linked using pointers and addresses. Each element is known as a node. Linked List permits insertion and deletion of nodes at any point in the list in constant time, but do not allow random access. It permits all elements including null.

***Example: Program that demonstrates List using LinkedList class***

```
import java.io.*;
import java.util.*;
class LinkedListDemo {
   public static void main(String[] args)
   {
      // Size of the LinkedList
      int n = 5;

      // Declaring the List with initial size n
      List<Integer> ll = new LinkedList<Integer>();

      // Appending the new elements
      // at the end of the list
      for (int i = 1; i <= n; i++)
         ll.add(i);

      // Printing elements
      System.out.println(ll);

      // Remove element at index 3
      ll.remove(3);

      // Displaying the list after deletion
      System.out.println(ll);

      // Printing elements one by one
      for (int i = 0; i < ll.size(); i++)
         System.out.print(ll.get(i) + " ");
   }
}
```

**Output:**

[1, 2, 3, 4, 5]
[1, 2, 3, 5]
1 2 3 5

### 1.4.3 ITERATING THROUGH THE LIST

There are multiple ways to iterate through the List. The most famous ways are by using the basic for loop in combination with a get() method to get the element at a specific index and the advanced for loop.

***Example: Program that iterates through an ArrayList***

```
import java.util.*;
public class ArrayListIteration {
    public static void main(String args[])
    {
        List<String> al= new ArrayList<String>();

        al.add("Ann");
        al.add("Bill");
        al.add("Cathy");

        // Using the Get method and the for loop
        for (int i = 0; i < al.size(); i++) {
            System.out.print(al.get(i) + " ");
        }

        System.out.println();

        // Using the for each loop
        for (String str : al)
            System.out.print(str + " ");
    }
}
```

**Output:**

Ann Bill Cathy

Ann Bill Cathy

## 1.5 Set

The set extends the Collection interface is an unordered collection of objects in which duplicate values cannot be stored. This interface is present in the java.util package and contains the methods inherited from the Collection interface.
The most popular classes which implement the Set interface are HashSet and TreeSet.
The List interface is declared as:
**public interface Set extends Collection**

### 1.5.1 METHODS OF SET INTERFACE

| Method | Description |
|---|---|
| add(element) | It is used to add a specific element to the set. |
| addAll(collection) | It is used to append all of the elements from the mentioned collection to the existing set |
| clear() | It is used to remove all the elements from the set but not delete the set. The reference for the set still exists. |
| contains(element) | It is used to check whether a specific element is present in the Set or not. |
| **isEmpty()** | It is used to check whether the set is empty or not. |
| iterator() | It is used to return the iterator of the set. The elements from the set are returned in a random order. |

### 1.5.2 HASHSET CLASS

HashSet class which is implemented in the collection framework is an inherent implementation of the hash table datastructure. This class does not allow storing duplicate elements but permits NULL elements. The objects that we insert into the hashset does not guarantee to be inserted in the same order. The objects are inserted based on their hashcode.

***Example: Program that demonstrates HashSet implementation***

```
import java.util.*;
class HashSetDemo{
   public static void main(String[] args)
   {
      Set<String> h = new HashSet<String>();
```

```
        // Adding elements into the HashSet
        h.add("India");
        h.add("Australia");
        h.add("South Africa");

        // Adding the duplicate element
        h.add("India");

        // Displaying the HashSet
        System.out.println(h);

        // Removing items from HashSet
        h.remove("Australia");
        System.out.println("Set after removing "+ "Australia:" + h);

        // Iterating over hash set items
        System.out.println("Iterating over set:");
        Iterator<String> i = h.iterator();
        while (i.hasNext())
            System.out.println(i.next());
    }
}
```

**Output:**

```
[South Africa, Australia, India]
Set after removing Australia:[South Africa, India]
Iterating over set:
South Africa
India
```



### 1.5.3 TREESET CLASS

TreeSet class which is implemented in the collections framework and implementation of the SortedSet Interface and SortedSet extends Set Interface. It behaves like a simple set with the exception that it stores elements in a sorted format. TreeSet uses a tree data structure for storage. Objects are stored in sorted, ascending order.

***Example: Program that demonstrates TreeSet implementation***

```
import java.util.*;

class TreeSetDemo {
   public static void main(String[] args)
   {
      Set<String> ts = new TreeSet<String>();

      // Adding elements into the TreeSet
      ts.add("India");
      ts.add("Australia");
      ts.add("South Africa");

      // Adding the duplicate element
      ts.add("India");

      // Displaying the TreeSet
      System.out.println(ts);

      // Removing items from TreeSet
      ts.remove("Australia");
      System.out.println("Set after removing "+ "Australia:" + ts);

      // Iterating over Tree set items
      System.out.println("Iterating over set:");
      Iterator<String> i = ts.iterator();
      while (i.hasNext())
         System.out.println(i.next());
   }
}
```

**Output:**

[Australia, India, South Africa]

Set after removing Australia:[India, South Africa]

Iterating over set:

India

South Africa

## 1.6 Maps

The Map interface present in java.util package represents a mapping between a key and a value. A map contains unique keys and each key can map to at most one value. Some implementations allow null key and null value like the HashMap and LinkedHashMap, but some do not like the TreeMap. The order of a map depends on the specific implementations. For example, TreeMap and LinkedHashMap have predictable order, while HashMap does not.

The Map interface is not a subtype of the Collection interface. Therefore, it behaves a bit differently from the rest of the collection types.

The classes which implement the Map interface are HashMap, LinkedHashMap and TreeMap.

### 1.6.1 METHODS OF MAP INTERFACE

| Method | Description |
|---|---|
| clear() | This method is used to clear and remove all of the elements or mappings from a specified Map collection. |
| equals(Object) | This method is used to check for equality between two maps. It verifies whether the elements of one map passed as a parameter is equal to the elements of this map or not. |
| get(Object) | This method is used to retrieve or fetch the value mapped by a particular key mentioned in the parameter. It returns NULL when the map contains no such mapping for the key. |
| hashCode() | This method is used to generate a hashCode for the given map containing key and values. |
| isEmpty() | This method is used to check if a map is having any entry for key and value pairs. If no mapping exists, then this returns true. |
| clear() | This method is used to clear and remove all of the elements or mappings from a specified Map collection. |

### 1.6.2 HASHMAP

HashMap class implements the Map interface which allows us *to store key and value pair*, where keys should be unique. If you try to insert the duplicate key, it will replace the element of the corresponding key. It is easy to perform operations

using the key index like updation, deletion, etc. HashMap class is found in the java.util package.

HashMap in Java is like the legacy Hashtable class, but it is not synchronized. It allows us to store the null elements as well, but there should be only one null key. Since Java 5, it is denoted as HashMap<K,V>, where K stands for key and V for value.

***Example:*** ***Program that demonstrates HashMap implementation***

```
import java.util.*;
public class HashMapDemo {
   public static void main(String[] args) {
      Map<String, Integer> map = new HashMap<>();
map.put("Angel", 10);
 map.put("Liza", 30);
 map.put("Steve", 20);
      for (Map.Entry<String, Integer> e : map.entrySet()) {
         System.out.println(e.getKey() + " " + e.getValue());
      }
   }
}
```

**Output:**

```
vaibhav 20
vishal 10
sachin 30
```

### 1.6.3 LINKEDHASHMAP

LinkedHashMap is just like HashMap with an additional feature of maintaining the order of elements inserted into it. HashMap provided the advantage of quick insertion, search and deletion but it never maintained the track and order of insertion which the LinkedHashMap provides where the elements can be accessed in their insertion order.

***Example: Program that demonstrates LinkedHashMap implementation***

```
import java.util.*;
 public class LinkedHashMapDemo {
    public static void main(String[] args)
   {
      Map<String, Integer> map= new LinkedHashMap<>();

      map.put("Angel", 10);
      map.put("Liza", 30);
      map.put("Steve", 20);

      for (Map.Entry<String, Integer> e : map.entrySet())
         System.out.println(e.getKey() + " "+ e.getValue());
   }
}
```

**Output:**

```
Angel 10
Liza 30
Steve 20
```

### 1.6.4 TREEMAP

The TreeMap in Java is used to implement Map interface and NavigableMap along with the Abstract Class. It provides an efficient means of storing key-value pairs in sorted order. TreeMap contains only unique elements and cannot have a null key but can have multiple null values.

TreeMap is non synchronized and maintains ascending order.

***Example: Program that demonstrates TreeMap implementation***

```
import java.util.*;
class TreeMapDemo {
   public static void main(String args[]) {
      TreeMap<Integer, String> map = new TreeMap<Integer, String>();
```

```
        map.put(100, "Angel");
        map.put(101, "Chris");
        map.put(103, "Bill");
        map.put(102, "Steve");
        for (Map.Entry m : map.entrySet()) {
            System.out.println(m.getKey() + " " + m.getValue());
        }
    }
}
```

**Output:**

100 Angel

101 Chris

102 Steve

103 Bill



## 1.7 Let Us Sum Up

- The "Collection Framework" holds all the collection classes and interface in it and interfaces that define all operations that you can perform uniformly on different types of data such as searching, sorting, insertion, manipulation, and deletion.
- The java.util package contains all the classes and interfaces for the Collection framework.
- Generics allows to program generically. It allows creating classes and methods that work in the same way on different types of objects while providing type-safety right at the compile-time.
- Generics can also be used with classes.
- A generic method declaration can have arguments of different types.
- Bounded type parameters can be used to restrict the kinds of types that are allowed to be passed to a type parameter.
- The question mark (?)known as the wildcard represents an unknown type and can be used in a variety of situations such as the type of a parameter, field, or local variable. There are three types of wildcards-Upper Bounded, Lower Bounded and UnBounded.

- List contains the index-based methods to insert, update, delete, and search the elements. It can have duplicate elements also.
- The Set follows the unordered way and can store only unique elements.
- Map represents a mapping between a key and a value. Each key is linked to a specific value.

## 1.8 List of References

1. Java 8 Programming, BlackBook, DreamTech Press, Edition 2015
2. Core Java 8 for Beginners, Sharanam Shah, Vaishali Shah, Third Edition, SPD

**Web References**

1. https://www.geeksforgeeks.org
2. https://www.javatpoint.com
3. https://www.tutorialspoint.com

## 1.9 Chapter End Exercises

Q1. What are the advantages of collections over primitive datatypes?

Q2. What are generics?List down the advantages of using generics.

Q3. Explain the concept of bounded parameters.

Q4. Differentiate between List,Set and Maps.

Q5. What is a wildcard character? Explain the various types of wildcard characters.

Q6. Explain the difference between ArrayList and LinkedList.

Q7. Explain the difference between HashMap, LinkedHashMap and TreeMap with example programs.

# 2

# LAMBDA EXPRESSIONS

**Unit Structure**



## 2.0 Objectives

After going through this chapter, you will be able to:

- Understand what are Lambda Expressions and how to write lambda expressions
- State the advantages of using lambda expressions
- Explain the different ways parameters can be passed in a Lambda Expressions
- Understand target type inferencing
- Simplify programs and reduce code length using Lambda Expressions

## 2.1 Introduction

**Lambda expressions** is a new and significant feature of Java which was introduced in Java SE 8 and became very popular as it simplifies code development. It provides a very clear and concise way to represent single method interfaces using an expression. It is very useful in collection library. It helps to iterate, filter and extract data from collection. Lambda Expressions is Java's first step towards functional

programming. Lambda expression is treated as a function, so compiler does not create .class file.

An interface having a single abstract method is called a **Functional Interface or Single Abstract Method Interface**. Lambda expression is used to provide the implementation of such a functional interface. In case of lambda expression, we don't need to define the method again for providing the implementation. Here, we just write the implementation code which saves a lot of code. Pre-Java 8, an approach for implementing functional interfaces were anonymous inner classes. However, syntax of anonymous inner classes may seem unclear and cumbersome at times.

To better understand lambda expressions, let us first look into an example for implementing functional interfaces with anonymous inner classes.

In the following program, we have an interface Square with a single abstract method area() in it. We have a class AnonymousClassEx which implements the method in interface Square using anonymous inner class.

***Example: Program to demonstrate Functional Interface Using Anonymous Classes***

```
interface Square {                    //Functional Interface
   public void area();
}
public class AnonymousClassEx {
   public static void main(String[] args) {
      int side = 10;
      /* Without Lambda Expressions, Implementation of Square interface using
Anonymous Inner Class */
      Square s = new Square() {      //Anonymous Class
         public void area()
         {
            System.out.println("Area of square  = " + side * side);
        }
      };
      s.area();
   }
}
```

***Output:***

```
Area of square = 100
```

Here you can note that we are rewriting the method declaration code public void area() written in the interface again in the anonymous inner class. This repetitive code can be eliminated if we use Lambda Expressions.

### 2.1.1 WHAT IS LAMBDA EXPRESSION?

A lambda expression refers to a method that has no name and no access specifier (private, public, or protected) and no return value declaration. This type of method is also known as 'Anonymous methods', 'Closures' or simply 'Lambdas'. It provides a way to represent one method interface simply by using an expression

Like anonymous class, a lambda expression can be used for performing a task without a name.

### 2.1.2 WHY TO USE LAMBDA EXPRESSIONS?

1. Lambda Expressions provides the ability to pass behaviours to methods
2. It provides simplified implementation of Functional Interfaces.
3. Clear and compact syntax
4. Reduces repetitive coding
5. Faster execution time

### 2.1.3 SYNTAX OF LAMBDA EXPRESSION

The basic structure of a lambda expression comprises:

```
(Parameters-list) -> {Expression body}
```

- **Parameter list:** It can be empty or non-empty as well. If non-empty, there is no need to declare the type of the parameters. The compiler can inference the same from the value of the parameter. Also, if there is only one parameter, the parenthesis around the parameter list is optional.
- **Arrow-token (->):** It is used to link parameters-list and body of expression.
- **Expression body:** It contains body of lambda expression. If the body has only one statement then curly braces are optional. You can also use a return statement, but it should be enclosed in braces as it is not an expression.

Consider the following code snippet for understanding the concept of lambda expression, in which a simple method is created for showing a message. Let's first declare a simple message as:

```
public void display() {
System.out.println("Hello World");
 }
```

Now, we can convert the above method into a lambda expression by removing the public access specifier, return type declaration, such as void, and the name of method 'display'.

The lambda expression is shown as follows:

```
() -> {
System.out.println("Hello World");
        }
```

Thus, we have simplified the code.

Now since we know how to write a Lambda Expression, let's rewrite *Example 2.1* using Lambda Expressions:

***Example: Program to demonstrate Functional Interface Implementation Using Lambda Expressions***

```
interface Square {                  //Functional Interface
   public void area();
}
public class LambdaExpressionEx {
   public static void main(String[] args) {
      int side=10;

      //Implementation of Square Interface using Lambda Expression
      Square s= () -> {
System.out.println("Area of square= "+side*side);
                        };
      s.area();
   }
 } }
```

***Output:***

```
Area of square  = 100
```

Here, you can observe that we have not written the method declaration public void area() present in the interface while implementing it in the class using lambda expression. Thus reducing repetitive code.

### 2.1.4 WHERE CAN LAMBDA EXPRESSIONS BE USED?

Lambda Expressions can be written in any context that has a target datatype. The contexts that have target type are:

- Variable declarations, assignments and array initializers
- Return statements
- Method or constructor arguments
- Ternary Conditional Expressions (?:)

## 2.2 LAMBDA TYPE INFERENCE

Type Inference means that the data type of any expression (for e.g., a method return type or parameter type) can be understood automatically by the compiler. Lambda Expressions support Type Inference. Type inference allows us to not specify datatypes for lambda expression parameters. Types in the parameter list can be omitted since java already know the types of the expected parameters for the **single abstract method** of **functional interface**. The compiler infers the type of a parameter by looking elsewhere for the type - in this case the method definition.

***Syntax:***

```
(parameter_name1, parameter_name2 …) -> { method body }
```

***Example:***

```
interface Operation{
   int add(int a,int b);
}

public class Addition {
   public static void main(String[] args) {
      Operation ad1=(a,b)->(a+b);
      System.out.println(ad1.add(10,20));
}
}
```

***Output:***

```
30
```

Here, the compiler can infer that a and b must be int because the lambda expression is assigned to a Operation reference variable.

## 2.3 LAMBDA PARAMETERS

- A lambda expression can be written with zero to any number of parameters.
- If there are no parameters to be passed, then empty parentheses can be given.

For Example:

```
() -> { System.out.println("Zero Parameter Lambda Expression"); }
```

***Example: Program demonstrating Lambda Expression with no parameter***

```
//Functional Interface
interface MyFunctionalInterface {
        //A method with no parameter
   public void say();   }

public class ZeroParamLambda {
  public static void main(String args[]) {
      // lambda expression
       MyFunctionalInterface msg = () -> {
              System.out.println("Hello");
       };
      msg.say();
   }
    }
```

***Output***

```
Hello
```

- When there is a single parameter, if its type is inferred, it is not mandatory to use parentheses.

For Example:

```
(str) -> { System.out.println("Single Parameter Lambda Expression" + str); }
```

***Example: Program demonstrating Lambda Expression with Single Parameter***

```
//Functional Interface
interface MyFunctionalInterface {
   //A method with single parameter
   public void say(String str);
}
public class SingleParamLambda {
   public static void main(String args[]) {
      // lambda expression
      MyFunctionalInterface msg = (str) -> {
         System.out.println(str);
      };
      msg.say("Hello World");
   }
}
```

***Output***

```
Hello World
```

- When there are multiple parameters, they are enclosed in parentheses and separated by commas.

For Example:

```
(str1, str2) ->{
System.out.println("Multiple Parameter Lambda Expression " +str1+ str2);
}
```

***Example: Program demonstrating Lambda Expression with Multiple Parameters***

```
//Functional Interface
interface MyFunctionalInterface {
   //A method with single parameter
   public void say(String str1, String str2);
}
public class SingleParamLambda {
   public static void main(String args[]) {
      // lambda expression
      MyFunctionalInterface msg = (str1, str2) -> {
         System.out.println(str1 +" "+ str2);
      };
      msg.say("Hello", "Java");
   }
}
```

***Output***

```
Hello Java
```



## 2.4 LAMBDA FUNCTION BODY

- The body of a lambda expression, and thus the body of the function / method it represents, is specified to the right of the -> in the lambda declaration.

```
() -> {
        System.out.println("Hello");
};
```

- A lambda Expression body can have zero to any number of statements.
- Statements should be enclosed in curly braces.
- If there is only one statement, curly brace is not needed.
- When there is more than one statement in body then these must be enclosed in curly brackets and the return type of the anonymous function is the same as the type of the value returned within the code block or void if nothing is returned.

## 2.5 RETURNING A VALUE FROM A LAMBDA EXPRESSION

Java lambda expressions can return values , just like from a method. You just add a return statement to the lambda function body, like this:

```
(param) -> {
  System.out.println("param: " + param);
  return "return value";
}
```

In case all your lambda expression is doing is to calculate a return value and return it, you can specify the return value in a shorter way. Instead of this:

```
(num1, num2) -> {return num1 > num2; }
```

You can write:

```
(num1, num2)  -> num1 > num2;
```

The compiler then figures out that the expression a1 > a2 is the return value of the lambda expression .

***Note:*** A return statement is **not** an expression in a lambda expression. We must enclose statements in curly braces ({}). However, we do not have to enclose a void method invocation in braces.

***Example: Program to demonstrate returning a value from lambda expression***

```
interface Operation{
   int add(int a,int b);
}

public class Addition {
   public static void main(String[] args) {

      // Lambda expression without return keyword.
      Operation ad1=(a,b)->(a+b);
      System.out.println(ad1.add(10,20));

      // Lambda expression with return keyword.
      Operation ad2=(int a,int b)->{
                  return (a+b);
                  };
      System.out.println(ad2.add(30,40));
   }
}
```

***Output***

```
30
70
```

## 2.6 LAMBDAS AS OBJECTS

A Java lambda expression is essentially an object. You can assign a lambda expression to a variable and pass it around, like you do with any other object. Here is an example:

```
public interface MyComparator {
   public boolean compare(int num1, int num2);
}
```

```
MyComparator c = (num1, num2) -> num1 > num2;
boolean result = c.compare(2, 5);
```

The first code block shows the interface which the lambda expression implements. The second code block shows the definition of the lambda expression, how the lambda expression is assigned to variable, and finally how the lambda expression is invoked by invoking the interface method it implements.

## 2.6 LAMBDAS IN COLLECTIONS

Lambda Expressions can also be used with different collections such as ArrayList, TreeSet, Treemap, etc… to sort elements in it.

***Example: Program to sort numbers in an ArrayList using Lambda Expression***

```
 import java.util.*;
public class Demo {
   public static void main(String[] args)
   {
      ArrayList<Integer> al = new ArrayList<Integer>();
      al.add(205);
      al.add(102);
      al.add(98);
      al.add(275);
      al.add(203);
      System.out.println("Elements of the ArrayList before sorting : " + al);

      // using lambda expression in place of comparator object
      Collections.sort(al, (o1, o2) -> (o1 > o2) ? -1 : (o1 < o2) ? 1 : 0);

      System.out.println("Elements of the ArrayList after sorting : " + al);
   }
}
```

***Output:***

```
Elements of the ArrayList before sorting : [205, 102, 98, 275, 203]
Elements of the ArrayList after sorting : [275, 205, 203, 102, 98]
```

## 2.7 MORE EXAMPLE PROGRAMS USING LAMBDA EXPRESSIONS

1. Write a program using Lambda expression to calculate average of 3 numbers.

```
interface Operation {
   double average(int a, int b, int c);
}
class Calculate {
   public static void main(String args[]) {
      Operation opr = (a, b, c) -> {
                              double sum = a + b + c;
                              return (sum / 3);
                             };
      System.out.println(opr.average(10, 20, 5));
   }
}
```

***Output***

```
11.666666666666666
```

2. Write a program to store integer values 1 to 5 in an ArrayList and print all numbers,even numbers and odd numbers using Lambda expression.

```
import java.util.ArrayList;
class Test
{
   public static void main(String args[])
   {
      // Creating an ArrayList with elements 1,2,3,4,5
      ArrayList<Integer> arrL = new ArrayList<Integer>();
      arrL.add(1);
      arrL.add(2);
      arrL.add(3);
      arrL.add(4);
      arrL.add(5);

      System.out.println("Displaying all numbers");
      // Using lambda expression to print all elements
      arrL.forEach(n -> System.out.println(n));

      System.out.println("Displaying even numbers");
      // Using lambda expression to print even elements
      arrL.forEach(n -> {
    if (n%2 == 0) System.out.println(n);
```

```
 }
                                );

       System.out.println("Displaying odd numbers");
       // Using lambda expression to print odd elements
       arrL.forEach(n -> {
     if (n%2 != 0) System.out.println(n);
}
    );
   }
       }
```

***Output***

```
Displaying all numbers
1
2
3
4
5
Displaying even numbers
2
4
Displaying odd numbers
1
3
5
```

3. Write a program using Lambda Expressions to calculate the area of circle with radius 5.

```
    interface Area {
   double calculate(int r);
}
class AreaofCircle {
   public static void main(String args[]) {
      Area a = (r) -> {
         double area = 3.142 * r * r;
         return (area);
      };
      System.out.println(a.calculate(5));
   }
}
```

***Output:***

```
Area of circle=78.55
```

4. Write a program to create a Student class to store student details such as id,name and age.Sort student names on the basis of their first names and age using Lambda Expression.

```
import java.util.ArrayList;
import java.util.List;
class Student {
  String name;
  int age;
  int id;
  public String getName() {
    return name;
  }
  public int getAge() {
    return age;
  }
  public int getId() {
    return id;
  }

  Student(String n, int a, int i){
    name = n;
    age = a;
    id = i;
  }
  @Override
   public String toString() {
    return ("Student[ "+"Name:"+this.getName()+
          " Age: "+ this.getAge() +
          " Id: "+ this.getId()+"]");
   }
}
 public class Test1 {
  public static void main(String[] args) {
    List<Student> studentlist = new ArrayList<Student>();
    studentlist.add(new Student("John", 22, 1001));
    studentlist.add(new Student("Steve", 19, 1003));
    studentlist.add(new Student("Kevin", 23, 1005));
    studentlist.add(new Student("Ron", 20, 1010));
```

```
    studentlist.add(new Student("Chris", 18, 1111));

    System.out.println("Before Sorting the student data:");
    //forEach for printing the list using lambda expression
    studentlist.forEach((s)->System.out.println(s));
    System.out.println("\nAfter Sorting the student data by Age:");
    //Lambda expression for sorting student data by age and printing it
    studentlist.sort((Student s1, Student s2)->s1.getAge()-s2.getAge());
    studentlist.forEach((s)->System.out.println(s));
    System.out.println("\nAfter Sorting the student data by Name:");
    //Lambda expression for sorting the list by student name and printing it
    studentlist.sort((Student s1, Student s2)-
>s1.getName().compareTo(s2.getName()));
    studentlist.forEach((s)->System.out.println(s));

  }
}
```

***Output:***

```
Before Sorting the student data:
Student[ Name:John Age: 22 Id: 1001]
Student[ Name:Steve Age: 19 Id: 1003]
Student[ Name:Kevin Age: 23 Id: 1005]
Student[ Name:Ron Age: 20 Id: 1010]
Student[ Name:Chris Age: 18 Id: 1111]
After Sorting the student data by Age:
Student[ Name:Chris Age: 18 Id: 1111]
Student[ Name:Steve Age: 19 Id: 1003]
Student[ Name:Ron Age: 20 Id: 1010]
Student[ Name:John Age: 22 Id: 1001]
Student[ Name:Kevin Age: 23 Id: 1005]
After Sorting the student data by Name:
Student[ Name:Chris Age: 18 Id: 1111]
Student[ Name:John Age: 22 Id: 1001]
Student[ Name:Kevin Age: 23 Id: 1005]
Student[ Name:Ron Age: 20 Id: 1010]
Student[ Name:Steve Age: 19 Id: 1003]
```

## 2.8 Let us Sum up

1. Lambda Expressions is a function which can be created without belonging to any class.
2. It has no name and no access specifier and no return value declaration.
3. It provides a very clear and concise way to represent functional interfaces.
4. Lambda Expression supports type inference.
5. Lambda Expression Parameter list can have zero to any number of parameters.
6. Lambda expression can be assigned to a variable and we can pass it around, like we do with any other object.

## 2.9 List of References

1. Java 8 Programming, BlackBook, DreamTech Press, Edition 2015
2. Core Java 8 for Beginners, Sharanam Shah, Vaishali Shah, Third Edition, SPD

**Web References**

1. http://tutorials.jenkov.com
2. https://www.geeksforgeeks.org
3. https://beginnersbook.com

## 2.10 Chapter End Exercises

Q1. What is Functional Interface? How do you use Lambda Expression with Functional Interfaces?

Q2. Which one of these is a valid Lambda Expression?

(int x, int y) -> x + y;

OR

(x, y) -> x + y

Q3. State the advantages of using Lambda Expressions.

Q4. Write a program using Lambda Expression to calculate the area of a triangle.

Q5. Can Lambda Expressions be passed to different target types? Justify.

❖❖❖❖❖

# 3

# JSP ARCHITECTURE, JSP BUILDING BLOCKS, SCRIPTING TAGS, IMPLICIT OBJECT INTRODUCTION TO JSP STANDARD TAG LIBRARY (JSTL) AND JSTL TAGS

**Unit Structure**

## 3.0 Objectives

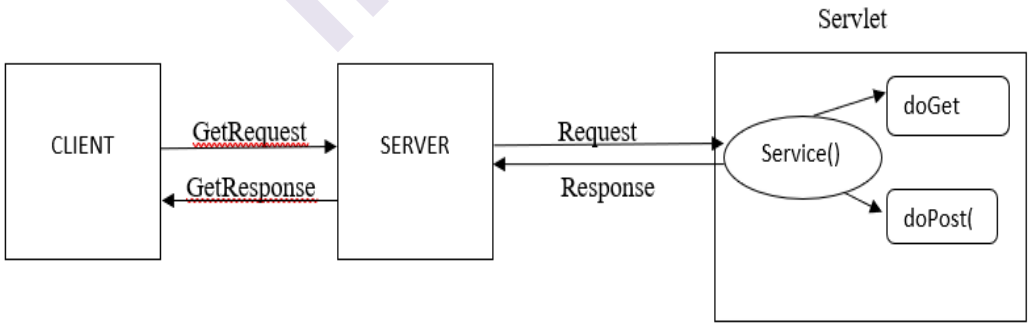
The primary objectives of learning this course will help the students to understand the basic concepts of Servlets, deployment of JSP, development of dynamic web pages, uses of various implicit objects and JSTL predefined library.

## 3.1 Introduction

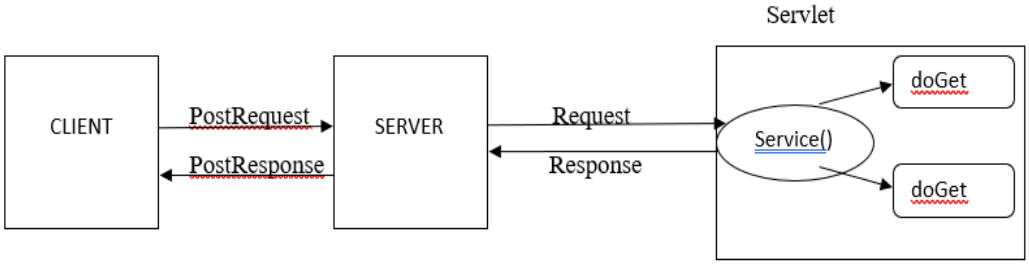
JSP (Java Server Pages) it is an extension or we can say add- on of Servlets technology. JSP provides more functionality as compared to Servlets. It is basically used to develop web applications. It is nothing but similar to Servlets which works on processing the request and generating the response objects, it is the same applies to JSP applications also. In Servlets we accept the request from the client, we process the request and then the Servlets generate the response. The same way we can implement the process of accepting the request and generating the response using JSP in much more simpler way without overriding much more methods as of Servlets. JSP comprises of HTML and JSP tags that is why it said that JSP pages are easier to maintain than Servlets because it gives you a separate space for designing and development of web applications.

Before starting with JSP we should have a rough idea about the Servlets and how it works and the reason behind the use of JSP over Servlets technology. It means that we can say how JSP overcomes the drawbacks of Servlets. Servlets are actually the base of web applications and it has its own life cycle which gets executed each time the request is made from the client side. Servlets are some time also known as server-side coding that is (Business logic).

**Diagram a) How Servlets works using GetRequest() method**



**b) How Servlets works using PostRequest()**




## 3.2 Problems of Servlets:

1) **Static Content: -** using Servlets we can generate static content, but it was difficult to generate dynamic content using Servlets. So in such cases designing in Servlets was a difficult task.

> **Solution using JSP: -** so using JSP we can create dynamic content(dynamic web pages) without wasting much more time on writing the code using HTML tags embedding the java code in it.

2) **For each client request you have to write service() method :-** In Servlets for every new request you have to write the service() method which was causing more processing time at the server end. In short to handle each request in Servlets, we need to have service() for each upcoming request from the client. So this creates a burden on the server-side to handle too much of request and its service() method for each request.

> **Solution using JSP: -** So the solution for it is there should no pressure of writing the code of service() method for each request made by the client. The request can be processed without taking much more time for processing the request. It means that the request should get processed directly without its service() method using JSP.

3) **Any modification made in static content the Servlets need to be recompiled and redeployed:-**So whenever we make any modification or changes in the static content that is (html code) in the presentation logic then the Servlets need to be recompiled and redeployed which is again time consuming part at the server-end.

> **Solution using JSP: -** Using JSP any changes or modification made in the presentation logic (front-end) that is your static content so it does not required to change its dynamic content that is your java code. It means that the java code needs not to be recompiled and redeployed. So this helps to lower the burden of processing the changes made at the server-side.

**Note: Before starting with JSP just go through the Servlets life cycles which will help you further to understand JSP in an easy way.**

## 3.3 Servlets Life Cycle:-

When we talk about the Servlets creation and its destruction there are some stages that the Servlets undergoes during its life cycle. The various stages are:

i) Servlets class is loaded

ii) Servlets instance (object) is created

iii) inti() method is called

iv) service() method is called

v) destroy() method is called

Basically Servlets is a server-side program which is actually resides at the server-end and whenever any request comes from the client to the server, the Servlets gets executed to generate the appropriate response for the request.

Suppose a very first time the request comes from the client its first go to the:-

a) Web Container: - it is responsible for maintaining the Servlets life cycle instance (object creation). So the web container is responsible for creating the object of an Servlets as the request comes from the client, it means that the Servlets class is loaded and the Servlets object is created (instantiated).

b) The second steps is to initializes the object of the Servlets, the Servlets will invoking the its init() method which is responsible for the initialization of Servlets.

c) After the init() method is called, at this stage the request came from the client has not yet processed. To processed the request of the client, the Servlets will call the service() method which is the most important method of Servlets because its contains the business logic of the Servlets.

**Note: In the entire life cycle of Servlets the object is created only once and its init() method is also called once.**

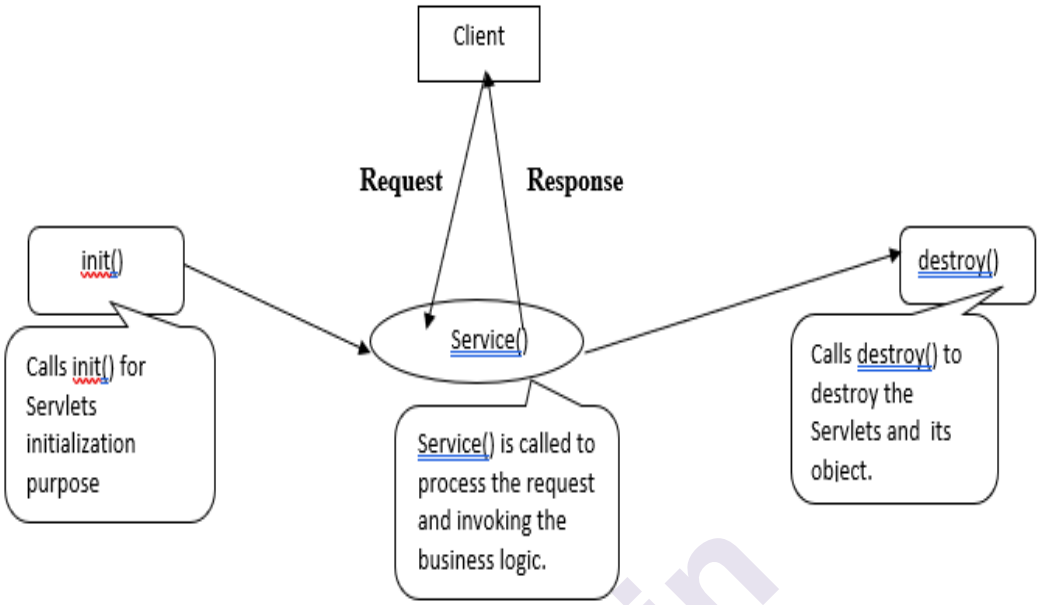
d) After calling the service() method it will starts processing the request of the client and the execution of business logic of service() method gets invoked.

**Note: - For every new request the service () method is called each time by the Servlets.**

e) After processing the request and the execution of service() method , the web container will starts destroying or you can say starts releasing the resources used by the Servlets for processing the request and invoking the service() method.

f) The last stage is to call or invoke the destroy() method of the Servlets and its object.

**Diagram 3.3 (a) Life Cycle of Servlets**




## 3.4 JSP Architecture

JSP: - it is known for server-side programming that gives features like creating dynamic web pages, platform independent method which helps in building web-based applications. JSP have gain access to the family of Java API, including JDBC API which allows to access enterprise databases.

JAVA + HTML = JSP

Before starting JSP you should know about the use of HTML tags. It is a Hypertext Markup Language which helps the programmer to describe or you can say design the structure of web pages. So java is embedded in html tags to create dynamic web pages called as JSP.

JSP: - it is a technology that is supported to develop dynamic web pages using Java code, Html tags and some scripting tags. The JSP pages follow these phases:

1) Translation of JSP page.
2) Compilation of JSP page
3) Classloading
4) Instantiation
5) Initialization
6) Request Processing( web container invokes jspService() method)
7) Destroy (web container invokes jspDestroy() method)

**Note: The life cycle methods of JSP are: jspInit(), jspService(), jspDestroy**

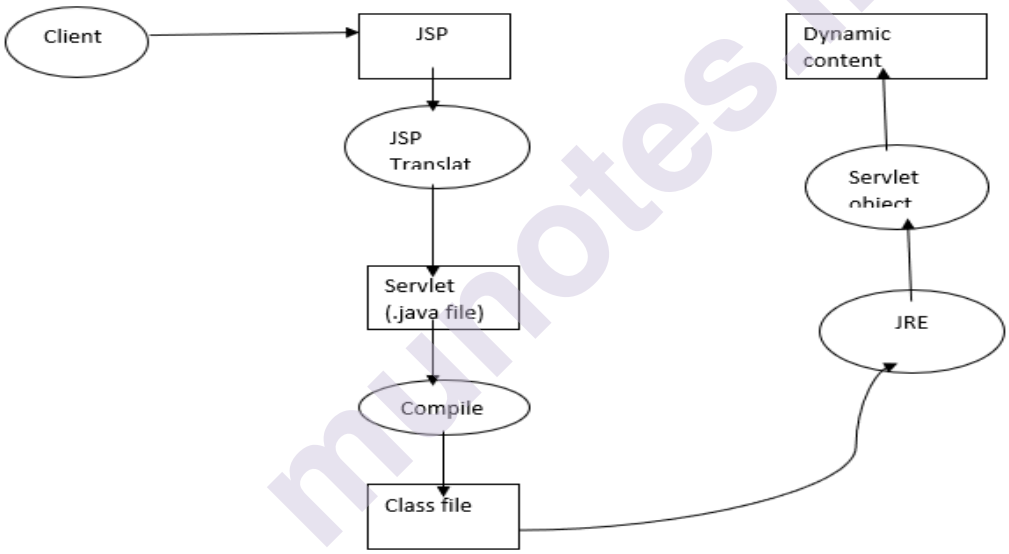
**Step 1:** With the help of JSP translator the JSP page is first translated into Servlets.

**Step 2:** The JSP translator is a part of web server which handles the translation of JSP page into Servlets.

**Step 3:** After that the Servlet page is compiled by the compiler and .class file is created which is binary file.

**Step 4:** All processes that take place in Servlets are performed same in JSL later that is initialization and sending response to the browser and then at the end invoking destroy () method.

**Diagram 3.4.a Life cycle of JSP Page:**






## 3.5 JSP building blocks

There are three building blocks of JSP code:

a) Scriptlet tag

b) Expression tag

c) Declaration tag

**3.5. a. Scriptlet Tag :-** Java provide various scripting elements that helps the programmer to insert java code from your JSP code into the Servlets. The scriptlet elements have different components which help to write the code in jsp. Scripting elements in JSP must be written within the <% %> tags. The JSP engine

will process any code written within the pair of the <% %> tags, and any other text within the Jsp page will be treated as a html code. In short the sciptlet tag is used to execute the java code in JSP.

Syntax: -<% Java code %>

Example:

```
<html>
<body>
<% out.print("Welcome to JSP") %>
</body>
</html>
```

**3.5. b. Expressions Tag: -** Expressions elements are comprises of scripting language expressions, which gets executed and converted to String by the JSP engine and it is meant as a response that is output stream. So there is no need for writing out.print() method.

Syntax: <%=statement %>

Example:

```
<html>
<body>
<%=" JSP based String" %.>
</body>
</html>
```



**3.5. c. Declaration Tag:-** It is used to define or you can say declare methods and variables in JSP. The code written inside the jsp declaration tag is placed outside the service() method.

Syntax: <%! Declaration %>

## 3.6 Scripting Tags

There are five different Scriptlet elements in JSP are:-

1) Comments
2) Directives
3) Declaration
4) JSP scriptlet Tag
5) Expressions

### 3.6.1 Comments

Comments are used to write some text or statements that are ignored by the JSP compiler. It is very useful when someone wants to remember some logic or information in future.

Syntax: <%-- A JSP COMMENT--%>

### 3.6.2 Directives

It is used to give some specific instructions to web container when the jsp page is translated. It has three subcategories:

- Page:<%@ page...>
- Include:<%@ include...%>
- Taglib:<%@ taglib....%>

### 3.6.3 Declaration

It is used to declare methods and variables used in java code within a jsp file. In jsp it is the rule to declare any variable before it is used.

Syntax: <%! Declaration;[declaration;]+…%>

Example: <%! int a=62; %>

### 3.6.4 JSP scriptlet Tag: please refer to( 3.5. a ) Scriptlet Tag

### 3.6.5 Expressions

Its contains the scripting language expressions which is gets evaluated and converted to String by the JSP engine and it is meant to the output stream of the response. So there is no need to write the out.print() method.

Example:

```
<html>
<body>
<%= “ a JSB String” %>
</body>
</html>
```

## 3.7 implicit objects

There are 9 implicit objects in JSP. These objects are created by the web container and it is available to all the JSP pages.

| Object | Type |
|---|---|
| Out | Javax.servlet.jsp.JspWriter |
| Request | Javax.servlet.http.HttpservletRequest |
| Response | Javax.servlet.http.HttpservletResponse |
| Config | Javax.servlet.ServletConfig |
| Application | Javax.servlet.ServletContext |
| Session | Javax.servlet.http.HttpSession |
| pageContext | Javax.servlet.jsp.PageContext |
| Page | Java.lang.Objects |
| Exception | Java.lang.Throwable |

**3.7.1out:-** The out implicit object is an instance of a javax.servlet.jsp.JspWriter object. It is used to send the content in a response.

```
Example:    out.println("Hello Java");
```

**3.7.2 Request: -** The request object is an instance of a javax.servlet.http.HttpServletRequest object. Each time a client requests a page the JSP engine creates a new object to represent that request. It is used to request information such as parameters, header information, server names, cookies, and HTTP methods.

```
Example:  String name=request.getParameter("rname");
```

Some of the methods of request implicit objects are:

| | |
|---|---|
| a) | getAttributesNames() |
| b) | getCookies() |
| c) | getParameterNames() |
| d) | getHeaderNames() |
| e) | getSesssion |
| f) | getSession(Boolean create) |
| g) | getLocale() |
| h) | getAttribute(String name) |

**3.7.3 Response: -** The response object is an instance of a javax.servlet.http.HttpServletResponse object. Just as the server creates the request object, it also creates an object to represent the response to the client.

```
Example:  response.sendRedirect(http://www.google.com);
```

**3.7.4 Config: -** In JSP, config is an implicit object of type ServletConfig. This object can be used to get initialization parameter for a particular JSP page. The config object is created by the web container for each jsp page.

```
Example:    String x=config.getParamter("rname");
```

**3.7.5 Application: -** In JSP, application is an implicit object of type ServletContext. The instance of ServletContext is created only once by the web container when application or project is deployed on the server.

```
Example:  String x=application.getInitParameter("rname");
```

**3.7.6 Session: -** In JSP, session is an implicit object of type HttpSession. In java developer can use this object to set, get, and remove attribute or to get session information.

```
Example:   session.setAttribute("user", x);
```

**3.7.7 pageContext :-** In JSP, pageContext is an implicit object of type PageContext class. The pageContext object can be used to set, get and remove attribute from one of the following scopes:

i) Page

ii) request

iii) session

iv) application

**3.7.8 Page: -** This object acts as an actual reference to the instance of the page. This object can be used to represent the entire jsp page.

```
Example:  pageContext.removeAttribute("attrName", PAGE_SCOPE);
```

**3.7.9 Exception: -** In JSP, exception is an implicit object of type java.lang.Throwable class. This object can be used to print the exception but it should be used only in error pages.

## 3.8 Introduction to JSP Standard Tag Library (JSTL) and JSTL Tags

JSTL –it is a collection of predefined tags to simply the jsp development. To develop any jsp application we write the code in java so for writing this java code we need some tags such as scriptlet tags. So writing the logical code in java, this leads the increase in the length of the code which results in more processing time.

**Advantages of JSTL:**

1) **Fast Development: -** provides many tags that simply the JSP. To write the logical code in jsp you need to use the scriptlet tags. To avoid using this logical code, JSTL provides some pre-defined logical tags.
2) **Code Reusability: -** We can use the JSTL tags on various pages.
3) **No need to use scriptlet tags**: - It avoids the use of scriptlet tags.

**JSTL Tags: -** It is a predefined library which provides custom tags.

1) **Core tags: -** it helps in variables support, URL management, flow control, etc.
2) **Functions tags: -** it provides support for String manipulation and getting the String length.
3) **Formatting tags: -** it provides support for messaging, number, date formatting etc.
4) **XML tags: -** it is used for manipulating and for creating XML documents.
5) **SQL tags:** - it provides SQL support such as database connectivity.

## 3.7 Summary

This course will helps to build skills gained by the students in Java fundamentals and advanced java programming skills.

## 3.8 Reference for further reading.

javadocs.in

javapoint.com

W3schools.in

## 3.9 Bilbliograpgy

https://www.javapoint.com

https://www.w3schools.in

## MCQ FOR PRACTICE

1. Which page directive should be used in JSP to generate a PDF page?
   a. contentType
   b. generatePdf
   c. typePDF
   d. contentPDF
2. Application is instance of which class?
   a. javax.servlet.Application
   b. javax.servlet.HttpContext
   c. javax.servlet.Context
   d. javax.servlet.ServletContext
3. _jspService() method of HttpJspPage class should not be overridden.
   a. True
   b. False
4. Which of the following is not a directive in JSP?
   a. Include
   b. Page
   c. Export
   d. useBean
5. In JSP config instance is of which class?
   a. javax.servlet.ServletContext
   b. javax.servlet.ServletConfig
   c. javax.servlet.Context
   d. javax.servlet.Application

# 4

# INTRODUCTION TO BEAN, STANDARD ACTIONS, SESSION TRACKING TYPES AND METHODS. CUSTOM TAGS

**Unit Structure**





## 4.0 Objectives

EJP stands for Enterprise Java Beans. It is a server-side component. It means that EJB is basically used at server-side coding. So if we have client-side and server-side, EJB is used for server-side and not for client-side.

## 4.1 Introduction

EJB is an essential part of a J2EE platform.J2EE application container contains the components that can be used by client for executing business logic. These components are called business logic and business data.EJB mainly comprises of business logic and business data. The EJB component always lies in some container which is called as EJB container. The EJB component is an EJB class which is written by the developer that implement business logic.

## 4.2 Introduction to Beans

JavaBeans are nothing it's a class that encapsulates many objects into a single object that is nothing but a bean. Java beans should follow some protocol such as:

1) They are serializable
2) Have a zero- argument constructor.
3) Allows access to properties using as getter and setter methods.

**Example of JavaBeans class- students.java**

```
package mypack;
public class students implements java.io.Serializable
{
private int RollNo;
private String name;
public students(){}
public void setRollNo(int id)
{
this.RollNo=RollNo;
}
public int getId()
{
return RollNo;
}
public void setName(String name)
{
this.name=name;
```

```
}
public String getName()
{
return name;
}
}
```

**How to access the JavaBean class?**

**To access the JavaBean class, we should use getter and setter methods.**

```
package mypack;
public class Test
{
public static void main(String args[])
{
students s=new students();   //object is created
s.setName("Rahul");          //setting value to the object
System.out.println(s.getName());
}
}
```

## 4.3 standard actions

Standard actions in JSP are used to control the behaviour of the Servlets engine. In JSP there are 11 standard actions tag. With the help of these tags we can dynamically insert a file, reuse the beans components, forward user etc.

**Syntax: <jsp:action_name attribute=”value” />**

3 jsp:useBean
4 jsp:include
5 jsp:setProperty
6 jsp:forward
7 jsp:plugin
8 jsp:attribute
9 jsp:body
10 jsp:text
11 jsp:param
12 jsp:attribute
13 jsp:output

**4.3.1 jsp:useBean:-** This action tag is used when we want to use beans in the JSP page. Using this tag the beans can be easily invoked

**Syntax: <jsp:useBean id="" class="" />**

**4.3.2 jsp:include:-** This tag is used to insert a jsp file into another file , same as include directive . It is compute at the time request processing phase.

**Syntax: <jsp:include page="page URL" flush="true/false">**

**4.3.3 jsp:setProperty:-** This tag used to set the property of a bean. Before setting this property we need define a bean.

**Syntax: <jsp:setProperty name="" property="">**

**4.3.4 jsp:getProperty:-** To get the property of a bean we use this tag. It inserts the output in a string format which is converted into string.

**Syntax: <jsp:getAttribute name="" property="">**

**4.3.5 jsp:forward:-** It is basically used to forward the request to another jsp or any static page. Here the request can be forwarded with or with no parameters.

**Syntax: <jsp:forward page="value">**

**4.3.6 jsp:plugin:-** It is used for introducing Java components into JSP which is detects the browser and adds the <object> or <embed> JSP tags into the file

**Syntax: <jsp:plugin type="applet/bean" code="objectcode" codebase="objectcodebase">**

**4.3.7 jsp:param:-** It is the child object of the plugin object. It contains one or more actions to provide additional parameters.

**Syntax: <jsp:params>**

**<jsp:param nname="val" value="val">**

**</jsp:param>**

**4.3.8 jsp:body:-** This tag is used for defining the xml dynamically that is the elements can be generated during the request time than at the compilation time.

**Syntax: <jsp:body></jsp:body>**

**4.3.9 jsp:attribute:-**This tag is used for defining the xml dynamically that is the elements can be generated during the request time than at the compilation time.

**Syntax: <jsp:attribute></jsp:attribute>**

**4.3.10 jsp:text:-** To template text in JSP pages this tag is used. The body of this tag does not contains any elements .It only contains text and EL expressions.

**Syntax: <jsp:text>template</jsp:text>**

**4.3.11jsp: output: -** Its consists of XML template text which is placed within text action objects.In this output is declared as XML and DOCTYPE.

**Syntax: <jsp:output doctype-root-element="" doctype –system="">**

## 4. 4 session tracking types amd methods:

There are four techniques which can be used to identify a user session.

a) Cookies

b) Hidden Fields

c) URL Rewriting

d) Session Object

### 4.4.1 Cookie: -

A cookie is small information which is sent by the web server to a web client. It is save at the client side for the given domain and path. It is basically used to identify a client when sending a subsequent request. There are two types of cookies:

a) **Session cookies:** these are temporary cookies and its get deleted as soon as the user closes the browser and next time whenever the client visits the same websites, server will treat the request as a new client as cookies are already deleted.

b) **Persistent Cookie**: its remains on hard drive, until we delete them or they gets expire.

### 4.4.2 Hidden Filed:

Hidden field are similar to other input fields with the only difference is that these fields are not get displayed on the page but the values of these fields are sent to other input fields.

For example: **<input type="hidden" name="sessionId" value="unique value"/>**

### 4.4.3 URL Rewriting:

It is a process of appending or modifying the url structure when loading a page. The request made by the client is always treated as new request and the server cannot identify whether the request is new one or the previous same client .so due to this property of HTTP protocol and web servers are called stateless.

### 4.4.4 Session Object:

It is used for session management. When a user enters a website for the first time Httpsession is obtained via request. When session is created, server generates aunique ID and attaches that ID with every request of that user to server with which the server identifies the client.

**How to access or get a session object: By calling getSession() method and it is an implicit object.**

a) HttpSession x=request.getSession()

b) HttpSession y=new request.getSession(Boolean)

## 4.5 Custom Tags:

- Custom tags, also known as JSP tag extensions (because they extend the set of built-in JSP tags), provide a way of encapsulating reusable functionality on JSP pages.
- One of the major drawbacks of scripting environments such as JSP is that it's easy to quickly put together an application without thinking about how it will be maintained and grown in the future.
- Use JavaBeans for representing and storing information and state. An example is building JavaBeans to represent the business objects in your application.
- Use custom tags to represent and implement actions that occur on those JavaBeans, as well as logic related to the presentation of information.
- A example from JSTL is iterating over a collection of objects or conditional logic.
- Custom tags have access to implicit objects like request, response, session, etc
- JavaBeans are java classes but all java class are not java beans.
- major one is Custom tag which can be use by the java beans to communicate with each other.
- JavaBeans are normal java classes and don't know anything about JSP.
- JavaBeans are normally used to maintain the data and custom tags for functionality or implementing logic on jsp page.

## 4.6 Reference for further reading.

https://docs.oracle.com/javase/7/docs/api/

## 4.7 Bilbliograpgy

https://www.wideskills.com/jsp/jsp-session-tracking-techniques

https://www.geeksforgeeks.org/url-rewriting-using-java-servlet/

https://www.javatpoint.com/java-bean

https://www.javatpoint.com/what-is-ejb

https://www.w3schools.in

https://www.java-samples.com/showtutorial.php?tutorialid=607

## MCQ FOR PRACTICE

1. Which page directive should be used in JSP to generate a PDF page?
   a. contentType
   b. generatePdf
   c. typePDF
   d. contentPDF

2. Application is instance of which class?
   a. javax.servlet.Application
   b. javax.servlet.HttpContext
   c. javax.servlet.Context
   d. javax.servlet.ServletContext

3. _jspService() method of HttpJspPage class should not be overridden.
   a. True
   b. False

4. Which of the following is not a directive in JSP?
   a. Include
   b. Page
   c. Export
   d. useBean

5. In JSP config instance is of which class?
   a. javax.servlet.ServletContext
   b. javax.servlet.ServletConfig
   c. javax.servlet.Context
   d. javax.servlet.Application

❖❖❖❖❖

Module 2

# 5

# INTRODUCTION TO SPRING

**Unit Structure**



### 5.1 Objectives:

This chapter would make you understand the following concepts:

- Spring Framework
- POJO programming model
- IoC container
- How to configure metadata with container

## 5.2 Introduction to Spring Framework

- Initially Java developers needed to use JavaBeans technology to create Web applications.
- Although JavaBeans helped in the development of user interface (UI) components, they were not able to provide services, such as transaction management and security, which were required for developing robust and secure enterprise applications.
- EJB (Enterprise Java Beans) was seen as a solution to this problem , which extends the Java components, such as Web and Enterprise components, and also provides services that help in enterprise application development.
- However, developing an enterprise application with EJB was not easy, as the developer needed to perform various tasks, such as creating Home and Remote interfaces and implementing lifecycle methods which lead to the

complexity of providing code, so developers started looking for an easier way to develop such applications.

- The Spring framework has developed as a solution to all these problems.
- This framework uses various new techniques such as Aspect-Oriented Programming (AOP), Plain Old Java Object (POJO), and dependency injection (DI), to develop enterprise applications.
- Spring is an open source lightweight framework that allows Java EE 7 developers to build simple, reliable, and scalable enterprise applications.
- It made the development of Web applications much easier as compared to classic Java frameworks and Application Programming Interfaces (APIs), such as Java database connectivity(JDBC), JavaServer Pages(JSP), and Java Servlet.

### 5.2.1 Spring Framework

- Spring is the most popular application development framework for enterprise Java. Millions of developers around the world use Spring Framework to create high performing, easily testable, and reusable code.
- Spring framework targets to make J2EE development easier to use and promotes good programming practices by enabling a POJO-based programming model.
- Spring could potentially be a one-stop shop for all your enterprise applications. However, Spring is modular, allowing you to pick and choose which modules are applicable to you, without having to bring in the rest.
- The following section provides details about all the modules available in Spring Framework.

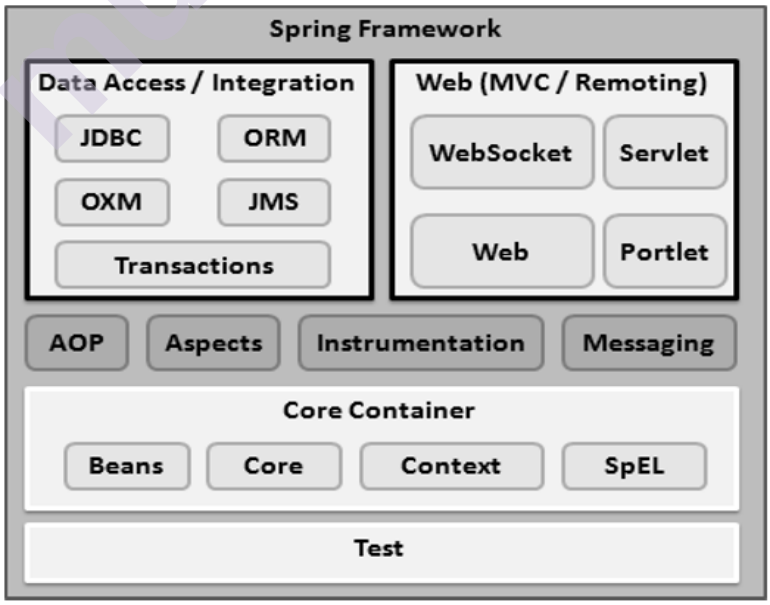



**Fig 1 .Spring Framework**

## Core Container

The Core Container consists of the Core, Beans, Context, and Expression Language modules the details of which are as follows −

- The **Core** module provides the fundamental parts of the framework, including the IoC and Dependency Injection features.
- The **Bean** module provides BeanFactory, which is a sophisticated implementation of the factory pattern.
- The **Context** module builds on the solid base provided by the Core and Beans modules and it is a medium to access any objects defined and configured. The ApplicationContext interface is the focal point of the Context module.
- The **SpEL** module provides a powerful expression language for querying and manipulating an object graph at runtime.

## Data Access/Integration

The Data Access/Integration layer consists of the JDBC, ORM, OXM, JMS and Transaction modules whose detail is as follows −

- The **JDBC** module provides a JDBC-abstraction layer that removes the need for tedious JDBC related coding.
- The **ORM** module provides integration layers for popular object-relational mapping APIs, including JPA, JDO, Hibernate, and iBatis.
- The **OXM** module provides an abstraction layer that supports Object/XML mapping implementations for JAXB, Castor, XMLBeans, JiBX and XStream.
- The Java Messaging Service **JMS** module contains features for producing and consuming messages.
- The **Transaction** module supports programmatic and declarative transaction management for classes that implement special interfaces and for all your POJOs.

## Web

The Web layer consists of the Web, Web-MVC, Web-Socket, and Web-Portlet modules the details of which are as follows −

- The **Web** module provides basic web-oriented integration features such as multipart file-upload functionality and the initialization of the IoC container using servlet listeners and a web-oriented application context.
- The **Web-MVC** module contains Spring's Model-View-Controller (MVC) implementation for web applications.

- The **Web-Socket** module provides support for WebSocket-based, two-way communication between the client and the server in web applications.
- The **Web-Portlet** module provides the MVC implementation to be used in a portlet environment and mirrors the functionality of Web-Servlet module.

**Miscellaneous**

There are few other important modules like AOP, Aspects, Instrumentation, Web and Test modules the details of which are as follows −

- The **AOP** module provides an aspect-oriented programming implementation allowing you to define method-interceptors and pointcuts to cleanly decouple code that implements functionality that should be separated.
- The **Aspects** module provides integration with AspectJ, which is again a powerful and mature AOP framework.
- The **Instrumentation** module provides class instrumentation support and class loader implementations to be used in certain application servers.
- The **Messaging** module provides support for STOMP as the WebSocket sub-protocol to use in applications. It also supports an annotation programming model for routing and processing STOMP messages from WebSocket clients.
- The **Test** module supports the testing of Spring components with JUnit or TestNG frameworks.

## 5.3 POJO Programming Model:

- POJO in Java stands for **Plain Old Java Object**.
- Generally, a POJO class contains variables and their Getters and Setters.
- The POJO classes are similar to Beans as both are used to define the objects to increase the readability and re-usability.
- The only difference between them that Bean Files have some restrictions but, the POJO files do not have any special restrictions.
- POJO simply *means* a class that is not forced to implement any interface, extend any specific class, contain any pre-described annotation or follow any pattern due to forced restriction.
- POJO class is used to define the object entities. For example, we can create an Employee POJO class to define its objects.
- Below is an example of Java POJO class:

**Employee.java:**

```
// POJO class Exmaple
package Jtp.PojoDemo;
public class Employee
{       private String name;
private String id;
private double sal;
public String getName() {
return name;  }
public void setName(String name) {
this.name = name;  }
public String getId() {
return id;  }
public void setId(String id) {
this.id = id;  }
public double getSal() {
return sal;  }
public void setSal(double sal) {
this.sal = sal;  }
}
```

**How to use POJO class in a Java Program :**

To access the objects from the POJO class, follow the below steps:

- ✓ Create a POJO class objects
- ✓ Set the values using the set() method
- ✓ Get the values using the get() method

For example, create a MainClass.java class file within the same package and write the following code in it:

**MainClass.java:**

```
//Using POJO class objects in MainClass Java program
package Jtp.PojoDemo;
public class MainClass {
   public static void main(String[] args) {
      // Create an Employee class object
   Employee obj= new Employee();        //POJO class object created.
  obj.setName("Alisha");              // Setting the values using the set() method
```

```
  obj.setId("A001");
  obj.setSal(200000);
  //Getting the values using the get() method
 System.out.println("Name: "+ obj.getName());
 System.out.println("Id: " + obj.getId());
 System.out.println("Salary: " +obj.getSal());
  }
}
```

**Output:**

```
Name: Alisha
Id: A001Salary: 200000.0
```

**Properties of POJO class:**

- ✓ The POJO class must be public.
- ✓ It must have a public default constructor.
- ✓ It may have the arguments constructor.
- ✓ All objects must have some public Getters and Setters to access the object values by other Java Programs.
- ✓ The object in the POJO Class can have any access modifies such as private, public, protected. But, all instance variables should be private for improved security of the project.
- ✓ A POJO class should not extend predefined classes.
- ✓ It should not implement prespecified interfaces.
- ✓ It should not have any prespecified annotation.

## 5.4 Lightweight Containers:

### 5.4.1 Spring IoC Container:

- The Spring container is the core of Spring Framework.
- The container, use for creating the objects and configuring them.
- Also, Spring IoC Containers use for managing the complete lifecycle from creation to its destruction.
- It uses Dependency Injection (DI) to manage components and these objects are called Spring Beans.
- The container uses configuration metadata which represent by Java code, annotations or XML along with Java POJO classes as seen below.

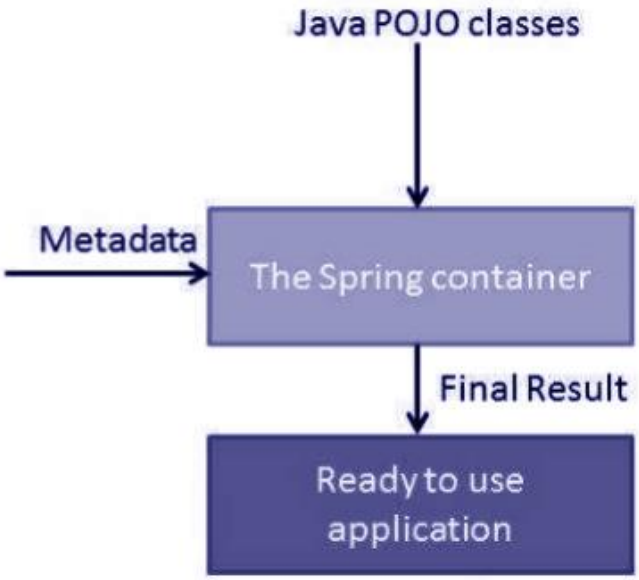



**Fig. Spring IoC Container**

**Types of IoC Containers:**

This are the two types of Spring IoC Containers.

**5.4.1.1 Spring BeanFactory Container:**

- Spring BeanFactory Container is the simplest container which provides basic support for DI.
- It is defined by org.springframework.beans.factory.BeanFactory interface.
- There are many implementations of BeanFactory interface that come with Spring where XmlBeanFactory being the most commonly used class.
- XmlBeanFactory reads configuration metadata from XML file for creating a fully configured application.
- The BeanFactory container prefer, where resources are limited to mobile devices or applet-based applications.
- You will look at a working example with Eclipse IDE with the following steps for creating Spring application.
    i. Create a project with a name SpringExample and a package packagecom.example. These should be under src folder of the created project.
    ii. Add the needed Spring libraries using Add External JARs.
    iii. Create Java classes HelloWorld and MainApp under the package packagecom .example.
    iv. Create Beans config file Beans.xml under src folder.
    v. At last, create content of all Java files and Beans configuration file and run the file as below.

The code of **HelloWorld.java** is as shown.

```
package com.example;

public class HelloWorld {

private String message;

public void setMessage(String message){

this.message = message;

}

public void getMessage(){

System.out.println("Your Message : " + message);

}

}
```

The following is the code of **MainApp.java**.

```
package com.example;

import org.springframework.beans.factory.InitializingBean;

import org.springframework.beans.factory.xml.XmlBeanFactory;

import org.springframework.core.io.ClassPathResource;

public class MainApp {

public static void main(String[] args) {

XmlBeanFactory factory = new XmlBeanFactory (new
ClassPathResource("Beans.xml"));

HelloWorld obj = (HelloWorld) factory.getBean("helloWorld");

obj.getMessage();

}

}
```

Please note:

- Write a factory object where you have used APIXmlBeanFactory() to load bean config file in CLASSPATH.
- Use getBean() which uses bean ID to return a generic object to get the required bean.

Following is the XML code for **Beans.xml**.

```
<?xml version = "1.0" encoding = "UTF-8"?>

<beans xmlns = "http://www.springframework.org/schema/beans"

xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation = "http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<bean id = "helloWorld" class = "com.example.HelloWorld">

<property name = "message" value = "Hello World!"/>

</bean>

</beans>
```

After you run the application you will see the following message as output.
**Your Message: Hello World!**



### 5.4.1.2 Spring ApplicationContext Container:

- The ApplicationContext container is Spring's advanced container.
- It is defined by org.springframework.context.ApplicationContext interface.
- The ApplicationContext container has all the functionalities of BeanFactory.
- It is generally recommended over BeanFactory.
- The most common implementations of ApplicationContext are:
  - FileSystemXmlApplicationContext: It is a type of container which loads the definitions of beans from an XML file. For that, you should be able to provide the full path of the XML bean config file to a constructor.

- ClassPathXmlApplicationContext: This type of container loads definitions of the beans from XML file but you don't need to provide the full path of the XML file. Only the CLASSPATH has to set properly as this container will look like Bean config XML file.
- WebXmlApplicationContext: This type of container loads the XML file with all bean definitions within a web application.

- You will better understand with a working example in Eclipse IDE with the following steps:
  i. Create a project with a name SpringExample and a package packagecom.example. These should under src folder of the created project.
  ii. Add the needed Spring libraries using Add External JARs.
  iii. Create Java classes HelloWorld and MainApp under the package packagecom .example.
  iv. Create Beans config file Beans.xml under src folder.
  v. At last, create content of all Java files and Beans configuration file and run the file as below.

The code for **HelloWorld.java** file:

```
package com.example;

public class HelloWorld {

private String message;

public void setMessage(String message){

this.message = message;

}

public void getMessage(){

System.out.println("Your Message : " + message);

}

}
```

The code for **MainApp.java**:

```
package com.example;

import org.springframework.context.ApplicationContext;

import
org.springframework.context.support.FileSystemXmlApplicationContext;

public class MainApp {

public static void main(String[] args) {

ApplicationContext context = new FileSystemXmlApplicationContext

("C:/Users/ADMIN/workspace/HelloSpring/src/Beans.xml");

HelloWorld obj = (HelloWorld) context.getBean("helloWorld");

obj.getMessage();

}

}
```



Please note:

- Using framework API FileSystemXmlApplicationContext create a factory object. This API takes care of creating and initializing all objects.
- Use getBean() which uses bean ID to return a generic object to get the required bean.

The code for **Beans.xml** is as given:

```
<?xml version = "1.0" encoding = "UTF-8"?>

<beans xmlns = "http://www.springframework.org/schema/beans"

xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation = "http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<bean id = "helloWorld" class = "com.example.HelloWorld">

<property name = "message" value = "Hello World!"/>

</bean>

</beans>
```



### 5.4.2 Configuration Metadata:

- The configuration metadata allows you to express the objects that compose your application and the rich interdependencies between such objects.
- Configuration metadata is supplied in a simple and intuitive XML format.
- Spring configuration consists of at least one and typically more than one object definition that the container must manage. XML- based configuration shows these objects as <object/> elements inside a top-level <objects/> element.
- These object definitions correspond to the actual objects that make up your application. Typically you define service layer objects, data access objects (DAOs), presentation objects such as ASP.NET page instances, infrastructure objects such as NHibernate SessionFactories, and so forth.
- The following example shows the basic structure of XML-based configuration metadata:

```
<objects xmlns="http://www.springframework.net">
  <object id="..." type="...">
    <!-- collaborators and configuration for this object go here -->
  </object>
  <object id="...." type="...">
    <!-- collaborators and configuration for this object go here -->
  </object>

  <!-- more object definitions go here -->
</objects>
```

Note : The id attribute is a string that you use to identify the individual object definition. The type attribute defines the type of the object.

### 5.4.3 Configuring and Using the Container:

- Instantiating a Spring IoC container is straightforward.
- The location path or paths suppied to an IApplicationContext constructor are actually resource strings that allow the container to load configuration metadata from a variety of external resources such as the local file system, embedded assembly resources, and so on.

**Example:**

IApplicationContext context = new XmlApplicationContext("services.xml", "data-access.xml");

(Note :the service layer objects (services.xml) configuration file.the data access objects (daos.xml) configuration file)

**service.xml:**

```
<objects xmlns="http://www.springframework.net">
<object id="PetStore" type="PetStore.Services.PetStoreService, PetStore">
        <property name="AccountDao" ref="AccountDao"/>
    <property name="ItemDao" ref="ItemDao"/>

<!-- additional collaborators and configuration for this object go here -->
    </object>
  <!-- more object definitions for services go here -->
</objects>
```

(Note : the service layer consists of the class PetStoreService.)

An IApplicationContext is the interface for an advanced factory capable of maintaining a registry of different objects and their dependencies. Using the method GetObject(string) or the indexer [string] you can retrieve instances of your objects.

The IApplicationContext enables you to read object definitions and access them as follows:

```
// create and configure objects

IApplicationContext context = new XmlApplicationContext("services.xml",
"daos.xml");

// retrieve configured instance

PetStoreService service = (PetStoreService)
context.GetObject("PetStoreService");

// use configured instance

IList userList = service.GetUserNames();
```

## 5.5 Summary:

- Spring is an open source lightweight framework.
- Spring is the most popular application development framework for enterprise Java.
- POJO in Java stands for Plain Old Java Object.
- POJO simply *means* a class that is not forced to implement any interface, extend any specific class.
- Spring container is the core of Spring Framework.
- Spring IoC Containers use for managing the complete lifecycle from creation to its destruction.
- The configuration metadata allows you to express the objects from your application in XML format.

## 5.6 References :

Reference Books:

- Java 6 Programming Black Book, Wiley–Dreamtech ISBN 10: 817722736X ISBN 13: 9788177227369
- Spring in Action, Craig Walls, 3rd Edition,Manning, ISBN 9781935182351
- Professional Java Development with the Spring Framework by Rod Johnsonet al.John Wiley & Sons 2005 (672 pages) ISBN:0764574833

- Beginning Spring , Mert Calıs kan and KenanSevindik Published by John Wiley & Sons, Inc. 10475 Crosspoint Boulevard Indianapolis, IN 46256

Web References:

- https://www.springframework.net/
- https://www.javadevjournal.com/
- https://www.tutorialspoint.com/
- https://www.geeksforgeeks.org/


## 5.7 Unit End Exercises :

1. Explain the Spring framework with neat labelled diagram.
2. What do you mean by POJO programming model? Give suitable example.
3. List the properties of POJO class.
4. Explain the concept of IoC container.
5. Explain different types of IoC container.
6. Explain the process of configuration of metadata with container.

❖❖❖❖❖

# 6

# SPRING FRAMEWORKS

**Unit Structure**



## 6.1 Objectives

This chapter would make you understand the following concepts:

- Dependency Injection
- Circular Dependency
- Overriding Bean
- Auto Wiring
- Bean Looksup
- Spring manages Beans

## 6.2 Dependency injection:

- Dependency injection (DI) is a process whereby objects define their dependencies, that is, the other objects they work with, only through constructor arguments and properties that are set on the object instance after it is constructed.
- The container injects these dependencies when it creates the object.
- Dependency injection exists in two major variants, Constructor-based dependency injection and Setter-based dependency injection.

### 6.2.1 Constructor-based dependency injection:

Constructor-based DI is accomplished by the container invoking a constructor with a number of arguments, each representing a dependency.

The following example shows a class that can only be dependency-injected with constructor injection.

```
public class SimpleMovieLister
{
  // the SimpleMovieLister has a dependency on a MovieFinder
  private IMovieFinder movieFinder;
  // a constructor so that the Spring container can 'inject' a MovieFinder
  public MovieLister(IMovieFinder movieFinder)
  {
    this.movieFinder = movieFinder;
  }
  // business logic that actually 'uses' the injected IMovieFinder is omitted...
}
```

### 6.2.2 Setter-based dependency injection :

Setter-based DI is accomplished by the container invoking setter properties on your objects after invoking a no-argument constructor or no-argument static factory method to instantiate your object.

The following eample shows a class that can only be dependency injected using pure setter injection.

```
public class MovieLister
{
  private IMovieFinder movieFinder;
  public IMovieFinder MovieFinder
  {
     set
     {
        movieFinder = value;
     }
  }
  // business logic that actually 'uses' the injected IMovieFinder is omitted...
}
```

## 6.3 Circular dependencies:

There are the issue caused during dependency injection when *spring-context* tries to load objects and one bean depends on another bean. Suppose when Object A & B depends on each other.
i.e. A depends on B and vice-versa.
Spring throws UnsatisfiedDependencyException while creating objects of A and B because A object cannot be created until unless B is created and visa-versa.

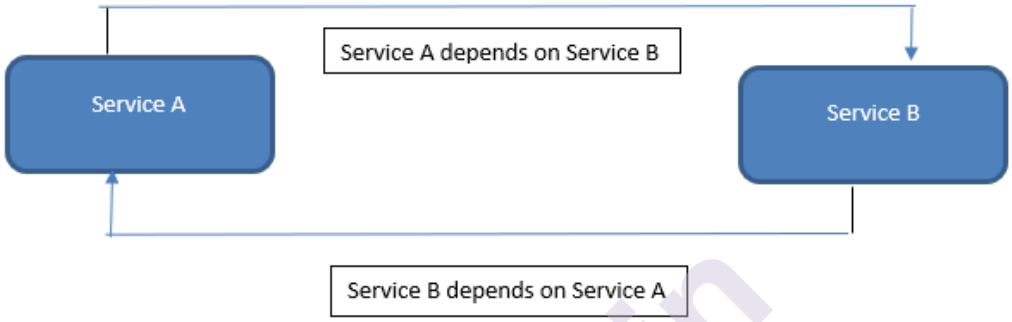



Fig. **Circular dependencies**

Let's understand it using the real code example.

Create two services ServiceA and ServiceB and try to
inject ServiceA into ServiceB and visa-versa as shown in the above picture.

**ServiceA.java**

```
import org.springframework.stereotype.Service;

@Service
public class ServiceA {
	private ServiceB serviceB;
	public ServiceA(ServiceB serviceB) {
		System.out.println("Calling Service A");
		this.serviceB = serviceB;
	}
}
```

**ServiceB.java**

```
import org.springframework.stereotype.Service;

@Service
public class ServiceB {
	private ServiceA serviceA;
	public ServiceB(ServiceA serviceA) {
		System.out.println("Calling Service B");
		this.serviceA = serviceA;
	}
}
```

To simulate the circular dependency issue, run the below class, and see the console log.

**CircularDependenciesTestApp.java**

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
@SpringBootApplication
public class CircularDependenciesTestApp {
	public static void main(String[] args) {
		SpringApplication.run(CircularDependenciesTestApp.class, args);
	}
}
```

When we execute CircularDependenciesTestApp class it won't be able to inject the dependencies due to circular dependencies on each other and will throw a checked exception as shown below:

**console log**

Error starting ApplicationContext. To display the conditions report re-run your application with 'debug' enabled.

2020-05-27 21:22:46.368 ERROR 4480 --- [ main] o.s.b.d.LoggingFailureAnalysisReporter :

**********

APPLICATION FAILED TO START

**********

**Description:**

The dependencies of some of the beans in the application context form a cycle:

```
┌─────┐
|  serviceA defined in file [F:\sts4-workspace\circular-dependencies-spring\target\classes\org\websparrow\service\ServiceA.class]
↑     ↓
|  serviceB defined in file [F:\sts4-workspace\circular-dependencies-spring\target\classes\org\websparrow\service\ServiceB.class]
└─────┘
```

### 6.3.1 How to resolve this issue?

To solve the circular dependency issue, you have two options: Using @Lazy with constructor injection and Using @Autowired along with @Lazy annotation.

#### 6.3.1.1 Using @Lazy with constructor injection

We can lazily initialize ServiceB bean during constructor injection in order to delay constructing ServiceB bean. Here are the code changes in ServiceA for more clarity:

**ServiceA.java**

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

import org.springframework.stereotype.Service;
@Service
public class ServiceA {
	private ServiceB serviceB;
	public ServiceA(@Lazy ServiceB serviceB) {
		System.out.println("Calling Service A");
		this.serviceB = serviceB;
	}
}
```

If you run the CircularDependenciesTestApp class again, you'll find the circular dependency issue is solved.

**console log:**

```
.   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v2.3.0.RELEASE)

2020-05-27 21:33:22.637  INFO 7156 --- [main]
o.w.CircularDependenciesTestApp : Starting CircularDependenciesTestApp on
Atul-PC with PID 7156 (F:\sts4-workspace\circular-dependencies-
spring\target\classes started by user1 in F:\sts4-workspace\circular-dependencies-
spring)
2020-05-27 21:33:22.640  INFO 7156 --- [main]
o.w.CircularDependenciesTestApp  : No active profile set, falling back to default
profiles: default
Calling Service A
Calling Service B
```

2020-05-27 21:33:23.251 INFO 7156 --- [main]
o.w.CircularDependenciesTestApp : Started CircularDependenciesTestApp in 0.98 seconds (JVM running for 1.667)

#### 6.3.1.2 Using @Autowired along with @Lazy annotation

Using @Autowired along with @Lazy annotation for injecting ServiceB in ServiceA.

Let's use these annotations to inject beans and test our application whether it resolves the issue:

**ServiceA.java**

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Lazy;
import org.springframework.stereotype.Service;
@Service
public class ServiceA {
	@Autowired
	@Lazy
	private ServiceB serviceB;
	/*
	public ServiceA(ServiceB serviceB) {
		System.out.println("Calling Service A");
		this.serviceB = serviceB;
	}
	*/
}
```

Here is the output on the console log when you run CircularDependenciesTestApp class again:

**console log:**

```
.   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v2.3.0.RELEASE)
```

2020-05-27 21:45:07.583 INFO 4036 --- [main] o.w.CircularDependenciesTestApp : Starting CircularDependenciesTestApp on Atul-PC with PID 4036 (F:\sts4-workspace\circular-dependencies-spring\target\classes started by user1 in F:\sts4-workspace\circular-dependencies-spring)

2020-05-27 21:45:07.586 INFO 4036 --- [ main] o.w.CircularDependenciesTestApp : No active profile set, falling back to default profiles: default

Calling Service B

2020-05-27 21:45:08.141 INFO 4036 --- [main] o.w.CircularDependenciesTestApp

## 6.4 Overriding Bean:

A bean definition can contain a lot of configuration information, including constructor arguments, property values, and container-specific information such as initialization method, static factory method name, and so on.

A child bean definition inherits configuration data from a parent definition. The child definition can override some values, or add others, as needed.

Spring Bean definition inheritance has nothing to do with Java class inheritance but the inheritance concept is same. You can define a parent bean definition as a template and other child beans can inherit the required configuration from the parent bean.

When you use XML-based configuration metadata, you indicate a child bean definition by using the parent attribute, specifying the parent bean as the value of this attribute.



**Example**

Let us have a working Eclipse IDE in place and take the following steps to create a Spring application −

**Steps Description:**

1. Create a project with a name SpringExample and create a package com.tutorialspoint under the src folder in the created project.
2. Add required Spring libraries using Add External JARs option as explained in the Spring Hello World Example chapter.

3. Create Java classes HelloWorld, HelloIndia and MainApp under the com.tutorialspoint package.
4. Create Beans configuration file Beans.xml under the src folder.
5. The final step is to create the content of all the Java files and Bean Configuration file and run the application as explained below.

Following is the configuration file **Beans.xml** where we defined "helloWorld" bean which has two properties message1 and message2. Next "helloIndia" bean has been defined as a child of "helloWorld" bean by using parent attribute. The child bean inherits message2 property as is, and overrides message1 property and introduces one more property message3.

```
<?xml version = "1.0" encoding = "UTF-8"?>
<beans xmlns = "http://www.springframework.org/schema/beans"
       xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation = "http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
 <bean id = "helloWorld" class = "com.tutorialspoint.HelloWorld">
              <property name = "message1" value = "Hello World!"/>
              <property name = "message2" value = "Hello Second World!"/>
</bean>
<bean id ="helloIndia" class = "com.tutorialspoint.HelloIndia" parent =
"helloWorld">
              <property name = "message1" value = "Hello India!"/>
              <property name = "message3" value = "Namaste India!"/>
       </bean> </beans>
```

Here is the content of **HelloWorld.java** file −

```
public class HelloWorld
{
  private String message1;
  private String message2;
  public void setMessage1(String message){
    this.message1 = message;
  }
  public void setMessage2(String message){
    this.message2 = message;
  }
  public void getMessage1(){
    System.out.println("World Message1 : " + message1);
  }
  public void getMessage2(){
    System.out.println("World Message2 : " + message2);
```

```
   }
}
```

Here is the content of **HelloIndia.java** file −

```
public class HelloIndia
{
   private String message1;
   private String message2;
   private String message3;
   public void setMessage1(String message){
      this.message1 = message;
   }
   public void setMessage2(String message){
      this.message2 = message;
   }
   public void setMessage3(String message){
      this.message3 = message;
   }
   public void getMessage1(){
      System.out.println("India Message1 : " + message1);
   }
   public void getMessage2(){
      System.out.println("India Message2 : " + message2);
   }
   public void getMessage3(){
      System.out.println("India Message3 : " + message3);
   }
}
```

Following is the content of the **MainApp.java** file −

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class MainApp
{
   public static void main(String[] args)
{
      ApplicationContext context = new
ClassPathXmlApplicationContext("Beans.xml");

      HelloWorld objA = (HelloWorld) context.getBean("helloWorld");
      objA.getMessage1();
      objA.getMessage2();
      HelloIndia objB = (HelloIndia) context.getBean("helloIndia");
      objB.getMessage1();
```

```
    objB.getMessage2();
    objB.getMessage3();
  }
}
```

Once you are done creating the source and bean configuration files, let us run the application. If everything is fine with your application, it will print the following message −

World Message1 : Hello World!
World Message2 : Hello Second World!
India Message1 : Hello India!
India Message2 : Hello Second World!
India Message3 : Namaste India!
(Note:If you observed here, we did not pass message2 while creating "helloIndia" bean, but it got passed because of Bean Definition Inheritance.)

## 6.5 Auto Wiring:

The Spring container can autowire relationships between collaborating beans without using <constructor-arg> and <property> elements, which helps cut down on the amount of XML configuration you write for a big Spring-based application.

**Autowiring Modes**

Following are the autowiring modes, which can be used to instruct the Spring container to use autowiring for dependency injection. You use the autowire attribute of the <bean/> element to specify autowire mode for a bean definition.

| Sr.No | Mode | Description |
|---|---|---|
| 1 | no | This is default setting which means no autowiring and you should use explicit bean reference for wiring. You have nothing to do special for this wiring. This is what you already have seen in Dependency Injection chapter. |
| 2 | byName | Autowiring by property name. Spring container looks at the properties of the beans on which autowire attribute is set to byName in the XML configuration file. It then tries to match and wire its properties with the beans defined by the same names in the configuration file. |

| | | |
|---|---|---|
| 3 | byType | Autowiring by property datatype. Spring container looks at the properties of the beans on which autowire attribute is set to byType in the XML configuration file. It then tries to match and wire a property if its type matches with exactly one of the beans name in configuration file. If more than one such beans exists, a fatal exception is thrown. |
| 4 | constructor | Similar to byType, but type applies to constructor arguments. If there is not exactly one bean of the constructor argument type in the container, a fatal error is raised. |
| 5 | autodetect | Spring first tries to wire using autowire by constructor, if it does not work, Spring tries to autowire by byType. |

Note : You can use byType or constructor autowiring mode to wire arrays and other typed-collections.

**Limitations with autowiring:**

Autowiring works best when it is used consistently across a project. If autowiring is not used in general, it might be confusing for developers to use it to wire only one or two bean definitions. Though, autowiring can significantly reduce the need to specify properties or constructor arguments but you should consider the limitations and disadvantages of autowiring before using them.

| **Sr.No.** | **Limitations** | **Description** |
|---|---|---|
| 1 | Overriding possibility | You can still specify dependencies using <constructor-arg> and <property> settings which will always override autowiring. |
| 2 | Primitive data types | You cannot autowire so-called simple properties such as primitives, Strings, and Classes. |

| | | |
|---|---|---|
| 3 | Confusing nature | Autowiring is less exact than explicit wiring, so if possible prefer using explict wiring. |

## 6.6 Bean Looksup:

A method annotated with @Lookup tells Spring to return an instance of the method's return type when we invoke it.
Essentially, Spring will override our annotated method and use our method's return type and parameters as arguments to BeanFactory#getBean.
**@Lookup is useful for:**

- Injecting a prototype-scoped bean into a singleton bean (similar to Provider)
- Injecting dependencies procedurally

### 6.6.1 Injecting prototype-scoped Bean Into a Singleton Bean:

**Step 1: let's create a prototype bean that we will later inject into a singleton bean**

```
@Component
@Scope("prototype")
public class SchoolNotification {
   // ... prototype-scoped state
}
```

**Step 2: create a singleton bean that uses @Lookup**

```
@Component
public class StudentServices {
   // ... member variables, etc.
   @Lookup
   public SchoolNotification getNotification() {
      return null;
   }
   // ... getters and setters
}
```

Using @Lookup, we can get an instance of SchoolNotification through our singleton bean:

**Step 3:**

```
@Test
public void whenLookupMethodCalled_thenNewInstanceReturned() {
   // ... initialize context
   StudentServices first = this.context.getBean(StudentServices.class);
   StudentServices second = this.context.getBean(StudentServices.class);

   assertEquals(first, second);
   assertNotEquals(first.getNotification(), second.getNotification());
}
```

(Note that in StudentServices, we left the getNotification method as a stub)

### 6.6.2 Injecting Dependencies Procedurally

Let's enhance StudentNotification with some state:

**Step 1:**

```
@Component
@Scope("prototype")
public class SchoolNotification {
   @Autowired Grader grader;
   private String name;
   private Collection<Integer> marks;
   public SchoolNotification(String name) {
      // ... set fields
   }
   // ... getters and setters
   public String addMark(Integer mark) {
      this.marks.add(mark);
      return this.grader.grade(this.marks);
   }
}
```

Now, it is dependent on some Spring context and also additional context that we will provide procedurally.

**Step 2:We can then add a method to StudentServices that takes student data and persists it.**

```
public abstract class StudentServices {
    private Map<String, SchoolNotification> notes = new HashMap<>();
   @Lookup
   protected abstract SchoolNotification getNotification(String name);
   public String appendMark(String name, Integer mark) {
      SchoolNotification notification
       = notes.computeIfAbsent(name, exists -> getNotification(name)));
      return notification.addMark(mark);
   }
}
```

At runtime, Spring will implement the method in the same way, with a couple of additional tricks.

**Please note:**

- ✓ It can call a complex constructor as well as inject other Spring beans, allowing us to treat SchoolNotification a bit more like a Spring-aware method.It does this by implementing getSchoolNotification with a call to beanFactory.getBean(SchoolNotification.class, name).
- ✓ Second, we can sometimes make the @Lookup-annotated method abstract, like the above example.

```
@Test
public void whenAbstractGetterMethodInjects_thenNewInstanceReturned() {
   // ... initialize context
   StudentServices services = context.getBean(StudentServices.class);
   assertEquals("PASS", services.appendMark("Alex", 89));
   assertEquals("FAIL", services.appendMark("Bethany", 78));
   assertEquals("PASS", services.appendMark("Claire", 96));
}
```

With this setup, we can add Spring dependencies as well as method dependencies to SchoolNotification.

## 6.7 Spring manages Beans:

A bean is the foundation of a Spring-managed application; all beans reside withing the IOC container, which is responsible for managing their life cycle.

We can get a list of all beans within this container in two ways:

1. Using a ListableBeanFactory interface
2. Using a Spring Boot Actuator

### 6.7.1 Using a ListableBeanFactory interface

The ListableBeanFactory interface
provides getBeanDefinitionNames() method which returns the names of all the beans defined in this factory. This interface is implemented by all the bean factories that pre-loads their bean definitions to enumerate all their bean instances.

You can find the list of all known subinterfaces and its implementing classes in the official documentation.

For this example, we'll be using a Spring Boot Application.

First, we'll create some Spring beans. Let's create a simple Spring Controller FooController:

```
@Controller
public class FooController {
@Autowired
   private FooService fooService;
@RequestMapping(value="/displayallbeans")
public String getHeaderAndBody(Map model){
model.put("header", fooService.getHeader());
model.put("message", fooService.getBody());
return "displayallbeans";
} }
```

**This Controller is dependent on another Spring bean FooService:**

```
@Service
public class FooService
{       public String getHeader()
{       return "Display All Beans";
        }
 public String getBody()
{       return "This is a sample application that displays all beans " + "in Spring IoC container using ListableBeanFactory interface " + "and Spring Boot Actuators.";
        }
}
```

**Note that we've created two different beans here:fooController and fooService**

**While executing this application, we'll use applicationContext object and call its getBeanDefinitionNames() method, which will return all the beans in our applicationContext container:**

```
@SpringBootApplication
public class Application
{       private static ApplicationContext applicationContext;
public static void main(String[] args) {
        applicationContext = SpringApplication.run(Application.class, args);
        displayAllBeans();
        }
       public static void displayAllBeans() {
       String[] allBeanNames = applicationContext.getBeanDefinitionNames();
       for(String beanName : allBeanNames) {
          System.out.println(beanName);
       }
   }
}
```

**This will print all the beans from applicationContext container:**

```
fooController
fooService
//other beans
```

### 6.7.2 Using Spring Boot Actuator

The Spring Boot Actuator functionality provides endpoints which are used for monitoring our application's statistics.

It includes many built-in endpoints, including */beans.* This displays a complete list of all the Spring managed beans in our application. You can find the full list of existing endpoints over on the official docs.

**Now, we'll just hit the URL *http://<address>:<management-port>/beans.* We can use our default server port if we haven't specified any separate management port. This will return a *JSON* response displaying all the beans within the Spring IoC Container:**

You will get output like this:

```
[ { "context": "application:8080", "parent": null, "beans": [ { "bean":
"fooController", "aliases": [], "scope": "singleton", "type":
"com.baeldung.displayallbeans.controller.FooController", "resource": "file
[E:/Workspace/tutorials-master/spring-boot/target
/classes/com/baeldung/displayallbeans/controller/FooController.class]",
```

"dependencies": [ "fooService" ] }, { "bean": "fooService", "aliases": [], "scope": "singleton", "type": "com.baeldung.displayallbeans.service.FooService", "resource": "file [E:/Workspace/tutorials-master/spring-boot/target/classes/com/baeldung/displayallbeans/service/FooService.class]", "dependencies": [] }, // ...other beans ] } ]

## 6.8 Summary:

- Dependency injection (DI) is a process whereby objects define their dependencies, that is, the other objects they work with.
- Constructor-based DI is accomplished by the container invoking a constructor.
- Setter-based DI is accomplished by the container invoking setter properties on your objects.
- During dependency injection when *spring-context* tries to load objects and one bean depends on another bean called as circular dependency.
- A child bean definition inherits configuration data from a parent definition. The child definition can override some values, or add others, as needed.
- Spring container can autowire relationships between collaborating beans without using <constructor-arg> and <property> elements.
- A method annotated with @Lookup tells Spring to return an instance of the method's return type when we invoke it.

## 6.9 References :

Reference Books:

- Java 6 Programming Black Book, Wiley–Dreamtech ISBN 10: 817722736X ISBN 13: 9788177227369
- Spring in Action, Craig Walls, 3rd Edition,Manning, ISBN 9781935182351
- Professional Java Development with the Spring Framework by Rod Johnsonet al.John Wiley & Sons 2005 (672 pages) ISBN:0764574833
- Beginning Spring , Mert Calıs kan and KenanSevindik Published by John Wiley & Sons, Inc. 10475 Crosspoint Boulevard Indianapolis, IN 46256

Web References:

- https://www.springframework.net/
- https://www.javadevjournal.com/
- https://www.tutorialspoint.com/
- https://www.geeksforgeeks.org/

## 6.10 Unit End Exercises :

1. Explain the concept of Dependency injection (DI).
2. Explain the different types of Dependency injection (DI).
3. What do you mean by circular dependency? Also give solution to issue raised in this.
4. How to implement overriding concepts in spring bean? Explain in detail with suitable example.
5. Write a note on Autowiring.
6. Explain the concept of @Lookup with example.
7. Explain how spring manages a bean in detail.

❖❖❖❖❖

# 7

# SPRING AND AOP

**Unit Structure**



## 7.0 Objectives

After going through this unit, you will be able to:

- what is spring AOP?
- how AOP different from OOPS
- terminology and concepts of AOP with example and advantages and disadvantages of spring AOP, types of advices.
- Introduction of AspectJ, spring boot AOP

## 7.1 Introduction

Aspect Oriented Programming (AOP) complements Object-Oriented Programming (OOP) by providing another way of thinking about program structure. The key unit of modularity in OOPis the class, whereas in AOP the unit of modularity is the aspect. Aspect Oriented Pro-gramming AspectJ integrated with Spring AOP provides very powerful mechanisms for stronger enforcement of security.Aspect-oriented programming (AOP) allows weaving a security aspect into an application providing additional security functionality or introducing completely new security mechanisms.Implementation of security with AOP is a flexible method to develop sepa-rated, extensible and reusable pieces of code called aspects.In this comparative study paper, we argue that Spring AOP pro-vides stronger enforcement of security than AspectJ

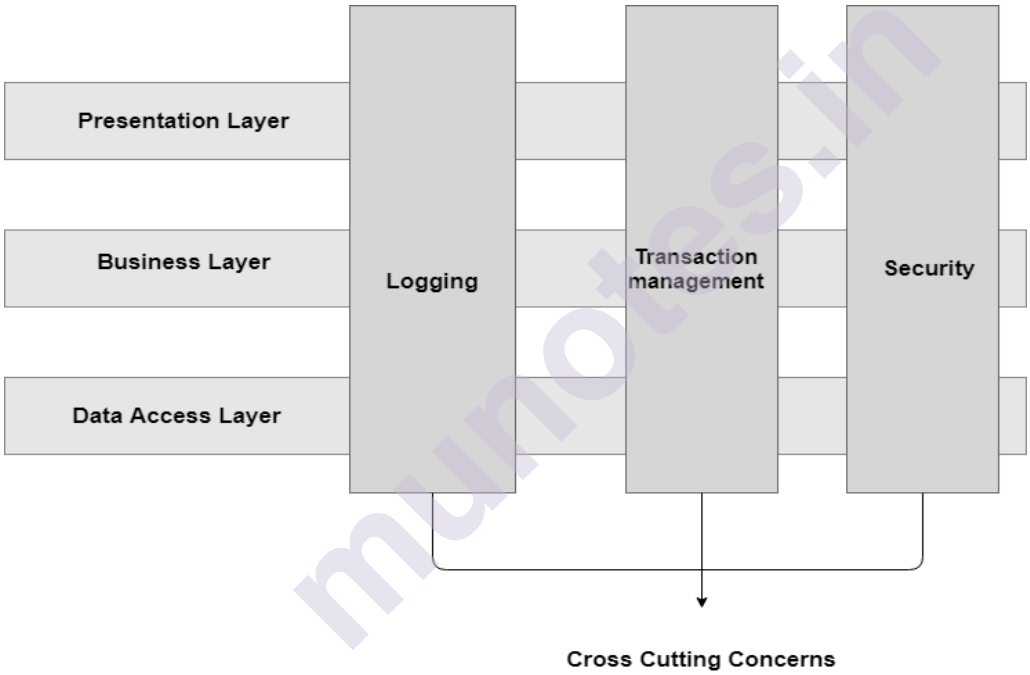
## 7.2 An Overview

### 7.2. 1. What is Spring AOP (Aspect Oriented Programming) and about AspectJ?

*Aspect-oriented programming* (AOP) is one of the major components of the Spring Framework. The Spring AOP helps in breaking down the logic of the program into several distinct parts called as concerns. Cross-cutting concerns is the functions which span multiple points of an application.

The **cross-cutting concerns** help in increasing the modularity and separate it from the business logic of an application. Also, a cross-cutting is a concern that affects the whole application and it should be centralized in one location in code as possible such as authentication, transaction management, logging etc.

Below diagram shows how the concerns like logging, **security**, and **transaction management** are cutting across different layer here:






**Examples of cross-cutting concerns**

AOP is a complement of OOP (Object Oriented Programming) and they can be used together to write powerful applications because both provide different ways of structuring your code. OOP is focused on making everything an object, while AOP introduces the aspect, which is a special type of object that injects and wraps its behavior to complement the behavior of other objects.

Examples of cross-cutting concerns:

- Logging
- Security
- Transaction management
- Auditing,

- Caching
- Internationalization
- Error detection and correction
- Memory management
- Performance monitoring
- Synchronization

We can implement Spring Framework AOP in pure Java, it doesn't require to the special compilation process. It is the best choice for use in a J2EE web container or application server because it doesn't require to control the class loader hierarchy.

**I. Need for Spring AOP**

The Spring AOP provides the pluggable way of dynamically adding the additional concern before, after or around the actual logic.

Consider there are 10 methods in the class as defined below:

```
class A{
public void m1(){...}
public void m2(){...}
public void m3(){...}
public void m4(){...}
public void m5(){...}
public void n1(){...}
public void n2(){...}
public void p1(){...}
public void p2(){...}
public void p3(){...}
}
```



These are the 5 methods that start from m while 2 methods start from n and 3 methods starting from p.

- **The Scenario:** You have to maintain the log and send the notification after calling methods starting from m.
- **A problem without using the AOP:** You can call methods which maintain logs and sends the notification from the methods starting with m. For that, you need to write code for all the 5 methods.
- **The solution with Aspect Oriented Programming:** You don't have to call methods from the method. You can define additional concerns like maintaining a log, sending notification etc. as a method of a class.

## II. Where to use Spring Aspect Oriented Programming

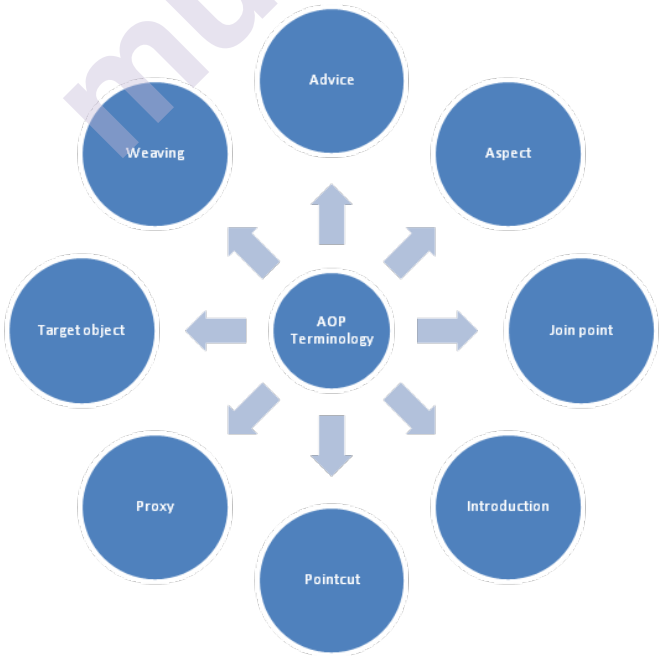
Some of the cases where AOP is frequently used: To provide declarative enterprise services. For example, as declarative transaction management. It allows users for implementing custom aspects

*** About AspectJ**

The important aspect of Spring is the Aspect-Oriented Programming (AOP) framework. As we all know, the key unit of modularity in OOP(Object Oriented Programming) is the class, similarly, in AOP the unit of modularity is the aspect. Aspects enable the modularization of concerns such as transaction management that cut across multiple types and objects. To implement these concerns, AspectJ comes into the picture. AspectJ, a compatible extension to the Java programming language, is one implementation of AOP. It has grown into a complete and popular AOP framework. Since AspectJ annotations are supported by more and more AOP frameworks, AspectJ-style aspects are more likely to be reused in other AOP frameworks that support AspectJ.

**AspectJ** is an original library that provided components for creating aspects is named AspectJ. It was developed by the Xerox PARC company and released in 1995. It defined a standard for AOP because of its simplicity and usability. The language syntax used to define aspects was similar to Java and allowed developers to define special constructs called aspects. The aspects developed in AspectJ are processed at compile time, so they directly affect the generated bytecode. Read more about AspectJ at **https://eclipse.org/aspectj/**

### 7.2.2 Concept and Terminology used in AOP



- **Aspect**: A modularization of a concern that cuts across multiple objects.
- **Join point**: A point during the execution of a pro- gram, such as the execution of a method or the handling of an exception. In Spring AOP, a join point always represents a method execution. Join point information is available in advice bodies by declaring org.aspectj.lang.JoinPoint parameter type
- **Advice**: Action taken by an aspect at a particular join point. Different types of advice include ″around,″ ″before″ and ″after″ advice. Many AOP frameworks, including Spring, model an advice as an interceptor, maintaining a chain of interceptors “around″ the join point.
- **Pointcut**: A predicate that matches join points. Advice is associated with a pointcut expression and runs at any join point matched by the point- cut (for example, the execution of a method with a certain name).The concept of join points as matched by pointcut expressions is central to AOP.Spring uses the AspectJ pointcut language by default.
- **Introduction**: Also known as an inter-type declaration.Declaring additional methods or fields on behalf of a type. Spring AOP allows introducing new interfaces and a corresponding implementa tion to any proxied object.
- **Target object**: Object being advised by one or more aspects. Also referred to as the advised ob- ject. Since Spring AOP is implemented using run- time proxies, this object will always be a proxied object.
- **Weaving**: Linking aspects with other application types or objects to create an advised object. This can be done at compile time (using the AspectJ compiler, for example), load time, or at runtime. Spring AOP, like other pure Java AOP frame- works, performs weaving at runtime.
- Proxy: It is used to implement aspect contracts, created by AOP framework. It will be a JDK dynamic proxy or CGLIB proxy in spring framework.

Above AOP terminology is not very inherent so I will explain with creating an example application**(Spring AOP + Aspectj)** and then relate the terminology with usage in the example.

Create a spring boot application with spring AOP.

**Note : “Spring Boot is a project that is built on the top of the Spring Framework. It provides an easier and faster way to set up, configure, and run both simple and web-based applications”.**

There are many ways to create a Spring Boot application. These are following:

>> Create Spring Boot Project With Spring Initializer

>> Create Spring Boot Project in Spring Tool Suite [STS]

- first, we need Maven Dependencies. In actual, you need to create a maven project and add all the dependencies. If you wish to learn how to configure Spring Framework you can refer to this **Spring Framework Tutorial**
- To create a Maven Project, install **Eclipse for JEE developers** and follow these steps.
- *Click on File -> New -> Other-> Maven Project -> Next-> Choose maven-archetype-quickstart-> Specify GroupID -> Artifact ID -> Package name and then click on finish.*
- Once you create a Maven Project, next thing that you have to do is to add maven dependencies in the *pom.xml* file.
- Your pom.xml file should consist of the below-mentioned dependencies for Aspect Oriented Programming.

Add Spring AOP starter to maven project pom.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<project
   xmlns="http://maven.apache.org/POM/4.0.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
   <modelVersion>4.0.0</modelVersion>
   <groupId>springboot2</groupId>
   <artifactId>springboot2-springaop-example</artifactId>
   <version>0.0.1-SNAPSHOT</version>
   <packaging>jar</packaging>
   <name>springboot2-springaop-example</name>
   <description>Demo project for Spring Boot</description>
   <parent>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-parent</artifactId>
      <version>2.1.4.RELEASE</version>
      <relativePath />
      <!-- lookup parent from repository -->
   </parent>
   <properties>
      <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
      <project.reporting.outputEncoding>UTF-
8</project.reporting.outputEncoding>
      <java.version>1.8</java.version>
   </properties>
```

```
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-aop</artifactId>
        </dependency>
    </dependencies>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
</project>
```

**Employee.java**

Create a simple Employee POJO class (You can use it as JPA entity for database operations):

```
public class Employee {
    private long id;
    private String firstName;
    private String lastName;
    private String emailId;
    public Employee() {
    }
    public Employee(long id, String firstName, String lastName, String emailId) {
        this.id = id;
        this.firstName = firstName;
        this.lastName = lastName;
        this.emailId = emailId;
    }
    public long getId() {
        return id;
```

```
    }
    public void setId(long id) {
        this.id = id;
    }
    public String getFirstName() {
        return firstName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
    public String getLastName() {
        return lastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
    public String getEmailId() {
        return emailId;
    }
    public void setEmailId(String emailId) {
        this.emailId = emailId;
    }
    @Override
    public String toString() {
        return "Employee [id=" + id + ", firstName=" + firstName + ", lastName=" +
lastName + ", emailId=" + emailId +
                "]";
    }
}
```

**EmployeeService.java**

To keep it simple, I will create an EmployeeService and manage in-memory objects:

```
import java.util.ArrayList;
import java.util.List;
import org.springframework.stereotype.Service;
/**
 * Employee Service
 *
 *
 */
@Service
public class EmployeeService {
    private List < Employee > employees = new ArrayList < > ();
    public List < Employee > getAllEmployees() {
        System.out.println("Method getAllEmployees() called");
        return employees;
    }
    public Employee getEmployeeById(Long employeeId) {
        System.out.println("Method getEmployeeById() called");
        for (Employee employee: employees) {
            if (employee.getId() == Long.valueOf(employeeId)) {
                return employee;
            }
        }
        return null;
    }
    public void addEmployee(Employee employee) {
        System.out.println("Method addEmployee() called");
        employees.add(employee);
    }
    public void updateEmployee(Employee employeeDetails) {
        System.out.println("Method updateEmployee() called");
        for (Employee employee: employees) {
            if (employee.getId() == Long.valueOf(employeeDetails.getId())) {
                employees.remove(employee);
                employees.add(employeeDetails);
            }
```

```
        }
    }
    public void deleteEmployee(Long employeeId) {
        System.out.println("Method deleteEmployee() called");
        for (Employee employee: employees) {
            if (employee.getId() == Long.valueOf(employeeId)) {
                employees.remove(employee);
            }
        }
    }
}
```

**LoggingAspect.java**

Now, let's create a LogginAspect class:

```
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.After;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Component;
/**
 * Aspect for logging execution.
 *
 *
 */
@Aspect
@Component
public class LoggingAspect {
    private final Logger LOGGER = LoggerFactory.getLogger(this.getClass());
    @Before("execution(*springboot2.springaop.service.EmployeeService.*(..))")
    public void logBeforeAllMethods(JoinPoint joinPoint) {
  LOGGER.debug("****LoggingAspect.logBeforeAllMethods() : " +
joinPoint.getSignature().getName());
    }
```

```
@Before("execution(*springboot2.springaop.service.EmployeeService.getEmployeeById(..))")
    public void logBeforeGetEmployee(JoinPoint joinPoint) {
  LOGGER.debug("****LoggingAspect.logBeforeGetEmployee() : " + joinPoint.getSignature().getName());
    }

@Before("execution(*springboot2.springaop.service.EmployeeService.createEmployee(..))")
    public void logBeforeAddEmployee(JoinPoint joinPoint) {
  LOGGER.debug("****LoggingAspect.logBeforeCreateEmployee() : " + joinPoint.getSignature().getName());
    }

    @After("execution(* springboot2.springaop.service.EmployeeService.*(..))")
    public void logAfterAllMethods(JoinPoint joinPoint)
    {
  LOGGER.debug("****LoggingAspect.logAfterAllMethods() : " + joinPoint.getSignature().getName());
    }

    @After("execution(*
springboot2.springaop.service.EmployeeService.getEmployeeById(..))")
    public void logAfterGetEmployee(JoinPoint joinPoint)
    {
     LOGGER.debug("****LoggingAspect.logAfterGetEmployee() : " + joinPoint.getSignature().getName());
    }


@After("execution(*springboot2.springaop.service.EmployeeService.addEmployee(..))")
    public void logAfterAddEmployee(JoinPoint joinPoint)
    {
     LOGGER.debug("****LoggingAspect.logAfterCreateEmployee() : " + joinPoint.getSignature().getName());
    }
}
```

### Application.java

Now test the AOP configuration and other stuff with main() method:

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;
import springboot2.springaop.model.Employee;
import springboot2.springaop.service.EmployeeService;
@SpringBootApplication
public class Application {

 public static void main(String[] args) {
  ApplicationContext applicationContext = SpringApplication.run(Application.class, args);
  EmployeeService employeeService = applicationContext.getBean(EmployeeService.class);
  employeeService.addEmployee(new Employee(100L, "ramesh", "fadatare", "ramesh@gmail.com"));
  employeeService.getEmployeeById(100L);
  employeeService.getAllEmployees();
 }
}
```

### Output

## Let us understand AOP Concepts and Terminology with Above Example

Aspect

Aspect is modularization of a concern that cuts across multiple classes. Transaction management is a good example of a crosscutting concern in enterprise Java applications. In Spring AOP, aspects are implemented by using regular classes (the schema-based approach) or regular classes annotated with the *@Aspect* annotation (the @AspectJ style).

In our example, we have created a *LoggingAspect* using Java-based configuration. To create an aspect, you need to apply *@Aspect* annotation on Spring component:

```
@Aspect
@Component
public class LoggingAspect {
   ...
}
```

**Join point**

Join point is a point during the execution of a program, such as the execution of a method or the handling of an exception. In Spring AOP, a join point always represents a method execution.

In our example, all the methods defined inside *EmployeeService* are joint points.

Advice

Advice is an action taken by an aspect at a particular join point.

In our example, *logBeforeAllMethods()*, *logBeforeGetEmployee()*, *logBeforeAddEmployee()*, *logAfterAllMethods()*, *logAfterGetEmployee()*, and *logAfterAddEmployee()* methods are advices.

Pointcut

A Pointcut is a predicate that helps match an Advice to be applied by an Aspect at a particular JoinPoint. The Advice is often associated with a Pointcut expression and runs at any Joinpoint matched by the Pointcut.

In our example, the expressions passed in @Before and @After annotations are pointcuts. For example:

```
@Before("execution(*springboot2.springaop.service.EmployeeService.*(..))")

@After("execution(* springboot2.springaop.service.EmployeeService.*(..))")
```

Target object
An object being advised by one or more aspects. Also referred to as the "advised object". Since **Spring AOP** is implemented by using runtime proxies, this object is always a proxied object.

In our example, *EmployeeService* is advised object hence it is the target object.

**AOP proxy**

An object created by the AOP framework in order to implement the aspect contracts (advise method executions and so on). In the Spring Framework, an AOP proxy is a JDK dynamic proxy or a CGLIB proxy.

In our example, a proxy object is created when we ask the bean reference for *EmployeeService* class.

### 7.2.3 Advantages and disadvantages of spring AOP

**Advantages of Spring AOP**

1. AOP is non-invasive:
   - Service or Domain classes get advice by the aspects (cross-cutting concerns) without adding Spring AOP related classes or interfaces into the service or domain classes.
   - Allows the developers to concentrate on the business logic, instead of the cross-cutting concerns.
2. AOP is implemented in pure Java:
   - There is no need for a special compilation unit or special class loader
3. It uses Spring's IOC container for dependency injection:
   - Aspects can be configured as normal spring beans.
4. Like any other AOP framework, it weaves cross-cutting concerns into the classes, without making a call to the cross-cutting concerns from those classes.
5. Centralizes or modularizes the cross-cutting concerns:
   - Easy to maintain and make changes to the aspects.
   - Changes only need to be made in one place.
6. Provision to create aspects using schema-based (XML configuration) or @AspectJ annotation based style.
7. Easy to configure.

**Disadvantages of Spring AOP**

1. A small difficulty is debugging the AOP framework-based application code.
   - Since the business classes are advised after the scene with aspects.
2. Since it uses proxy-based AOP, only method-level advising is supported; it does not support field-level interception
   - So join-points can be at method level not at field level in a class.
3. Only methods with public visibility will be advised:
   - Methods with private, protected, or default visibility will not be advised.
4. There's small runtime overhead, but its negotiable:
   - The overhead is in nano-seconds.
5. Aspects cannot advise other Aspects - it's not possible to have aspects as targets of advice from other aspects.
   - Because once you mark one class as an aspect (either use XML or annotation), Spring excludes it from being auto-proxied.
6. Local or internal method calls within an advised class don't get intercepted by proxy, so the advice method of the aspect does not get fired or invoked.
7. It is not for advising fine-grained objects (or domain objects)—it is best suitable for coarse-grained objects due to performance.

### 7.2.4 Types of advices

There are different types of advices:

- **Before advice**: Advice that executes before a join point, but which does not have the ability to prevent execution flow proceeding to the join point (unless it throws an exception).

Before advice is declared in an aspect using the @Before annotation:

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
@Aspect
public class BeforeExample {
  @Before("com.xyz.myapp.SystemArchitecture.dataAccessOperation()")
  public void doAccessCheck() {
    // ...
  }
}
```

- **After returning advice*:*** Advice to be executed after a join point completes normally: for example, if a method returns without throwing an exception.

It is declared using the @AfterReturning annotation:

```
import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.AfterReturning;

@Aspect

public class AfterReturningExample {


@AfterReturning("com.xyz.myapp.SystemArchitecture.dataAccessOperation()")

  public void doAccessCheck() {

    // ...

  }
}
```

- **After throwing advice**: Advice to be executed if a method exits by throwing an exception.

It is declared using the @AfterThrowing annotation:

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.AfterThrowing;
@Aspect
public class AfterThrowingExample {

@AfterThrowing("com.xyz.myapp.SystemArchitecture.dataAccessOperation()")
  public void doRecoveryActions() {
    // ...
  }
}
```

- **After (finally) advice**: Advice to be executed regardless of the means by which a join point exits (normal or exceptional return).

It is declared using the @After annotation. After advice must be prepared to handle both normal and exception return conditions. It is typically used for releasing resources, etc.

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.After;
@Aspect
public class AfterFinallyExample {
 @After("com.xyz.myapp.SystemArchitecture.dataAccessOperation()")
 public void doReleaseLock() {
  // ...
 }
}
```

- **Around advice**: Advice that surrounds a join point such as a method invocation. This is the most powerful kind of advice. Around advice can perform custom behavior before and after the method invocation. It is also responsible for choosing whether to proceed to the join point or to shortcut the advised method execution by returning its own return value or throwing an exception.

Example:

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.ProceedingJoinPoint;
@Aspect
public class AroundExample {
 @Around("com.xyz.myapp.SystemArchitecture.businessService()")
 public Object doBasicProfiling(ProceedingJoinPoint pjp) throws Throwable {
  // start stopwatch
  Object retVal = pjp.proceed();
  // stop stopwatch
  return retVal;
 }
}
```

### 7.2.5 Definition of Pointcut Designator, Pointcut Annotations

- **Pointcut Designator:** A pointcut expression starts with a **pointcut designator (PCD)**, which is a keyword telling Spring AOP what to match. There are several pointcut designators, such as the execution of a method, a type, method arguments, or annotations.

  **execution:** This is used to match method execution (join-points). This is a primary designator, and is used most of the time while working with Spring AOP.

  **within:** This designator has the limitation of matching of join-points within certain types only. It's not as flexible as execution. For example, it's not allowed to specify return types or method parameter mapping. If the patterns with within are up to the Java package, it matches all methods of all classes within that package. If the pattern is pointing to a specific class, then this designator will cover all.

- **Pointcut Annotations:** Using annotations is more convenient than using patterns. While patterns might be anything between a big cannon and a scalpel the annotations are definitely a scalpel, by only getting the pointcut that the developer has manually specified.

  **@Aspect** − Mark a class as a class containing advice methods.

  **@PointCut** − Mark a function as a PointCut.

## 7.3 Summary

In this Spring AOP chapter, you learned about the Aspect-oriented programming in Spring Framework. You also saw the need for using Spring AOP, where to use it. Along with that, you saw its terminologies ,advantages and disadvantages of spring AOP the types of advice with help of examples, introduction of AspectJ, spring boot . Still, you had a doubt go through references and bibliography

## 7.4 Exercise :-

Q 1. What is Spring AOP? What is its use
Q 2. What are the different implementations of Spring AOP ?
Q 3. Explain different AOP terminologies with the help of example?
Q 4. What are the different types of Spring Advice ?
Q 5. What are the Spring AOP advantages and disadvantages?
Q 6. When to use Spring AOP?

Q 7. What is Aspect in Spring AOP?
Q 8. How to declare a Spring AOP Aspect?
Q 9. What is Advice in Spring AOP?
Q 10. What are different Spring AOP Advice types?
Q 11. How to declare an Advice in Spring AOP?
Q 12. What is “execute” in Pointcut expression?
Q 13. What are other PointCut Designators?
Q 14. Define Pointcut designator and annotations with types

## 7.5 References:-


1. Kotrappa Sirbi, Prakash Jayanth Kulkarni Stronger Enforcement of Security Using AOP & Spring AOP
2. Ravi Kumar, Dalip and Munishwar Rai A Comparative Study of AOP Approaches:AspectJ, Spring AOP, JBoss AOP
3. https://www.javaguides.net/2019/05/understanding-spring-aop-concepts-and-terminology-with-example.html
4. Spring https://docs.spring.io/spring-framework/docs/2.5.x/reference/aop.html
5. journaldev https://www.journaldev.com/2583/spring-aop-example-tutorial-aspect-advice- pointcut-joinpoint-annotations
6. Edureka https://www.edureka.co/blog/spring-aop-tutorial/
7. Dzone https://dzone.com/
8. Data flair traning https://data-flair.training/blogs/spring-aop-tutorial/
9. Javatpoint https://www.javatpoint.com/spring-boot-aop
10. How to do java https://howtodoinjava.com/spring-aop-tutorial/
11. Tutorialspoint https://www.tutorialspoint.com/springaop/springaop_implementations.htm
12. Mkyong https://mkyong.com/spring/spring-aop-examples-advice/
13. Baeldung https://www.baeldung.com/spring-aop
14. Javaguides https://www.javaguides.net/2019/04/create-spring-boot-project-with-spring-initializer.html


❖❖❖❖❖

# 8

# JDBC DATA ACCESS WITH SPRING

**Unit Structure**



## 8.0 OBJECTIVES

JDBC stands for Java Database Connectivity. Driver play role like to move an object from one place to another. Vehicle drivers are playing role to move vehicle as well objects whose included inside the vehicles from one place to another. JDBC APIs are used to access virtually any kind of data source from anywhere. JDBC is one type of API which connect and execute the query with the database. JDBC is part of JAVA SE (Java Standard Edition). JDBC API uses JDBC drivers to connect with different types of databases.

# 8.1 INTRODUCTION

JDBC Drivers are used to manipulate data from database with the help of java platform. JDBC perform all types of SQL operations with java. JDBC have its five different types of drivers as follows:

a. JDBC-ODBC Bridge
b. Native Driver
c. Network Protocol Driver
d. Thin Driver
e. Highly-Functional Driver

### 10.1.1 Type 1 driver- JDBC-ODBC Bridge:

- The JDBC type 1 driver, also known as the **JDBC-ODBC bridge**, is a database driver implementation that employs the ODBC driver to connect to the database. The driver converts JDBC method calls into ODBC function calls.
- The driver is platform-dependent as it makes use of ODBC which in turn depends on native libraries of the underlying operating system the JVM is running upon. Also, use of this driver leads to other installation dependencies; for example, ODBC must be installed on the computer having the driver and the database must support an ODBC driver. The use of this driver is discouraged if the alternative of a pure-Java driver is available. The other implication is that any application using a type 1 driver is non-portable given the binding between the driver and platform. This technology isn't suitable for a high-transaction environment. Type 1 drivers also don't support the complete Java command set and are limited by the functionality of the ODBC driver.
- Sun(now Oracle) provided a JDBC-ODBC Bridge driver: `sun.jdbc.odbc.JdbcOdbcDriver`. This driver is native code and not Java, and is closed source. Sun's/Oracle's JDBC-ODBC Bridge was removed in Java 8 (other vendors' are available).
- If a driver has been written so that loading it causes an instance to be created and also calls `DriverManager.registerDriver` with that instance as the parameter, then it is in the DriverManager's list of drivers and available for creating a connection.

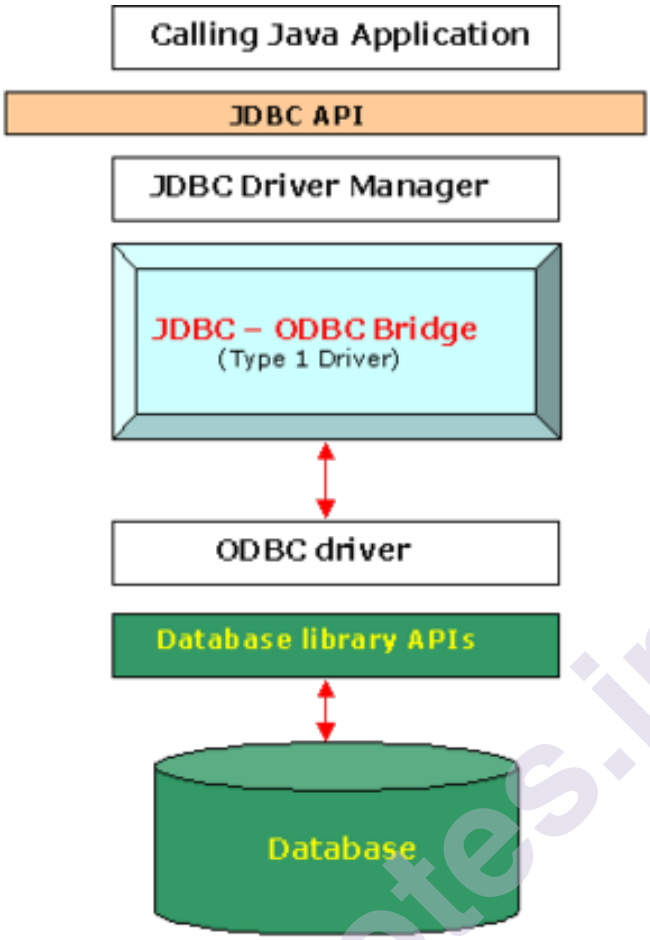



Figure 8.1 shows architecture of Type 1 Driver (JDBC-ODBC Bridge)

**Advantages**

- Almost any database for which an ODBC driver is installed can be accessed, and data can be retrieved.

**Disadvantages**

Performance overhead since the calls have to go through the JDBC( java database connectivity) bridge to the ODBC(open database connectivity) driver, then to the native database connectivity interface (thus may be slower than other types of drivers).

- The ODBC driver needs to be installed on the client machine.
- Not suitable for applets, because the ODBC driver needs to be installed on the client.
- Specific ODBC drivers are not always available on all platforms; hence, portability of this driver is limited.
- No support from JDK 1.8 (Java 8).

### 10.1.2 Type 2 driver – Native-API driver:

- The JDBC type 2 driver, also known as the **Native-API driver**, is a database driver implementation that uses the client-side libraries of the database. The driver converts JDBC method calls into native calls of the database API. For example: Oracle OCI driver is a type 2 driver.

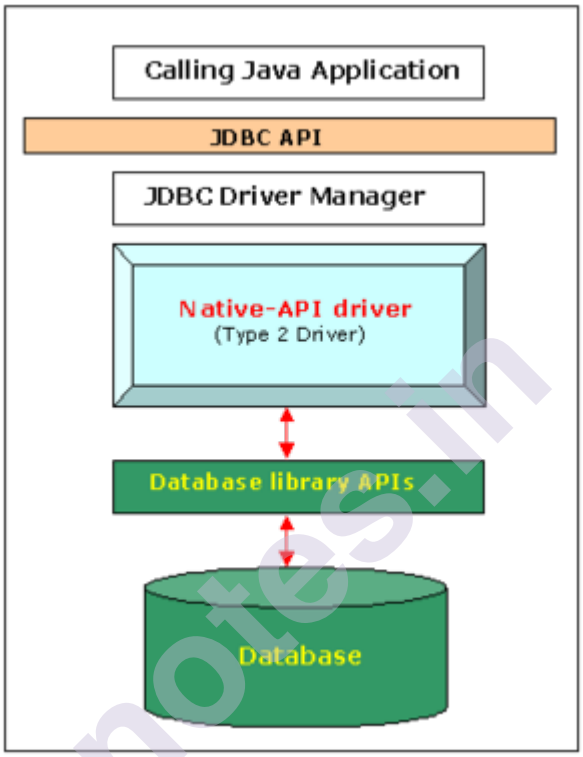



Figure 8.2 shows architecture of Type 2 Driver (Native API Driver)

**Advantages**

- As there is no implementation of JDBC-ODBC bridge, it may be considerably faster than a Type 1 driver.

**Disadvantages**

- The vendor client library needs to be installed on the client machine.
- Not all databases have a client-side library.
- This driver is platform dependent.
- This driver supports all Java applications except applets.

### 10.1.3 Type 3 driver – Network-Protocol driver (middleware driver):

- The JDBC type 3 driver, also known as the Pure Java driver for database middleware, is a database driver implementation which makes use of a middle tier between the calling program and the database. The middle-tier

(application server) converts JDBC calls directly or indirectly into a vendor-specific database protocol.

- This differs from the type 4 driver in that the protocol conversion logic resides not at the client, but in the middle-tier. Like type 4 drivers, the type 3 driver is written entirely in Java.
- The same client-side JDBC driver may be used for multiple databases. It depends on the number of databases the middleware has been configured to support. The type 3 driver is platform-independent as the platform-related differences are taken care of by the middleware. Also, making use of the middleware provides additional advantages of security and firewall access.

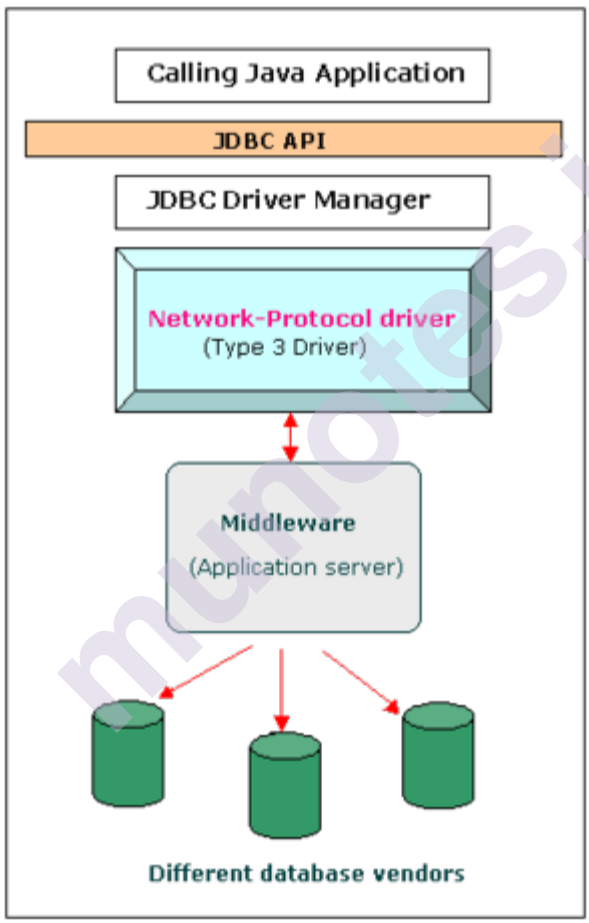



Figure 8.3 shows architecture of Type 3 Driver (Network-Protocol Driver)

**Functions:**

- Sends JDBC API calls to a middle-tier net server that translates the calls into the DBMS-specific network protocol. The translated calls are then sent to a particular DBMS.
- Follows a three-tier communication approach.

- Can interface to multiple databases – Not vendor specific.
- The JDBC Client driver written in java, communicates with a middleware-net-server using a database independent protocol, and then this net server translates this request into database commands for that database.
- Thus, the client driver to middleware communication is database independent.

**Advantages:**

- Since the communication between client and the middleware server is database independent, there is no need for the database vendor library on the client. The client need not be changed for a new database.
- The middleware server (which can be a full-fledged J2EE Application server) can provide typical middleware services like caching (of connections, query results, etc.), load balancing, logging, and auditing.
- A single driver can handle any database, provided the middleware supports it.
- E.g.: IDA Server

**Disadvantages:**

- Requires database-specific coding to be done in the middle tier.
- The middle ware layer added may result in additional latency, but is typically overcome by using better middle ware services.

### 8.1.4 Type 4 driver – Database-Protocol driver/Thin Driver (Pure Java driver):

- The JDBC type 4 driver, also known as the Direct to Database **Pure Java Driver**, is a database driver implementation that converts JDBC calls directly into a vendor-specific database protocol.
- Written completely in Java, type 4 drivers are thus platform independent. They install inside the Java Virtual Machine of the client. This provides better performance than the type 1 and type 2 drivers as it does not have the overhead of conversion of calls into ODBC or database API calls. Unlike the type 3 drivers, it does not need associated software to work.
- As the database protocol is vendor specific, the JDBC client requires separate drivers, usually vendor supplied, to connect to different types of databases.

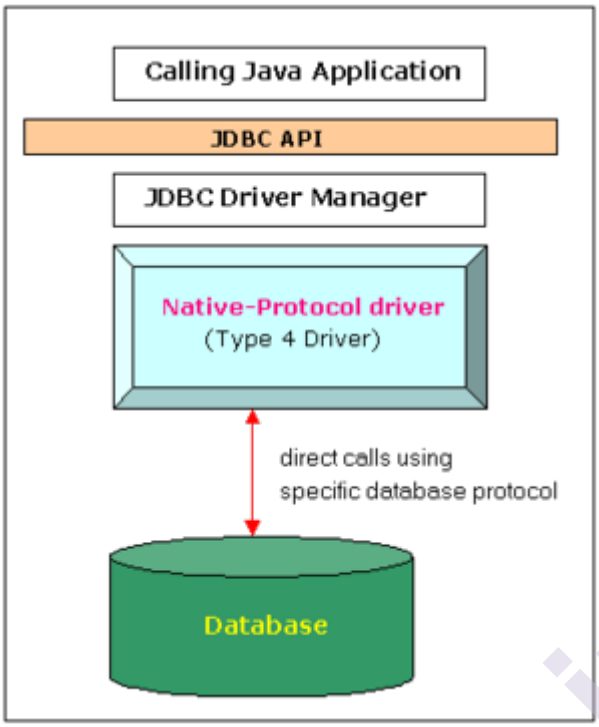



Figure 8.1.4 shows architecture of Type 4 Driver (Pure Java Driver)

**Advantages:**

- Completely implemented in Java to achieve platform independence.
- These drivers don't translate the requests into an intermediary format (such as ODBC).
- The client application connects directly to the database server. No translation or middleware layers are used, improving performance.
- The JVM can manage all aspects of the application-to-database connection; this can facilitate debugging.

**Disadvantages:**

- Drivers are database specific, as different database vendors use widely different (and usually proprietary) network protocols.

### 8.1.5 Type 5: highly-functional drivers with superior performance:

- Type 5 JDBC drivers (such as DataDirect JDBC drivers) offer advanced functionality and superior performance over other driver types.
- Any JDBC drivers that are fundamentally based on the Type 4 architecture yet are designed to address all or most of these limitations represent a drastic departure from the norm. In fact, such drivers could be classified as an

entirely new type. Call them what you will, but for the purposes of this discussion, they are "Type 5."

- Not all developers truly understand the role JDBC middleware plays in application-to-data operations, beyond simply enabling connectivity. In fact, with the increased abstraction of object modeling and higher-level applications, many a developer views JDBC drivers as vital but "dumb" pipes rather than critical cogs that not only drive the success of an application stack but also enhance it.
- For this reason, some in the developer community may take a "so-what?" attitude toward the notion of a JDBC Type 5 driver. They may even reject the whole notion of a data connectivity driver as a component where innovations such as application failover for high availability ought to take place. Such innovations, they could reason, are more appropriately handled at the higher application level. This reasoning, however, is very debatable.

➢ **The Limits of previous driver (Type 4):**

Among developers who are knowledgeable about the behind-the-scenes workings of middleware data connectivity using JDBC drivers, the limitations of a Type 4 driver are generally undisputable. These include:

- The need to write and maintain code specific to each supported data source. Even with modern framework-based object-relational mapping (ORM) models, JDBC Type 4 drivers typically require the use of proprietary code to support variant database features such as BLOBs and CLOBs, high availability, and XA.
- Poor and/or inconsistent driver response times or data throughput performance when the driver is deployed in certain runtime environments (such as different JVMs) or with ORMs and application servers.
- The inability to tune or optimize critical applications, or the ability to do so only with considerable application downtime.
- Poor virtual machine (VM) consolidation ratios due to inefficient runtime CPU usage and high memory footprints.
- Deployment headaches such as having to deploy multiple JARs to support different JVMs or database versions, or DLLs to support certain driver or database functionality.

These limitations point to the following key trends and advances in the modern Java environment as the sources of today's Type 4 driver challenges:

- Application support of multiple databases.
- Increased server virtualization and data center consolidation.
- Rapid adoption of ORM models such as Hibernate or app servers such as Jboss that sit on top of JDBC drivers and permit no access to JDBC code.

➢ **The Advantages of Type 5:**

So, how would a Type 5 driver address all these limitations? Apart from the superior client-side, single-tier, 100% Java architecture of Type 4 drivers, what other characteristics would a Type 5 driver have?

- **Unrestricted performance** — The driver should be capable of delivering maximized and consistent data throughput regardless of the runtime environment or data-access model.
- **Codeless enhancement** — The driver should offer the ability to add, configure, or tune features and functionality for any application without requiring any changes to application code, regardless of environment or data-access model. While this is, of course, pursuant to unrestricted performance, it is also important for ensuring that new database or driver functionality is available across all supported JVMs or hardware and can be accessed despite the use of ORM or app server models that prevent access to the JDBC code, which is required to enable such features and functionality. Comprehensive driver-connection options are a way in which this could be accomplished.
- **Resource efficiency** — The use of application runtime CPU and memory should be minimized, and should be tunable in the driver to fit specific runtime environment parameters or limits. The consumption of such resources by middleware data-access operations is often overlooked until it adversely impacts server consolidation goals in virtualization initiatives.
- **All-in-one deployment** — A JDBC Type 5 driver should require a single JAR file, regardless of Java environment or application requirements. It should require no client libraries or external DLLs, regardless of the deployment environment or features used by the application — including bulk data loading, security, high availability, and XA features.
- **Streamlined standard** — A JDBC Type 5 driver ought to require no proprietary extensions to the JDBC specification for any supported data source. This would address the requirement, typical of most Type 4 drivers, for proprietary code to support features such as BLOBs and CLOBs, high availability, and XA.

- While a formal committee approval of a new JDBC standard would be preferable in the long run, the current limitations of Type 4 drivers frankly have become too glaring and counterproductive to wait for that drawn-out process. Organizations that rely on modern data-driven Java applications need to be able to implement JDBC Type 5 drivers now.
- Suppose you could run eight VMs on a server instead of four by merely tweaking some configuration settings in a JDBC driver. This is an entirely plausible scenario where a Type 5 driver can enhance the overall usefulness of the application stack and overall IT environment much more easily than by any other available solution. Think of the application stack as a lever and the enhancements as a fulcrum. The further back on the lever the fulcrum is placed, the greater the leverage. This analogy applies to JDBC drivers at the data-connectivity level — provided they are built to deliver the enhancements discussed here, very simple and nonintrusive changes can have dramatic impact on how the entire application stack performs.

## 10.2 MANAGING JDBC CONNECTION

Java Database Connectivity (JDBC) connections allow the Gateway to query external databases and then use the query results during policy consumption.

- Java Database Connectivity with 5 Steps:
  - Register the Driver class
  - Create connection
  - Create statement
  - Execute queries
  - Close connection

**1) Register the driver class:**

The forName() method of Class class is used to register the driver class. This method is used to dynamically load the driver class.

Syntax of forName() method

public static void forName(String className)throws ClassNotFoundException

Example to register the OracleDriver class

Here, Java program is loading oracle driver to esteblish database connection.

Class.forName("oracle.jdbc.driver.OracleDriver");

**2) Create the connection object:**

The getConnection() method of DriverManager class is used to establish connection with the database.

Syntax of getConnection() method

1) public static Connection getConnection(String url)throws SQLException
2) public static Connection getConnection(String url,String name,String password)

throws SQLException

Example to establish connection with the Oracle database

```
Connection con=DriverManager.getConnection(
"jdbc:oracle:thin:@localhost:1234:vs","system","password");
```

**3) Create the Statement object:**

The createStatement() method of Connection interface is used to create statement. The object of statement is responsible to execute queries with the database.

Syntax of createStatement() method

```
public Statement createStatement()throws SQLException
```

Example to create the statement object

```
Statement stmt=con.createStatement();
```

**4) Execute the query:**

The executeQuery() method of Statement interface is used to execute queries to the database. This method returns the object of ResultSet that can be used to get all the records of a table.

Syntax of executeQuery() method

```
public ResultSet executeQuery(String sql)throws SQLException
```

Example to execute query

```
ResultSet rs=stmt.executeQuery("select * from student");
while(rs.next()){
System.out.println(rs.getInt(1)+" "+rs.getString(2));
}
```

**5) Close the connection object:**

By closing connection object statement and ResultSet will be closed automatically. The close() method of Connection interface is used to close the connection.

Syntax of close() method

```
public void close()throws SQLException
```

Example to close connection

```
con.close();
```

## 10.3 CONFIGURING DATA SOURCE TO OBTAIN JDBC CONNECTION

To set up JDBC connectivity, you configure connection pools, Data Source objects (always recommended, but optional in some cases), and MultiPools (optional) by defining attributes in the Administration Console or, for dynamic connection pools, in application code or at the command line.

- **There are three types of transaction scenarios:**
  - Local transactions—non-distributed transactions
  - Distributed transactions using an XA Driver—distributed transactions with multiple participants that use two-phase commit
  - Distributed transactions using a non-XA Driver—transactions with a single resource manager and single database instance that emulate two-phase commit
- You configure Data Source objects (DataSources and TxDataSources), connection pools, and MultiPools according to the way transactions are handled in your system. The following table summarizes how to configure these objects for use in the three transaction scenarios:

**Table Summary of JDBC Configuration Guidelines**

| Description/Object | Local Transactions | Distributed Transactions<br>XA Driver | Distributed Transactions<br>Non-XA Driver |
|---|---|---|---|
| **JDBC driver** | ▪ WebLogic jDriver for Oracle and Microsoft SQL Server.<br>▪ Compliant third-party drivers. | ▪ WebLogic jDriver for Oracle/XA.<br>▪ Compliant third-party drivers. | ▪ WebLogic jDriver for Oracle and Microsoft SQL Server<br>▪ Compliant third-party drivers. |
| **Data Source** | Data Source object recommended. (If there is no Data | Requires Tx Data Source. | Requires Tx Data Source.<br>Select Emulate Two-Phase Commit for non-XA Driver |

| | | | |
|---|---|---|---|
| | Source, use the JDBC API.) | | (set enable two-phase commit=true) if more than one resource is involved. See Configuring Non-XA JDBC Drivers for Distributed Transactions. |
| **Connection Pool** | Requires Data Source object when configuring in the Administration Console. | Requires Tx Data Source. | Requires Tx Data Source. |
| **MultiPool** | Connection Pool and Data Source required. | Not supported in distributed transactions. | Not supported in distributed transactions. |

### 8.3.1 Following steps are representing Java Database Connectivity with MySQL:

- To connect Java application with the MySQL database, we need to follow 5 following steps.
- In this example we are using MySql as the database. So we need to know following informations for the mysql database:
- Driver class: The driver class for the mysql database is com.mysql.jdbc.Driver.
- Connection URL: The connection URL for the mysql database is jdbc:mysql://localhost:3306/idol where jdbc is the API, mysql is the database, localhost is the server name on which mysql is running, we may also use IP address, 3306 is the port number and idol is the database name. We may use any database, in such case, we need to replace the idol with our database name.
- ✓ Username: The default username for the mysql database is root.

- ✓ Password: It is the password given by the user at the time of installing the mysql database. In this example, we are going to use root as the password.
- ➢ Let's first create a table in the mysql database, but before creating table, we need to create database first.

  create database idol;

  use idol;

  create table student(id int(10),name varchar(40),age int(3));

**Code:**

```
import java.sql.*;
class MysqlCon{
public static void main(String args[]){
try{
Class.forName("com.mysql.jdbc.Driver");
Connection con=DriverManager.getConnection(
"jdbc:mysql://localhost:3306/ idol ","root","root");
//here sonoo is database name, root is username and password
Statement stmt=con.createStatement();
ResultSet rs=stmt.executeQuery("select * from student ");
while(rs.next())
System.out.println(rs.getInt(1)+"  "+rs.getString(2)+"  "+rs.getString(3));
con.close();
}catch(Exception e){ System.out.println(e);}
}
}
```

- ➢ To connect java application with the mysql database, mysqlconnector.jar file is required to be loaded.
  - download the jar file mysql-connector.jar
- ➢ Two ways to load the jar file:
  - Paste the mysqlconnector.jar file in jre/lib/ext folder
- ➢ Set classpath

  1) Paste the mysqlconnector.jar file in JRE/lib/ext folder:

     Download the mysqlconnector.jar file. Go to jre/lib/ext folder and paste the jar file here.

  2) Set classpath:

- There are two ways to set the classpath:
- temporary
- permanent

- How to set the temporary classpath
- open command prompt and write:
  C:>set classpath=c:\folder\mysql-connector-java-5.0.8-bin.jar;.;

- How to set the permanent classpath
- Go to environment variable then click on new tab. In variable name write classpath and in variable value paste the path to the mysqlconnector.jar file by appending mysqlconnector.jar;.; as C:\folder\mysql-connector-java-5.0.8-bin.jar;.;

## 10.4 DATA ACCESS OPERATIONS WITH JDBC TEMPLATE AND SPRING

- To understand the concepts related to Spring JDBC framework with JdbcTemplate class, let us write a simple example, which will implement all the CRUD operations on the following Student table.

```
CREATE TABLE Student(
ID   INT NOT NULL AUTO_INCREMENT,
NAME VARCHAR(20) NOT NULL,
AGE  INT NOT NULL,
PRIMARY KEY (ID) );
```

Before proceeding, let us have a working Eclipse IDE in place and take the following steps to create a Spring application –

Steps with Description:

1) Create a project with a name SpringExample and create a package com.tutorialspoint under the src folder in the created project.

2) Add required Spring libraries using Add External JARs option as explained in the Spring Hello World Example chapter.

3) Add Spring JDBC specific latest libraries mysql-connector-java.jar, org.springframework.jdbc.jar and org.springframework.transaction.jar in the project. You can download required libraries if you do not have them already.

4) Create DAO interface StudentDAO and list down all the required methods. Though it is not required and you can directly write StudentJDBCTemplate class, but as a good practice, let's do it.

5) Create other required Java classes Student, StudentMapper, StudentJDBCTemplate and MainApp under the com.tutorialspoint package.

6) Make sure you already created Student table in TEST database. Also make sure your MySQL server is working fine and you have read/write access on the database using the give username and password.

7) Create Beans configuration file Beans.xml under the src folder.

8) The final step is to create the content of all the Java files and Bean Configuration file and run the application as explained below.

Following is the content of the Data Access Object interface file StudentDAO.java

```
package com.tutorialspoint;
import java.util.List;
import javax.sql.DataSource;
public interface StudentDAO {
  /**
    * This is the method to be used to initialize
    * database resources ie. Connection.
  */
  public void setDataSource(DataSource ds);
  /**
    * This is the method to be used to create
    * a record in the Student table.
  */
  public void create(String name, Integer age);
  /**
    * This is the method to be used to list down
    * a record from the Student table corresponding
    * to a passed student id.
  */
  public Student getStudent(Integer id);
  /**
    * This is the method to be used to list down
```

```
    * all the records from the Student table.
  */
  public List<Student> listStudents();
    /**
    * This is the method to be used to delete
    * a record from the Student table corresponding
    * to a passed student id.
  */
  public void delete(Integer id);
    /**
    * This is the method to be used to update
    * a record into the Student table.
  */
  public void update(Integer id, Integer age);
}
```

Following is the content of the Student.java file

```
package com.tutorialspoint;
public class Student {
  private Integer age;
  private String name;
  private Integer id;
  public void setAge(Integer age) {
    this.age = age;
  }
  public Integer getAge() {
    return age;
  }
  public void setName(String name) {
    this.name = name;
  }
  public String getName() {
    return name;
  }
  public void setId(Integer id) {
    this.id = id;
  }
  public Integer getId() {
```

```
    return id;
  }
}
```

Following is the content of the StudentMapper.java file

```
package com.tutorialspoint;
import java.sql.ResultSet;
import java.sql.SQLException;
import org.springframework.jdbc.core.RowMapper;
public class StudentMapper implements RowMapper<Student> {
public Student mapRow(ResultSet rs, int rowNum) throws SQLException {
    Student student = new Student();
    student.setId(rs.getInt("id"));
    student.setName(rs.getString("name"));
    student.setAge(rs.getInt("age"));
    return student;
  }
}
```

Following is the implementation class file StudentJDBCTemplate.java for the defined StufentDAO

```
interface StudentDAO.
Package com.tutorialspoint;
import java.util.List;
import javax.sql.DataSource;
import org.springframework.jdbc.core.JdbcTemplate;
public class StudentJDBCTemplate implements StudentDAO {
private DataSource dataSource;
  private JdbcTemplate jdbcTemplateObject;
  public void setDataSource(DataSource dataSource) {
    this.dataSource = dataSource;
    this.jdbcTemplateObject = new JdbcTemplate(dataSource);
  }
  public void create(String name, Integer age) {
    String SQL = "insert into Student (name, age) values (?, ?)";
    jdbcTemplateObject.update( SQL, name, age);
    System.out.println("Created Record Name = " + name + " Age = " + age);
    return;
  }
```

```
  public Student getStudent(Integer id) {
    String SQL = "select * from Student where id = ?";
    Student student = jdbcTemplateObject.queryForObject(SQL,
      new Object[]{id}, new StudentMapper());
      return student;
  }
  public List<Student> listStudents() {
    String SQL = "select * from Student";
    List <Student> students = jdbcTemplateObject.query(SQL, new
StudentMapper());
    return students;
  }
  public void delete(Integer id) {
    String SQL = "delete from Student where id = ?";
    jdbcTemplateObject.update(SQL, id);
    System.out.println("Deleted Record with ID = " + id );
    return;
  }
  public void update(Integer id, Integer age){
    String SQL = "update Student set age = ? where id = ?";
    jdbcTemplateObject.update(SQL, age, id);
    System.out.println("Updated Record with ID = " + id );
    return;
  }
}
```

Following is the content of the MainApp.java file

```
package com.tutorialspoint;
import java.util.List;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.tutorialspoint.StudentJDBCTemplate;
public class MainApp {
public static void main(String[] args) {
    ApplicationContext context = new
ClassPathXmlApplicationContext("Beans.xml");
    StudentJDBCTemplate studentJDBCTemplate =
(StudentJDBCTemplate)context.getBean("studentJDBCTemplate");
```

```
    System.out.println("------Records Creation--------" );
    studentJDBCTemplate.create("Virat", 33);
    studentJDBCTemplate.create("Rohit", 34);
    studentJDBCTemplate.create("Hardik", 29);
    System.out.println("------Listing Multiple Records--------" );
    List<Student> students = studentJDBCTemplate.listStudents();
    for (Student record : students) {
      System.out.print("ID : " + record.getId() );
      System.out.print(", Name : " + record.getName() );
      System.out.println(", Age : " + record.getAge());
    }
    System.out.println("----Updating Record with ID = 2 -----" );
    studentJDBCTemplate.update(2, 20);
    System.out.println("----Listing Record with ID = 2 -----" );
    Student student = studentJDBCTemplate.getStudent(2);
    System.out.print("ID : " + student.getId() );
    System.out.print(", Name : " + student.getName() );
    System.out.println(", Age : " + student.getAge());
  }
}
```

Following is the configuration file Beans.xml

```
<?xml version = "1.0" encoding = "UTF-8"?>
<beans xmlns = "http://www.springframework.org/schema/beans"
  xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation = "http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans-3.0.xsd ">
  <!–Initialization for data source →
  <bean id="dataSource"
    class = "org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name = "driverClassName" value = "com.mysql.jdbc.Driver"/>
    <property name = "url" value = "jdbc:mysql://localhost:3306/TEST"/>
    <property name = "username" value = "root"/>
    <property name = "password" value = "password"/>
  </bean>
  <!–Definition for studentJDBCTemplate bean →
  <bean id = "studentJDBCTemplate"
    class = "com.tutorialspoint.StudentJDBCTemplate">
```

```
    <property name = "dataSource" ref = "dataSource" />
  </bean>
</beans>
```

Once you are done creating the source and bean configuration files, let us run the application. If everything is fine with your application, it will print the following message –

```
------Records Creation--------
Created Record Name = Virat Age = 33
Created Record Name = Rohit Age = 34
Created Record Name = Hardik Age = 29
------Listing Multiple Records--------
ID : 1, Name : Virat, Age : 33
ID : 2, Name : Rohit, Age : 34
ID : 3, Name : Hardik, Age : 29
----Updating Record with ID = 2 -----
Updated Record with ID = 1
----Listing Record with ID = 1 -----
ID : 1, Name : Virat, Age : 34
```

You can try and delete the operation yourself, which we have not used in the example, but now you have one working application based on Spring JDBC framework, which you can extend to add sophisticated functionality based on your project requirements. There are other approaches to access the database where you will use NamedParameterJdbcTemplate and SimpleJdbcTemplate classes, so if you are interested in learning these classes then kindly check the reference manual for Spring Framework.



## 8.5 RDBMS OPERATION CLASSES

- RDBMS stands for relational database management system.
- JdbcTemplate is ideal for simple queries and updates, and when you need to build SQL strings dynamically, but sometimes you might want a higher level of abstraction, and a more object-oriented approach to database access. This is provided by the org.springframework.jdbc.object package. It contains the SqlQuery, SqlMappingQuery, SqlUpdate, and StoredProcedure classes that are intended to be the central classes used by most Spring JDBC applications. These classes are used together with a DataSource and the SqlParameter class. Each of the RDBMS Operation classes is based on the RDBMSOperation class and they all use a JdbcTemplate internally for database access. As a user of these classes you will have to provide either an

existing JdbcTemplate or you can provide a DataSource and the framework code will create a JdbcTemplate when it needs one.

- Spring's RDBMS Operation classes are parameterized operations that are threadsafe once they are prepared and compiled. You can safely create a single instance for each operation that you define. The preparation consists of providing a datasource and defining all the parameters that are needed for the operation. We just mentioned that they are threadsafe once they are compiled. This means that we have to be a little bit careful when we create these operations. The recommended method is to define the parameters and compile them in the constructor. That way there will not be any risk for thread conflicts.

## 8.6 MODELLING JDBC OPERATIONS AS JAVA OBJECT

The org.springframework.jdbc.object package contains classes that allow one to access the database in a more object-oriented manner. By way of an example, one can execute queries and get the results back as a list containing business objects with the relational column data mapped to the properties of the business object. One can also execute stored procedures and run update, delete and insert statements.

### 8.6.1 SqlQuery:

- SqlQuery is a reusable, threadsafe class that encapsulates an SQL query. Subclasses must implement the newRowMapper(..) method to provide a RowMapper instance that can create one object per row obtained from iterating over the ResultSet that is created during the execution of the query. The SqlQuery class is rarely used directly since the MappingSqlQuery subclass provides a much more convenient implementation for mapping rows to Java classes. Other implementations that extend SqlQuery are MappingSqlQueryWithParameters and UpdatableSqlQuery.

### 8.6.2 MappingSqlQuery:

- MappingSqlQuery is a reusable query in which concrete subclasses must implement the abstract mapRow(..) method to convert each row of the supplied ResultSet into an object of the type specified. Below is a brief example of a custom query that maps the data from the t_actor relation to an instance of the Actor class.

```
Public class ActorMappingQuery extends MappingSqlQuery<Actor> {
  public ActorMappingQuery(DataSource ds) {
  super(ds, "select id, first_name, last_name from t_actor where id = ?");
```

```
    super.declareParameter(new SqlParameter("id", Types.INTEGER));
    compile();
    }
    @Override
    protected Actor mapRow(ResultSet rs, int rowNumber) throws
SQLException {
        Actor actor = new Actor();
        actor.setId(rs.getLong("id"));
        actor.setFirstName(rs.getString("first_name"));
        actor.setLastName(rs.getString("last_name"));
        return actor;
    }
}
```

The class extends MappingSqlQuery parameterized with the Actor type. We provide a constructor for this customer query that takes the DataSource as the only parameter. In this constructor we call the constructor on the superclass with the DataSource and the SQL that should be executed to retrieve the rows for this query. This SQL will be used to create a PreparedStatement so it may contain place holders for any parameters to be passed in during execution. Each parameter must be declared using the declareParameter method passing in an SqlParameter. The SqlParameter takes a name and the JDBC type as defined in java.sql.Types. After all parameters have been defined we call the compile() method so the statement can be prepared and later be executed. This class is thread safe once it has been compiled, so as long as these classes are created when the DAO is initialized they can be kept as instance variable and be reused.

```
Private ActorMappingQuery actorMappingQuery;

@Autowired
public void setDataSource(DataSource dataSource) {
    this.actorMappingQuery = new ActorMappingQuery(dataSource);
}
public Customer getCustomer(Long id) {
return actorMappingQuery.findObject(id);
}
```

The method in this example retrieves the customer with the id that is passed in as the only parameter. Since we only want one object returned we simply call the convenience method findObject with the id as parameter. If we instead had a query the returned a list of objects and took additional

parameters then we would use one of the execute methods that takes an array of parameter values passed in as varargs.

```
Public List<Actor> searchForActors(int age, String namePattern) {
   List<Actor> actors = actorSearchMappingQuery.execute(age,
namePattern);
   return actors;
}
```

### 8.6.3 SqlUpdate:

- The SqlUpdate class encapsulates an SQL update. Like a query, an update object is reusable, and like all RdbmsOperation classes, an update can have parameters and is defined in SQL. This class provides a number of update(..) methods analogous to the execute(..) methods of query objects. This class is concrete. Although it can be subclassed (for example to add a custom update method – like in this example where we call it execute) it can easily be parameterized by setting SQL and declaring parameters.

```
Import java.sql.Types;
import javax.sql.DataSource;
import org.springframework.jdbc.core.SqlParameter;
import org.springframework.jdbc.object.SqlUpdate;
public class UpdateCreditRating extends SqlUpdate {
public UpdateCreditRating(DataSource ds) {
      setDataSource(ds);
      setSql("update customer set credit_rating = ? where id = ?");
      declareParameter(new SqlParameter("creditRating",
Types.NUMERIC));
      declareParameter(new SqlParameter("id", Types.NUMERIC));
      compile();
   }
   /**
    * @param id for the Customer to be updated
    * @param rating the new value for credit rating
    * @return number of rows updated
    */
   public int execute(int id, int rating) {
   return update(rating, id);
   }
}
```

### 8.6.4 StoredProcedure:

- The StoredProcedure class is a superclass for object abstractions of RDBMS stored procedures. This class is abstract, and its various execute(..) methods have protected access, preventing use other than through a subclass that offers tighter typing.
- The inherited sql property will be the name of the stored procedure in the RDBMS.
- To define a parameter to be used for the StoredProcedure class, you use an SqlParameter or one of its subclasses. You must specify the parameter name and SQL type in the constructor. The SQL type is specified using the java.sql.Types constants. We have already seen declarations like:

  new SqlParameter("in_id", Types.NUMERIC),

  new SqlOutParameter("out_first_name", Types.VARCHAR),
- The first line with the SqlParameter declares an in parameter. In parameters can be used for both stored procedure calls and for queries using the SqlQuery and its subclasses covered in the following section.
- The second line with the SqlOutParameter declares an out parameter to be used in the stored procedure call. There is also an SqlInOutParameter for inout parameters, parameters that provide an in value to the procedure and that also return a value
- In addition to the name and the SQL type you can specify additional options. For in parameters you can specify a scale for numeric data or a type name for custom database types. For out parameters you can provide a RowMapper to handle mapping of rows returned from a REF cursor. Another option is to specify an SqlReturnType that provides and opportunity to define customized handling of the return values.
- Here is an example of a simple DAO that uses a StoredProcedure to call a function, sysdate(), that comes with any Oracle database. To use the stored procedure functionality you have to create a class that extends StoredProcedure. In this example the StoredProcedure class is an inner class, but if you need to reuse the StoredProcedure you would declare it as a top-level class. There are no input parameters in this example, but there is an output parameter that is declared as a date type using the class SqlOutParameter. The execute() method executes the procedure and extracts the returned date from the results Map. The results Map has an entry for each declared output parameter, in this case only one, using the parameter name as the key.

```
Import java.sql.Types;
import java.util.Date;
import java.util.HashMap;
import java.util.Map;
import javax.sql.DataSource;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.SqlOutParameter;
import org.springframework.jdbc.object.StoredProcedure;
public class StoredProcedureDao {
   private GetSysdateProcedure getSysdate;
   @Autowired
   public void init(DataSource dataSource) {
      this.getSysdate = new GetSysdateProcedure(dataSource);
   }
    public Date getSysdate() {
     return getSysdate.execute();
   }
   private class GetSysdateProcedure extends StoredProcedure {
 private static final String SQL = “sysdate”;
      public GetSysdateProcedure(DataSource dataSource) {
         setDataSource(dataSource);
         setFunction(true);
         setSql(SQL);
         declareParameter(new SqlOutParameter(“date”, Types.DATE));
         compile();
      }
      public Date execute() {
         // the ‘sysdate’ sproc has no input parameters, so an empty Map is
supplied…
         Map<String, Object> results = execute(new HashMap<String,
Object>());
         Date sysdate = (Date) results.get(“date”);
         return sysdate;
      }
   }
```

```
}
```

Below is an example of a StoredProcedure that has two output parameters (in this case Oracle REF cursors).

```
Import oracle.jdbc.OracleTypes;
import org.springframework.jdbc.core.SqlOutParameter;
import org.springframework.jdbc.object.StoredProcedure;
import javax.sql.DataSource;
import java.util.HashMap;
import java.util.Map;
public class TitlesAndGenresStoredProcedure extends StoredProcedure {
private static final String SPROC_NAME = "AllTitlesAndGenres";
public TitlesAndGenresStoredProcedure(DataSource dataSource) {
      super(dataSource, SPROC_NAME);
      declareParameter(new SqlOutParameter("titles",
OracleTypes.CURSOR, new TitleMapper()));
      declareParameter(new SqlOutParameter("genres",
OracleTypes.CURSOR, new GenreMapper()));
      compile();
   }
   public Map<String, Object> execute() {
      // again, this sproc has no input parameters, so an empty Map is
supplied
      return super.execute(new HashMap<String, Object>());
   }
}
```

- Notice how the overloaded variants of the declareParameter(..) method that have been used in the TitlesAndGenresStoredProcedure constructor are passed RowMapper implementation instances; this is a very convenient and powerful way to reuse existing functionality. (The code for the two RowMapper implementations is provided below in the interest of completeness.)
- First the TitleMapper class, which simply maps a ResultSet to a Title domain object for each row in the supplied ResultSet.

```
Import org.springframework.jdbc.core.RowMapper;
import java.sql.ResultSet;
```

```
import java.sql.SQLException;
import com.foo.domain.Title;
public final class TitleMapper implements RowMapper<Title> {
public Title mapRow(ResultSet rs, int rowNum) throws SQLException {
      Title title = new Title();
      title.setId(rs.getLong("id"));
      title.setName(rs.getString("name"));
      return title;
   }
}
```

Second, the GenreMapper class, which again simply maps a ResultSet to a Genre domain object for each row in the supplied ResultSet.

```
Import org.springframework.jdbc.core.RowMapper;
import java.sql.ResultSet;
import java.sql.SQLException;
import com.foo.domain.Genre;
public final class GenreMapper implements RowMapper<Genre> {
public Genre mapRow(ResultSet rs, int rowNum) throws SQLException {
return new Genre(rs.getString("name"));
   }
}
```

- If you need to pass parameters to a stored procedure (that is the stored procedure has been declared as having one or more input parameters in its definition in the RDBMS), you should code a strongly typed execute(..) method which would delegate to the superclass' (untyped) execute(Map parameters) (which has protected access); for example:

```
import oracle.jdbc.OracleTypes;
import org.springframework.jdbc.core.SqlOutParameter;
import org.springframework.jdbc.core.SqlParameter;
import org.springframework.jdbc.object.StoredProcedure;
import javax.sql.DataSource;
import java.sql.Types;
import java.util.Date;
import java.util.HashMap;
import java.util.Map;
public class TitlesAfterDateStoredProcedure extends StoredProcedure {
private static final String SPROC_NAME = "TitlesAfterDate";
private static final String CUTOFF_DATE_PARAM = "cutoffDate";
```

```
public TitlesAfterDateStoredProcedure(DataSource dataSource) {
    super(dataSource, SPROC_NAME);
    declareParameter(new SqlParameter(CUTOFF_DATE_PARAM,
Types.DATE);
    declareParameter(new SqlOutParameter("titles",
OracleTypes.CURSOR, new TitleMapper()));
    compile();
  }
  public Map<String, Object> execute(Date cutoffDate) {
    Map<String, Object> inputs = new HashMap<String, Object>();
    inputs.put(CUTOFF_DATE_PARAM, cutoffDate);
    return super.execute(inputs);
  } }
```

## 8.5 Conclusion

This module explored the role of JDBC in providing solution developers with enterprise-wide database connectivity capabilities and the JDBC architecture. You studied the relationship between JDBC and Java. You learned about various database systems and how JDBC was designed to work with relational DBMSs and the relational command language, SQL. You learned about two and n-tier application models. Finally, you learned about how JBDC was designed to leverage the power of Java.

Specifically, you learned to:

1. Distinguish the role and place of JDBC among the Java technologies
2. Differentiate between DBMS types
3. Explain the relational model
4. Describe design considerations for JDBC and ODBC in a solution
5. Explain what SQL is and its role in database processing
6. Explain JDBC as it functions in two and n-tier system designs
7. Describe the capabilities of Java and a DBMS used with JDBC

## 8.6 List Of References

https://www.javatpoint.com/steps-to-connect-to-the-database-in-java

https://en.wikipedia.org/wiki/Java_Database_Connectivity

# 9

# JDBC ARCHITECTURE

**Unit Structure**



## 9.0 Objectives

The Java Database Connectivity is a standard Java API specifying interfaces for connectivity between the Java applications and a wide range of databases. The JDBC contains methods to connect to a database server, send queries to create a table, update, delete, insert records in a database, retrieve and process the results obtained from a database. Nowadays, there are a lot of frameworks that are built for easier work with databases. But they contain the JDBC under the hood.

## 9.1 Introduction

JDBC or Java Database Connectivity is a specification from Sun microsystems that provides a standard abstraction(that is API or Protocol) for java applications to communicate with various databases. It provides the language with java database connectivity standard. It is used to write programs required to access databases. JDBC along with the database driver is capable of accessing databases and spreadsheets. The enterprise data stored in a relational database(RDB) can be accessed with the help of JDBC APIs.

## 9.2 Jdbc Architecture

Java Database Connectivity supports two-tier and three-tier architectures to access a database. They are also called as processing models for database access. Let’s look closer at them.

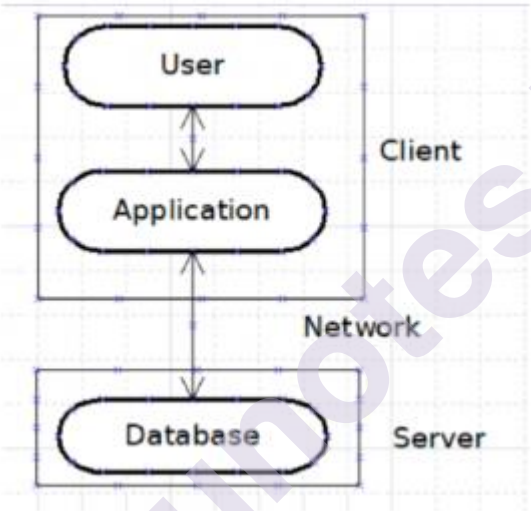
It contains two types of architecture.

a) Two-Tier Architecture

b) Three-Tier Architecture

### 9.2.1 Two-tier Architecture:

- In this kind of architecture, a java application is directly communicating with a database. It requires one of the specific Java Database Connectivity drivers to connect to a specific database. All queries and requests are sending by the user to the database and results are receiving back by the user. The database can be running on the same machine. Also, it can be on a remote machine and to be connected via a network. This approach is called a client-server architecture.





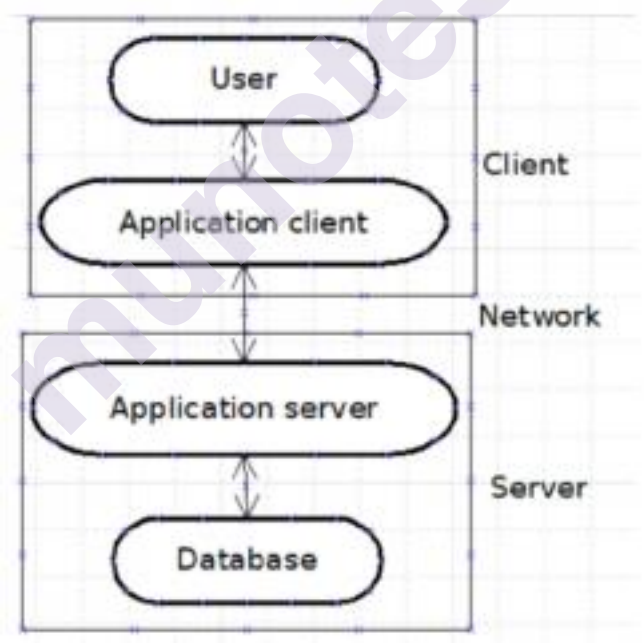

⇨ Diagram represent two-tier architecture of JDBC.

- In the two-tier model, a Java applet or application talks directly to the data source. This requires a JDBC driver that can communicate with the particular data source being accessed. A user's commands are delivered to the database or other data source, and the results of those statements are sent back to the user. The data source may be located on another machine to which the user is connected via a network. This is referred to as a client/server configuration, with the user's machine as the client, and the machine housing the data source as the server. The network can be an intranet, which, for example, connects employees within a corporation, or it can be the Internet.
- In the three-tier model, commands are sent to a "middle tier" of services, which then sends the commands to the data source. The data source processes the commands and sends the results back to the middle tier, which then sends

them to the user. MIS directors find the three-tier model very attractive because the middle tier makes it possible to maintain control over access and the kinds of updates that can be made to corporate data. Another advantage is that it simplifies the deployment of applications. Finally, in many cases, the three-tier architecture can provide performance advantages.

### 9.2.2 Three-tier Architecture:

- In the three-tier model, there is no direct communication. First of all, all requests and queries are sent to the middle tier. A middle tier can be a browser with a web page or desktop application, that sends a request to the java application. After that request is sent to the database. The database processes the request and sends the result back to the middle tier. And the middle tier then communicates with the user. This model has better performance and simplifies application deployment.
- All of these different executables are able to access a database with the use of a JDBC driver as a middle tier: Java Desktop Applications, Java Applets, Java Servlets, Java Server Pages (JSPs), Enterprise JavaBeans (EJBs).

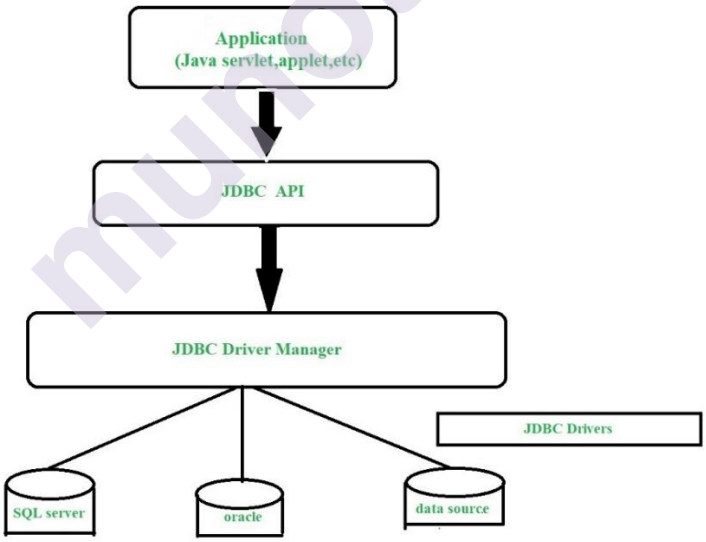



⇨ Diagram represent three-tier architecture of JDBC.

**For the two-tier model and the three-tier model, JDBC has two main layers:**

- JDBC API (an application-to-JDBC Manager connection), JDBC Driver API (this supports the JDBC Manager-to-Driver Connection).

- The JDBC driver manager and database-specific drivers provide the possibility to connect to heterogeneous databases. The function of the JDBC driver manager is to ensure that the correct driver is used to access each database. The driver manager is capable to support more than one driver. And the drivers can be concurrently connected to multiple different data sources.
- Until recently, the middle tier has often been written in languages such as C or C++, which offer fast performance. However, with the introduction of optimizing compilers that translate Java bytecode into efficient machine-specific code and technologies such as Enterprise JavaBeans™, the Java platform is fast becoming the standard platform for middle-tier development. This is a big plus, making it possible to take advantage of Java's robustness, multithreading, and security features.
- With enterprises increasingly using the Java programming language for writing server code, the JDBC API is being used more and more in the middle tier of a three-tier architecture. Some of the features that make JDBC a server technology are its support for connection pooling, distributed transactions, and disconnected rowsets. The JDBC API is also what allows access to a data source from a Java middle tier.

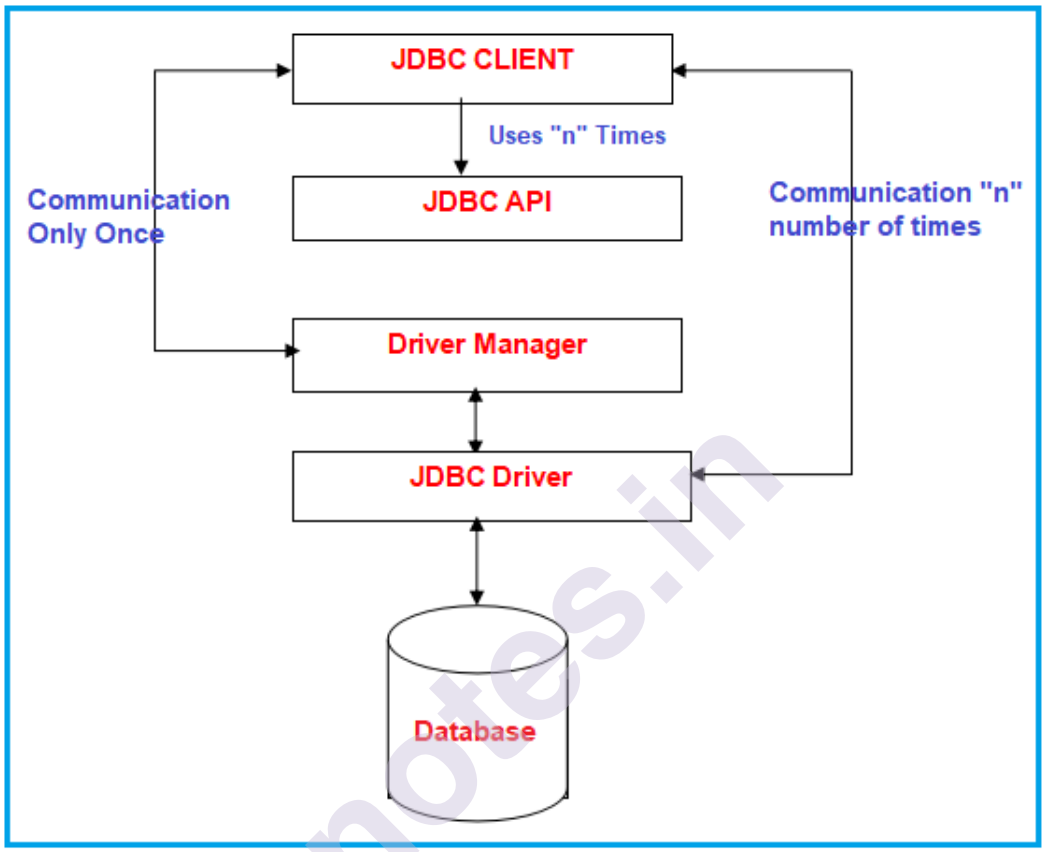



⇨ Diagram represent architecture of JDBC.

For example, your application can connect to the Oracle database and MySQL database at the same time. The connections will be created by the use of different drivers. You can read data from the Oracle database and insert it into MySQL and vise versa.

JDBC Architecture in Java:

In order to understand the JDBC Architecture, please have a look at the following image.






⇨ Diagram represent architecture of JDBC.

**Components of JDBC Driver:**

JDBC architecture is a client-server architecture. JDBC architecture has 5 elements.

1. **JDBC client**: Here JDBC Client is nothing but a java application which wants to communicate with the database.,
2. **JDBC API**: To write a java program which communicates with any database without changing the code Sun Microsystem has released JDBC API. By JDBC API implementation we can call JDBC Driver.
3. **JDBC Driver**: It is a java application that acts as an interface between java application and database. It understands the given java calls and converts into database calls and vice-versa.
4. **JDBC Driver Manager**: It acts as a data structure that collects a group of JDBC Drivers and allows the programmer to select the driver dynamically for the database connection.

5. **Database Server**: It is nothing but the Database server like Oracle, MySQL, SQL Server, etc. with which the JDBC client wants to communicate.

## 9.3 JDBC Program using DML Operation:

**DML**: DML stands Data Manipulation Language; its directly affected on data.

⇨ Data manipulation language (DML) provides statements like selecting, inserting, deleting and updating data in a database. Component of DML includes performing read-only queries of data. SQL is also known as DML or data manipulation language as it is used retrieve and manipulate data in a relational database.

⇨ executeupdate() is used to perform as Data manipulation language, it is used to modify stored data but not the schema or database objects. Manipulation takes place in form of persistent database objects, e.g., tables or stored procedures, via the SQL schema statements,rather than the data stored within them.

➢ **Insert:**

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;
/** Copyright (c), AnkitMittal JavaMadeSoEasy.com */
public class StatementInsertExample {
   public static void main(String... arg) {
        Connection con = null;
        Statement stmt = null;
        try {
            // registering Oracle driver class
            Class.forName("oracle.jdbc.driver.OracleDriver");

            // getting connection
            con = DriverManager.getConnection(
                    "jdbc:oracle:thin:@localhost:1521:orcl",
                    "vaibhav", "Oracle123");
            System.out.println("Connection established successfully!");
                stmt = con.createStatement();
            //execute insert query
```

```
            int numberOfRowsInserted=stmt.executeUpdate("INSERT into
EMPLOYEE(ID,NAME)" + "values (1, 'Nikita') ");
            System.out.println("numberOfRowsInserted=" +
numberOfRowsInserted);
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } catch (SQLException e) {
            e.printStackTrace();
        }
        finally{
            try {
                if(stmt!=null) stmt.close(); //close Statement
                if(con!=null) con.close(); // close connection
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }
}
 /*OUTPUT
 Connection established successfully!
numberOfRowsInserted=1
 */
```

Similarly, we can use remaining DML operations in JDBC program with java.

## 9.4 Conclusion

JDBC play role like a driver to move an object from one place to another by using java platform. This chapter explained the knowledge about JDBC architecture and some database programs.

## 9.5 List Of References

https://dotnettutorials.net/lesson/jdbc-architecture/

❖❖❖❖❖

Module 2

# 10

# SPRING

# GETTING STARTED WITH SPRING BOOT

**Unit Structure**



## 10.0 Objectives

The primary goals of Spring Boot are:

- To provide a radically faster and widely accessible 'getting started' experience for all Spring development.
- To be opinionated out of the box, but get out of the way quickly as requirements start to diverge from the defaults.

- To provide a range of non-functional features that are common to large classes of projects (e.g. embedded servers, security, metrics, health checks, externalized configuration).
- Spring Boot does not generate code and there is absolutely no requirement for XML configuration.

## 10.1 Introduction

- Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that you can "just run".
- We take an opinionated view of the Spring platform and third-party libraries so you can get started with minimum fuss. Most Spring Boot applications need minimal Spring configuration.
- Spring Boot is an open source, microservice-based Java web framework. The Spring Boot framework creates a fully production-ready environment that is completely configurable using its prebuilt code within its codebase. The microservice architecture provides developers with a fully enclosed application, including embedded application servers.

## 10.2 Spring Boot Overview

Spring Boot is just extension of the already existing and expansive Spring frameworks, but it has some specific features that make the application easier for working within the developer ecosystem.

That extension includes pre-configurable web starter kits that help facilitate the responsibilities of an application server that are required for other Spring projects.

### 10.2.1 Spring Boot Benefits

Spring Boot has a number of features that make it a great fit for quickly developing Java applications, including auto-configuration, health checks, and opinionated dependencies.

| **Benefits of Spring Boot** | |
|---|---|
| **Feature** | Benefit |
| **Standalone Application** | Can simply build the application jar and run the application with no need to customize the deployment. |
| **Embedded Servers** | Comes with prebuilt Tomcat, Jetty and Undertow application servers that do not require further installation to use. This also provides faster more efficient deployments resulting to shorting restart times. |

| | |
|---|---|
| **Auto-Configurable** | Spring and other 3rd party frameworks will be configured automatically. |
| **Production-Like Features** | Health checks, metrics and externalized configurations. |
| **Starter Dependencies** | This will provide opinionated dependencies designed to simplify the build configuration. This also provides complete build tool flexibility (Maven and Gradle). |

### 10.2.2 What's the Difference Between Spring Boot and Spring MVC?

- The major differences between Spring Boot and Spring MVC come down to differences between context and overall scale.
- Spring MVC is a specific Spring-based web framework in a traditional sense. That means it requires manual build configurations, specifying dependencies separately, and the use an application server.
- Spring Boot, on the other hand, is more like a module of Spring designed to package Spring applications or frameworks with automation and defaults.
- So in theory one could have a Spring MVC project packaged as a Spring Boot application. Both can be classified as Spring frameworks however the scale of Spring Boot encompasses many types of Spring frameworks. While, Spring MVC on the other hand specifies the design of the framework.

### 10.2.3 Spring Boot Staters

- Spring Boot starters were built to address exactly this problem. Starter POMs are a set of convenient dependency descriptors that you can include in your application. You get a one-stop-shop for all the Spring and related technology that you need, without having to hunt through sample code and copy-paste loads of dependency descriptors.

➢ **The Web Starter**

- First, let's look at developing the REST service; we can use libraries like Spring MVC, Tomcat and Jackson – a lot of dependencies for a single application.
- Spring Boot starters can help to reduce the number of manually adde d dependencies just by adding one dependency.

```
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

- **The Test Starter**
  - For testing we usually use the following set of libraries: Spring Test, JUnit, Hamcrest, and Mockito. We can include all of these libraries manually, but Spring Boot starter can be used to automatically include these libraries in the following way:

    <dependency>

    <groupId>org.springframework.boot</groupId>

    <artifactId>spring-boot-starter-test</artifactId>

    <scope>test</scope>

    </dependency>

  - Notice that you don't need to specify the version number of an artifact. Spring Boot will figure out what version to use – all you need to specify is the version of spring-boot-starter-parent artifact. If later on you need to upgrade the Boot library and dependencies, just upgrade the Boot version in one place and it will take care of the rest.
- **There are two ways to test the controller:**
  - Using the mock environment
  - Using the embedded Servlet container (like Tomcat or Jetty)
- **The Data JPA Starter**
  - Most web applications have some sort of persistence – and that's quite often JPA.

  Instead of defining all of the associated dependencies manually – let's go with the starter instead:

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
   <groupId>com.h2database</groupId>
   <artifactId>h2</artifactId>
   <scope>runtime</scope>
</dependency>
```

Notice that out of the box we have automatic support for at least the following databases: H2, Derby and Hsqldb.

- **The Mail Starter**
  - A very common task in enterprise development is sending email, and dealing directly with Java Mail API usually can be difficult.
  - Spring Boot starter hides this complexity – mail dependencies can be specified in the following way:

    <dependency>

    <groupId>org.springframework.boot</groupId>

    <artifactId>spring-boot-starter-mail</artifactId>

    </dependency>

### 10.2.4 Advantages of Spring Boot

- Spring Framework can be employed on all architectural layers used in the development of web applications.
- Uses the very lightweight POJO model when writing classes.
- Allows you to freely link modules and easily test them.
- Supports declarative programming.
- Eliminates the need to independently create factory and singleton classes.
- Supports various configuration methods;
- Provides middleware-level service.

### 10.2.5 Disadvantages of Spring Boot

- Spring boot may include dependencies that are not used thereby causing huge deployment file size.
- Turning legacy spring applications into Spring boot requires a lot of effort and a time-consuming process.
- Limited control of your application.

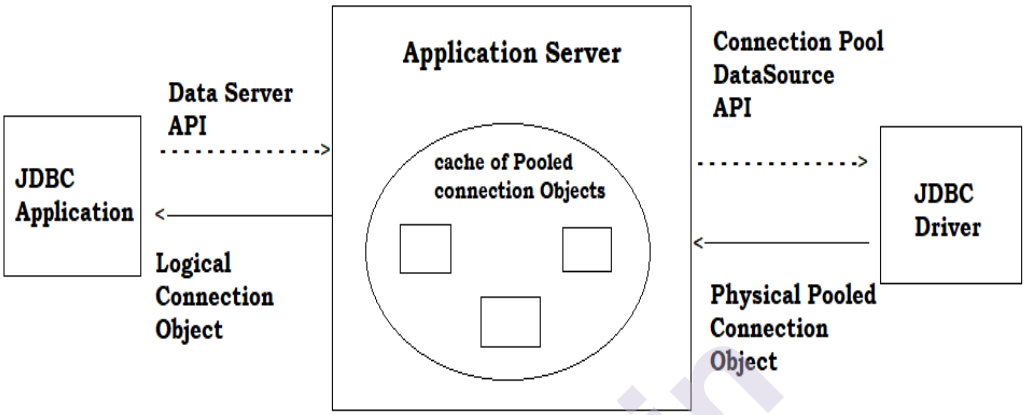
## 10.3 Spring Boot And Database

Spring Boot JDBC provides starter and libraries for connecting an application with JDBC.

### 10.3.1 JDBC Connection Pooling

Connection pooling is a mechanism to create and maintain a collection of JDBC connection objects. The primary objective of maintaining the pool of connection

object is to leverage re-usability. A new connection object is created only when there are no connection objects available to reuse. This technique can improve overall performance of the application. This article will try to show how this pooling mechanism can be applied to a Java application.




⇨ Diagram represents the structure of JDBC Connection Pooling.

It increases the speed of data access and reduces the number of database connections for an application. It also improves the performance of an application. Connection pool performs the following tasks:

- Manage available connection
- Allocate new connection
- Close connection

### 10.3.2 Why Do We Need Connection Pooling?

- Establishing a database connection is a very resource-intensive process and involves a lot of overhead. Moreover, in a multi-threaded environment, opening and closing a connection can worsen the situation greatly. To get a glimpse of what actually may happen with each request for creating new database connection, consider the following points. Database connections are established using either DriverManager of DataSource objects.
    - An application invokes the getConnection() method.
    - The JDBC driver requests a JVM socket.
    - JVM has to ensure that the call does not violate security aspects (as the case may be with applets).

- The invocation may have to percolate through a firewall before getting into the network cloud.
- On reaching the host, the server processes the connection request.

- The database server initializes the connection object and returns back to the JDBC client (again, going through the same process).
- And, finally, we get a connection object.
- This is just an overview of what actually goes on behind the scenes. Rest assured, the actual process is more complicated and elaborate than this. In a single-threaded controlled environment, database transactions are mostly linear, like opening a connection, doing database transaction, and closing the connection when done. Real-life applications are more complex; the mechanism of connection pooling can add to the performance although there are many other properties that are critical to overall performance of the application.
- The complexity of the concept of connection pooling gets nastier as we dive deep into it. But, thanks go to the people who work to produce libraries specifically for the cause of connection pooling. These libraries adhere to the norms of JDBC and provide simpler APIs to actually implement them in a Java application.

## 10.4 Spring Boot Web Application Developement

- Despite the advantages of Spring Framework, authors decided to provide developers with some utilities that automate the configuration procedure and speed up the process of creating and deploying Spring applications. These utilities were combined under the general name of Spring Boot.
- While the Spring Framework focuses on providing flexibility, Spring Boot seeks to reduce code length and simplify web application development. By leveraging annotation and boilerplate configuration, Spring Boot reduces the time it takes to develop applications. This capability helps you create standalone applications with less or almost no configuration overhead.
- how to configure a Spring Boot application to connect to MySQL database server, in these two common scenarios:
  - A Spring Boot console application with Spring JDBC and JdbcTemplate
  - A Spring Boot web application with Spring Data JPA and Hibernate framework

- Basically, in order to make a connection to a MySQL server, you need to do the following steps:
- Declare a dependency for MySQL JDBC driver, which enables Java application to communicate with MySQL server.
- Declare a dependency for Spring JDBC or Spring Data JPA
- Specify data source properties for the database connection information
- In case of Spring JDBC, use JdbcTemplate APIs for executing SQL statements against the database
- In case of Spring Data JPA, you need to create an entity class, a repository interface and then use the Spring Data JPA API.

Below are the details for connecting to MySQL server in a Spring Boot application.

**1. Declare dependency for MySQL JDBC Driver**

To use MySQL JDBC driver, declare the following dependency in the Maven pom.xml file of your Spring Boot project:

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
</dependency>
```

You don't need to specify the version as Spring Boot uses the default version specified in the parent POM.

**2. Specify Data Source Properties**

Next, you need to specify the following properties in the Spring Boot application configuration file (application.properties):

```
spring.datasource.url=jdbc:mysql://localhost:3306/bookshop
spring.datasource.username=root
spring.datasource.password=password
```

Update the data source URL, username and password according to your MySQL configuration. If you connect to a remote MySQL server, you need to replace localhost by IP address or hostname of the remote host.

### 3. Connect to MySQL with Spring JDBC

Spring JDBC provides a simple API on top of JDBC (JdbcTemplate), which you can use in simple cases, e.g. executing plain SQL statements. You need to declare the following dependency:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
```

### 4. Connect to MySQL with Spring Data JPA

Spring Data JPA provides more advanced API that greatly simplifies database programming based on Java Persistence API (JPA) specification with Hibernate as the implementation framework.

You need to declare dependency for Spring Data JPA as follows:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

For data source properties, before the URL, username and password you can also specify these additional properties:

```
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL57InnoDBDialect
```

And then, you need to code an entity class that maps to a table in the database, for example:

```
import javax.persistence.*;
@Entity
@Table(name = "users")
public class User {
   @Id
```

```
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    private String email;
    private String password;
      // getters and setters...
}
```

And declare a corresponding repository interface:

```
import org.springframework.data.jpa.repository.JpaRepository;
public interface UserRepository extends JpaRepository<User, Integer> {
}
```

And then you can use the repository in a Spring MVC controller or business class like this:

```
@Controller
public class UserController {
  @Autowired
  private UserRepository repo;
  @GetMapping("/users")
  public String listAll(Model model) {
    List<User> listUsers = repo.findAll();
    model.addAttribute("listUsers", listUsers);
    return "users";
  }
}
```

### 10.4.1 Ease of Dependency Management

- To speed up the dependency management process, Spring Boot implicitly packages the required third-party dependencies for each type of Spring-based application and provides them through so-called starter packages (spring-boot-starter-web, spring-boot-starter-data-jpa, etc.).
- Starter packages are collections of handy dependency descriptors that you can include in your application. They allow you to get a universal solution for all Spring-related technologies, removing the necessity to search for code examples and load the required dependency descriptors from them.
- For instance, if you want to start using Spring Data JPA to access your database, just include the spring-boot-starter-data-jpa dependency in your

project and you'll be done (no need to look for compatible Hibernate database drivers and libraries).

- If you want to create a Spring web application, just add the spring-boot-starter-web dependency, which will pull all the libraries you need to develop Spring MVC applications into your project, such as spring-webmvc, jackson-json, validation-api, and Tomcat.
- To say in a few words what is Spring Boot used for, it collects all common dependencies and defines them in one place, which allows developers to get to work right away instead of reinventing the wheel every time they create a new application.
- Therefore, the pom.xml file contains much fewer lines when used in Spring Boot than in regular Spring.

### 10.4.2 Automatic Configuration

- One of the advantages of Spring Boot, that is worth mentioning is automatic configuration of the application.
- After choosing a suitable starter package, Spring Boot will try to automatically configure your Spring application based on the jar dependencies you added. For example, if you add Spring-boot-starter-web, Spring Boot will automatically configure registered beans such as DispatcherServlet, ResourceHandlers, and MessageSource.
- If you are using spring-boot-starter-jdbc, Spring Boot automatically registers the DataSource, EntityManagerFactory, and TransactionManager beans and reads the database connection information from the application.properties file.
- If you are not going to use a database and do not provide any details about connecting manually, Spring Boot will automatically configure the database in the memory without any additional configuration on your part (if you have H2 or HSQL libraries). Automatic configuration can be completely overridden at any time by using user preferences.

### 10.4.3 Native Support for Application Server – Servlet Container

- Every Spring Boot application includes an embedded web server. Developers no longer have to worry about setting up a servlet container and deploying an application to it. The application can now run itself as an executable jar file using the built-in server. If you need to use a separate HTTP server, simply exclude the default dependencies. Spring Boot provides separate starter packages for different HTTP servers.

- Building stand-alone web applications with embedded servers is not only convenient for development but also a valid solution for enterprise-grade applications; what's more, it's becoming increasingly useful in the world of microservices. The ability to quickly package an entire service (such as user authentication) into a standalone and fully deployable artefact that also provides an API makes installing and deploying an application much easier.
- Spring Boot is part of the next generation of tools that simplify the configuration process for Spring applications. It is not a tool for automatic code generation, but a plugin for project build automation systems (supporting Maven and Gradle).
- The plugin provides capabilities for testing and deploying Spring applications. The mvn spring-boot:run command runs your application on port 8080. In addition, Spring Boot allows you to package your application into a separate jar file with a full Tomcat container embedded. This approach was borrowed from the Play framework's application deployment model (however, Spring Boot can also create traditional war files).
- One of the key advantages of Spring Boot is the configuration of resources based on the content of the classpath. Spring Boot is pretty intuitive when it comes to the default configuration. You may not always agree with its choice of settings, but at least it will provide you with a working module. This is a very useful approach, especially for novice developers who can start with the default settings and make changes to them as they explore the alternatives later down the line. This is much better than answering a bunch of difficult questions every time you begin a new project.
- In addition, there are a number of full-fledged tutorials on Spring Boot's official webpage. These will help you quickly understand and practically implement all the main types of projects at the initial level.
- Spring Boot is still in its infancy, and it will naturally have to go through many metamorphoses before becoming fully stable. It may be too early to use it for building serious systems, but it is quite suitable for performing all sorts of personal, training, and test projects, in case you want to eliminate large amounts of unproductive, routine work that is in no way related to the creation of useful functionality.
- As far as Spring Boot's potential for growing into a serious development tool is concerned, the presence of acceptable technical documentation looks very encouraging.

## 10.5 Spring Boot Restful Webservices

- REST stands for REpresentational State Transfer. It is developed by Roy Thomas Fielding, who also developed HTTP. The main goal of RESTful web services is to make web services more effective. RESTful web services try to define services using the different concepts that are already present in HTTP. REST is an architectural approach, not a protocol.
- It does not define the standard message exchange format. We can build REST services with both XML and JSON. JSON is more popular format with REST. The key abstraction is a resource in REST. A resource can be anything. It can be accessed through a Uniform Resource Identifier (URI).

### 10.5.1 Why REST is popular:

1. It allows the separation between the client and the server.
2. It doesn't rely on a single technology or programming language.
3. You can build the scalable application or even integrate two different applications using REST APIs
4. REST is stateless and uses basic HTTP methods like GET, PUT, POST, and DELETE, etc. for communication between client and server.

➢ The resource has representations like XML, HTML, and JSON. The current state capture by representational resource. When we request a resource, we provide the representation of the resource. The important methods of HTTP are:

GET: It reads a resource.

PUT: It updates an existing resource.

POST: It creates a new resource.

DELETE: It deletes the resource.

For example, if we want to perform the following actions in the social media application, we get the corresponding results.

POST /users: It creates a user.

GET /users/{id}: It retrieves the detail of a user.

GET /users: It retrieves the detail of all users.

DELETE /users: It deletes all users.

DELETE /users/{id}: It deletes a user.

GET /users/{id}/posts/post_id: It retrieve the detail of a specific post.

POST / users/{id}/ posts: It creates a post of the user.

**Rest Controller**

The @RestController annotation is used to define the RESTful web services. It serves JSON, XML and custom response. Its syntax is shown below −

```
@RestController
public class ProductServiceController {
}
```

**Request Mapping**

The @RequestMapping annotation is used to define the Request URI to access the REST Endpoints. We can define Request method to consume and produce object. The default request method is GET.

```
@RequestMapping(value = "/products")
public ResponseEntity<Object> getProducts() { }
```

**Request Body**

The @RequestBody annotation is used to define the request body content type.

```
public ResponseEntity<Object> createProduct(@RequestBody Product product)
{
}
```

**Path Variable**

The @PathVariable annotation is used to define the custom or dynamic request URI. The Path variable in request URI is defined as curly braces {} as shown below –

```
public ResponseEntity<Object> updateProduct(@PathVariable("id") String id) {
}
```

**Request Parameter**

The @RequestParam annotation is used to read the request parameters from the Request URL. By default, it is a required parameter. We can also set default value for request parameters as shown here −

```
public ResponseEntity<Object> getProduct(
   @RequestParam(value = "name", required = false, defaultValue = "honey")
String name) {
}
```

**GET API**

The default HTTP request method is GET. This method does not require any Request Body. You can send request parameters and path variables to define the custom or dynamic URL.

**POST API**

The HTTP POST request is used to create a resource. This method contains the Request Body. We can send request parameters and path variables to define the custom or dynamic URL.

**PUT API**

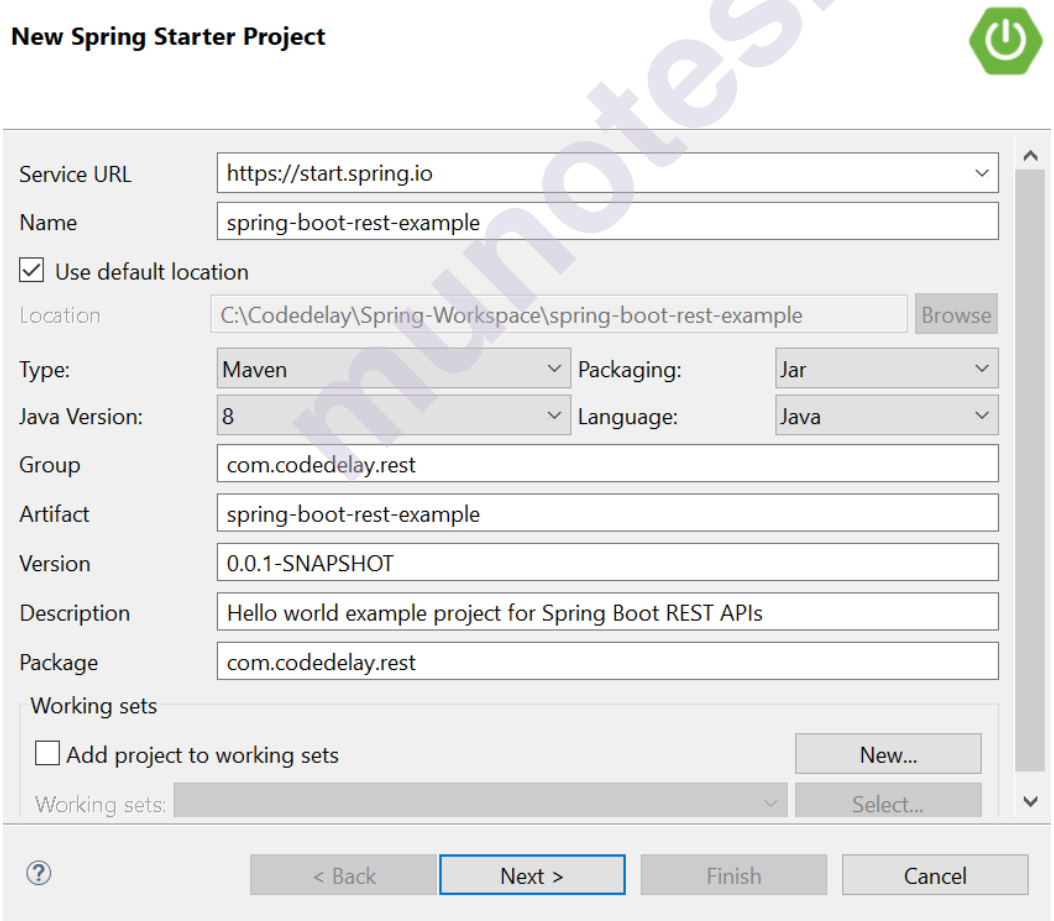
The HTTP PUT request is used to update the existing resource. This method contains a Request Body. We can send request parameters and path variables to define the custom or dynamic URL.

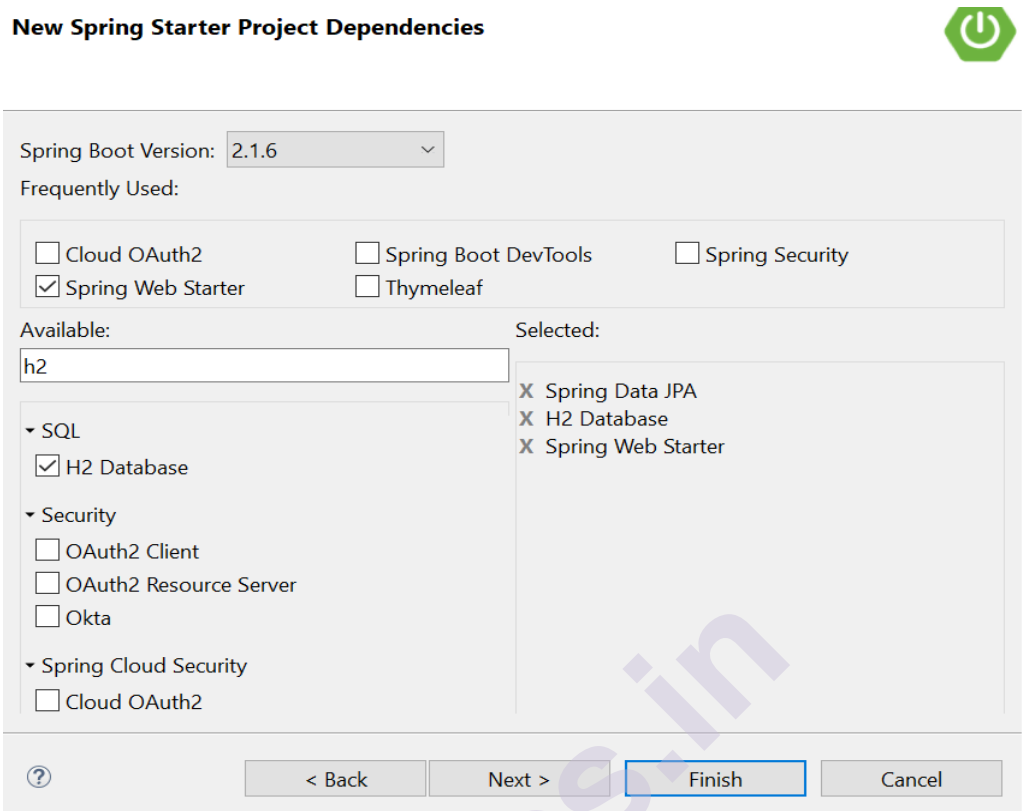
### 6.5.2 Spring Boot REST Example

To create the Spring Boot REST program, you have to follow below four steps:

1. Create a new project using the Spring Tool Suite or Spring Initializr Project.
2. Add dependencies in the Maven POM file.
3. Add a controller and expose REST APIs.
4. Add a service layer to communicate with Spring Data JPA's CrudRepository interface.

Create a new Spring Boot Project






Spring boot rest example dependencies:

As you could see in the above example, I have created a new project spring-boot-rest-example and added dependencies for Spring Data JPA, H2 Database, and Spring Web Starter.

You can verify those dependencies in pom.xml.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
        <modelVersion>4.0.0</modelVersion>
        <parent>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-parent</artifactId>
                <version>2.1.6.RELEASE</version>
                <relativePath /> <! – lookup parent from repository – >
        </parent>
        <groupId>com.codedelay.rest</groupId>
        <artifactId>spring-boot-rest-example</artifactId>
        <version>0.0.1-SNAPSHOT</version>
        <name>spring-boot-rest-example</name>
```

```
    <description>Hello world example project for Spring Boot REST APIs
</description>
    <properties>
        <java.version>1.8</java.version>
        <maven-jar-plugin.version>3.1.1</maven-jar-plugin.version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>com.h2database</groupId>
            <artifactId>h2</artifactId>
            <scope>runtime</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-devtools</artifactId>
            <scope>runtime</scope>
            <optional>true</optional>
        </dependency>
    </dependencies>
    <build>
        <plugins>
            <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
</project>
```

**Maven Dependency Tree**

[INFO] com.codedelay.rest:spring-boot-rest-example:jar:0.0.1-SNAPSHOT
[INFO] +- org.springframework.boot:spring-boot-starter-data-jpa:jar:2.1.6.RELEASE:compile
[INFO] | +- org.springframework.boot:spring-boot-starter-aop:jar:2.1.6.RELEASE:compile
[INFO] | | +- org.springframework:spring-aop:jar:5.1.8.RELEASE:compile
[INFO] | | \- org.aspectj:aspectjweaver:jar:1.9.4:compile
[INFO] | +- org.springframework.boot:spring-boot-starter-jdbc:jar:2.1.6.RELEASE:compile
[INFO] | | +- com.zaxxer:HikariCP:jar:3.2.0:compile
[INFO] | | \- org.springframework:spring-jdbc:jar:5.1.8.RELEASE:compile
[INFO] | +- javax.transaction:javax.transaction-api:jar:1.3:compile
[INFO] | +- javax.xml.bind:jaxb-api:jar:2.3.1:compile
[INFO] | | \- javax.activation:javax.activation-api:jar:1.2.0:compile
[INFO] | +- org.hibernate:hibernate-core:jar:5.3.10.Final:compile
[INFO] | | +- org.jboss.logging:jboss-logging:jar:3.3.2.Final:compile
[INFO] | | +- javax.persistence:javax.persistence-api:jar:2.2:compile
[INFO] | | +- org.javassist:javassist:jar:3.23.2-GA:compile
[INFO] | | +- net.bytebuddy:byte-buddy:jar:1.9.13:compile
[INFO] | | +- antlr:antlr:jar:2.7.7:compile
[INFO] | | +- org.jboss:jandex:jar:2.0.5.Final:compile
[INFO] | | +- com.fasterxml:classmate:jar:1.4.0:compile
[INFO] | | +- org.dom4j:dom4j:jar:2.1.1:compile
[INFO] | | \- org.hibernate.common:hibernate-commons-annotations:jar:5.0.4.Final:compile
[INFO] | +- org.springframework.data:spring-data-jpa:jar:2.1.9.RELEASE:compile
[INFO] | | +- org.springframework.data:spring-data-commons:jar:2.1.9.RELEASE:compile
[INFO] | | +- org.springframework:spring-orm:jar:5.1.8.RELEASE:compile
[INFO] | | +- org.springframework:spring-context:jar:5.1.8.RELEASE:compile
[INFO] | | +- org.springframework:spring-tx:jar:5.1.8.RELEASE:compile
[INFO] | | +- org.springframework:spring-beans:jar:5.1.8.RELEASE:compile
[INFO] | | \- org.slf4j:slf4j-api:jar:1.7.26:compile

```
[INFO] |  \- org.springframework:spring-aspects:jar:5.1.8.RELEASE:compile
[INFO] +- org.springframework.boot:spring-boot-starter-
web:jar:2.1.6.RELEASE:compile
[INFO] |  +- org.springframework.boot:spring-boot-
starter:jar:2.1.6.RELEASE:compile
[INFO] |  |  +- org.springframework.boot:spring-boot-starter-
logging:jar:2.1.6.RELEASE:compile
[INFO] |  |  |  +- ch.qos.logback:logback-classic:jar:1.2.3:compile
[INFO] |  |  |  |  \- ch.qos.logback:logback-core:jar:1.2.3:compile
[INFO] |  |  |  +- org.apache.logging.log4j:log4j-to-slf4j:jar:2.11.2:compile
[INFO] |  |  |  |  \- org.apache.logging.log4j:log4j-api:jar:2.11.2:compile
[INFO] |  |  |  \- org.slf4j:jul-to-slf4j:jar:1.7.26:compile
[INFO] |  |  +- javax.annotation:javax.annotation-api:jar:1.3.2:compile
[INFO] |  |  \- org.yaml:snakeyaml:jar:1.23:runtime
[INFO] |  +- org.springframework.boot:spring-boot-starter-
json:jar:2.1.6.RELEASE:compile
[INFO] |  |  +- com.fasterxml.jackson.core:jackson-databind:jar:2.9.9:compile
[INFO] |  |  |  +- com.fasterxml.jackson.core:jackson-annotations:jar:2.9.0:compile
[INFO] |  |  |  \- com.fasterxml.jackson.core:jackson-core:jar:2.9.9:compile
[INFO] |  |  +- com.fasterxml.jackson.datatype:jackson-datatype-
jdk8:jar:2.9.9:compile
[INFO] |  |  +- com.fasterxml.jackson.datatype:jackson-datatype-
jsr310:jar:2.9.9:compile
[INFO] |  |  \- com.fasterxml.jackson.module:jackson-module-parameter-
names:jar:2.9.9:compile
[INFO] |  +- org.springframework.boot:spring-boot-starter-
tomcat:jar:2.1.6.RELEASE:compile
[INFO] |  |  +- org.apache.tomcat.embed:tomcat-embed-core:jar:9.0.21:compile
[INFO] |  |  +- org.apache.tomcat.embed:tomcat-embed-el:jar:9.0.21:compile
[INFO] |  |  \- org.apache.tomcat.embed:tomcat-embed-
websocket:jar:9.0.21:compile
[INFO] |  +- org.hibernate.validator:hibernate-validator:jar:6.0.17.Final:compile
[INFO] |  |  \- javax.validation:validation-api:jar:2.0.1.Final:compile
[INFO] |  +- org.springframework:spring-web:jar:5.1.8.RELEASE:compile
[INFO] |  \- org.springframework:spring-webmvc:jar:5.1.8.RELEASE:compile
```

[INFO] | \- org.springframework:spring-expression:jar:5.1.8.RELEASE:compile

[INFO] +- com.h2database:h2:jar:1.4.199:runtime

[INFO] +- org.springframework.boot:spring-boot-starter-test:jar:2.1.6.RELEASE:test

[INFO] | +- org.springframework.boot:spring-boot-test:jar:2.1.6.RELEASE:test

[INFO] | +- org.springframework.boot:spring-boot-test-autoconfigure:jar:2.1.6.RELEASE:test

[INFO] | +- com.jayway.jsonpath:json-path:jar:2.4.0:test

[INFO] | | \- net.minidev:json-smart:jar:2.3:test

[INFO] | | \- net.minidev:accessors-smart:jar:1.2:test

[INFO] | | \- org.ow2.asm:asm:jar:5.0.4:test

[INFO] | +- junit:junit:jar:4.12:test

[INFO] | +- org.assertj:assertj-core:jar:3.11.1:test

[INFO] | +- org.mockito:mockito-core:jar:2.23.4:test

[INFO] | | +- net.bytebuddy:byte-buddy-agent:jar:1.9.13:test

[INFO] | | \- org.objenesis:objenesis:jar:2.6:test

[INFO] | +- org.hamcrest:hamcrest-core:jar:1.3:test

[INFO] | +- org.hamcrest:hamcrest-library:jar:1.3:test

[INFO] | +- org.skyscreamer:jsonassert:jar:1.5.0:test

[INFO] | | \- com.vaadin.external.google:android-json:jar:0.0.20131108.vaadin1:test

[INFO] | +- org.springframework:spring-core:jar:5.1.8.RELEASE:compile

[INFO] | | \- org.springframework:spring-jcl:jar:5.1.8.RELEASE:compile

[INFO] | +- org.springframework:spring-test:jar:5.1.8.RELEASE:test

[INFO] | \- org.xmlunit:xmlunit-core:jar:2.6.2:test

[INFO] \- org.springframework.boot:spring-boot-devtools:jar:2.1.6.RELEASE:runtime (optional)

[INFO] +- org.springframework.boot:spring-boot:jar:2.1.6.RELEASE:compile

[INFO] \- org.springframework.boot:spring-boot-autoconfigure:jar:2.1.6.RELEASE:compile

Controller to expose REST APIs

For this tutorial, we will CRUD APIs for User Management System.

By using these APIs we can add, retrieve, update, or delete the user details from the database.

To create a User Management System, let's focus on writing 5 basic APIs

HTTP GET /getAll will return a list of all user details.
HTTP GET /find/{id} will return a user's detail by an id.
HTTP POST /add is to add a user into the database.
HTTP PUT /update/{id} can be used to update a user based on an id.
HTTP DELETE /delete/{id} can be used to delete a user from the database.

Let's create a controller (UserController) to expose REST endpoints.

```
package com.codedelay.rest.controller;
import javax.validation.Valid;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.bind.annotation.RestController;
import com.codedelay.rest.entity.User;
import com.codedelay.rest.service.UserManageService;
@RestController
@RequestMapping("/api/user")
public class UserController {
	@Autowired
	private UserManageService mService;
	@GetMapping("/getAll")
	public Iterable<User> getAllUsers() {
		return mService.getAllUsers();
	}
	@PostMapping("/add")
	@ResponseStatus(HttpStatus.CREATED)
	public User addUser(@Valid @RequestBody User user) {
		return mService.addUser(user);
	}
	@GetMapping("/find/{id}")
```

```
	public User findUserById(@PathVariable("id") int id) {
		return mService.findUserById(id);
	}

	@PutMapping("/update/{id}")
	public User addOrUpdateUserById(@RequestBody User user,
@PathVariable("id") int id) {
		return mService.addOrUpdateUserById(user, id);
	}
	@DeleteMapping("/delete/{id}")
	public void deleteUser(@PathVariable("id") int id) {
		mService.deleteUser(id);
	}
}
```

**Service layer:**

The service layer acts as an intermediate layer between a controller and a repository class.

```
package com.codedelay.rest.service;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import com.codedelay.rest.dao.UserRepository;
import com.codedelay.rest.entity.User;
import com.codedelay.rest.exception.UserNotFoundException;
@Service
public class UserManageService {
	@Autowired
	private UserRepository mRepository;
	public Iterable<User> getAllUsers() {
		return mRepository.findAll();
	}
	public User addUser(User user) {
		return mRepository.save(user);
	}
	public User findUserById(int id) {
		return mRepository.findById(id).get();
	}
```

```
	public User addOrUpdateUserById(User user, int id) {
		return mRepository.findById(id).map(x -> {
			x.setUser_name(user.getUser_name());
			x.setPassword(user.getPassword());
			return mRepository.save(x);
		}).orElseGet(() -> {
			user.setUser_id(id);
			return mRepository.save(user);
		});
	}
	public void deleteUser(int id) {
		mRepository.deleteById(id);
	}
}
```

**Entity class:**

Now, let's create an entity class which is a simple POJO class annotated with JPA annotations.

Entity class also represents a table in the database.

In our case, we will create a User class inside com.codedelay.rest.entity package.

```
package com.codedelay.rest.entity;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;
@Entity
@Table(name = "user_details")
public class User {
	@Id
	@GeneratedValue(strategy = GenerationType.AUTO)
	private int user_id;
	@Column(unique = true, nullable = false, length = 10)
	private String user_name;
	@Column(nullable = false, length = 12, updatable = true)
	private String password;
	public int getUser_id() {
```

```
        return user_id;
    }
    public void setUser_id(int user_id) {
        this.user_id = user_id;
    }
    public String getUser_name() {
        return user_name;
    }
    public void setUser_name(String user_name) {
        this.user_name = user_name;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }
}
```

**Write the Repository Interface.**

```
package com.codedelay.rest.dao;
import org.springframework.data.repository.CrudRepository;
import org.springframework.stereotype.Repository;
import com.codedelay.rest.entity.User;
public interface UserRepository extends CrudRepository<User, Integer> {
}
```

UserRepository interface extends CrudRepository.

CrudRepository is a magical interface from Spring Data JPA.

It allows writing simple CRUD functions without writing a single line of code.

**Exception handling in Spring Boot REST**

A good REST API also covers exception scenarios.

Let discuss one simple scenario.

What will happen if HTTP GET /find/{id} doesn't find a particular user in the database?

It should throw an exception. Isn't it?

Let's add one more class UserNotFoundException class in com.codedelay.rest.exception package.

```
package com.codedelay.rest.exception;
public class UserNotFoundException extends RuntimeException  {
        public UserNotFoundException(int id) {
                super("User id not found : " + id);
        }
}
```

Now add the Service class to throw the UserNotFoundException exception if there are no user details available in the database for that particular user id.

```
public User findUserById(int id) {
                return mRepository.findById(id).orElseThrow(() -> new
UserNotFoundException(id));
        }
```

It is not sufficient to throw java exception.

We have to return some HTTP error when UserNotFoundException occurs.

For this, let's create a class GlobalExceptionHandler which will return HttpStatus.NOT_FOUND error when UserNotFoundException occurs.

```
package com.codedelay.rest.exception;
import java.io.IOException;
import javax.servlet.http.HttpServletResponse;
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
@ControllerAdvice
public class GlobalExceptionHandler {
        @ExceptionHandler(UserNotFoundException.class)
   public void handleUserNotFoundError(HttpServletResponse response) throws
IOException {
      response.sendError(HttpStatus.NOT_FOUND.value());
   }
}
```

## 10.6 Conclusion

Spring Boot has become an integral part of the Java ecosystem, offering an efficient and scalable toolbox for building Spring applications with a microservices architecture. It speeds up the development and deployment

processes by using intuitive default settings for unit and integration tests. What's more, Spring Boot helps developers build robust applications with clear and secure configurations without spending a lot of time and effort on figuring out the intricacies of Spring. If you're not sure about whether or not you should use this solution for your Java project, carefully review the pros and cons of using Spring Boot, its core features, and see how they align with your business goals. Alternatively, you can entrust a reliable software vendor with the development process.

## 10.7 List Of References

https://docs.spring.io/spring-boot/docs/current/reference/html/using.html

https://bambooagile.eu/insights/pros-and-cons-of-using-spring-boot/

❖❖❖❖❖

# 11

# UNDERSTANDING TRANSACTION MANAGEMENT IN SPRING

**Unit Structure**



## 11.0 Objectives

Any application involves a number of services or components making a call to other services or components. Transaction propagation indicates if any component or service will or will not participate in a transaction and how will it behave if the calling component/service already has or does not have a transaction created already.

One of the most compelling reasons to use the Spring Framework is the comprehensive transaction support. The Spring Framework provides a consistent abstraction for transaction management that delivers the following benefits:

- Provides a consistent programming model across different transaction APIs such as JTA, JDBC, Hibernate, JPA, and JDO.
- Supports declarative transaction management.
- Provides a simpler API for programmatic transaction management than a number of complex transaction APIs such as JTA.
- Integrates very well with Spring's various data access abstractions.

This chapter is divided up into a number of sections, each detailing one of the value-adds or technologies of the Spring Framework's transaction support. The chapter closes up with some discussion of best practices surrounding transaction

management (for example, choosing between declarative and programmatic transaction management).

- The first section, entitled Motivations, describes *why* one would want to use the Spring Framework's transaction abstraction as opposed to EJB CMT or driving transactions via a proprietary API such as Hibernate.
- The second section, entitled Key abstractions outlines the core classes in the Spring Framework's transaction support, as well as how to configure and obtain DataSource instances from a variety of sources.
- The third section, entitled Declarative transaction management, covers the Spring Framework's support for declarative transaction management.
- The fourth section, entitled Programmatic transaction management, covers the Spring Framework's support for programmatic (that is, explicitly coded) transaction management.

## 11.1 Introduction

A transaction is an action or series of actions that are being performed by a single user or application program, which reads or updates the contents of the database.

A transaction can be defined as a logical unit of work on the database. This may be an entire program, a piece of a program, or a single command (like the SQL commands such as INSERT or UPDATE), and it may engage in any number of operations on the database. In the database context, the execution of an application program can be thought of as one or more transactions with non-database processing taking place in between.

In traditional relational database design, transactions are completed by COMMIT or ROLLBACK SQL statements, which indicate a transaction's beginning or end. The ACID acronym defines the

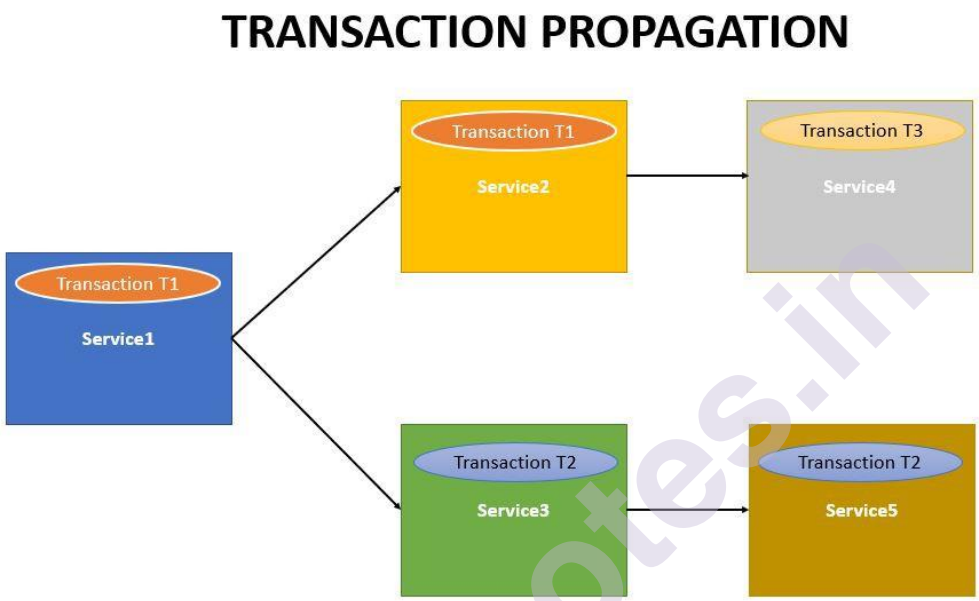
properties of a database transaction, as follows:

- **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:** Transaction data changes must be available, even in the event of database failure.

## 11.2 An Overview

Transaction represents the series of different actions.

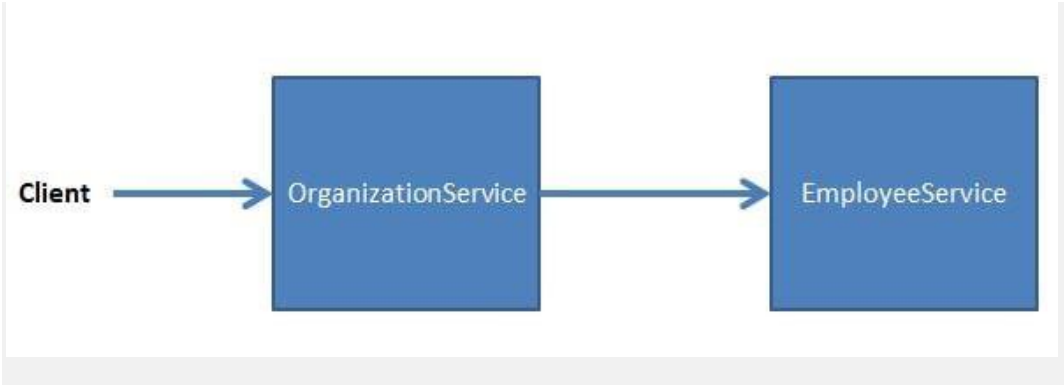




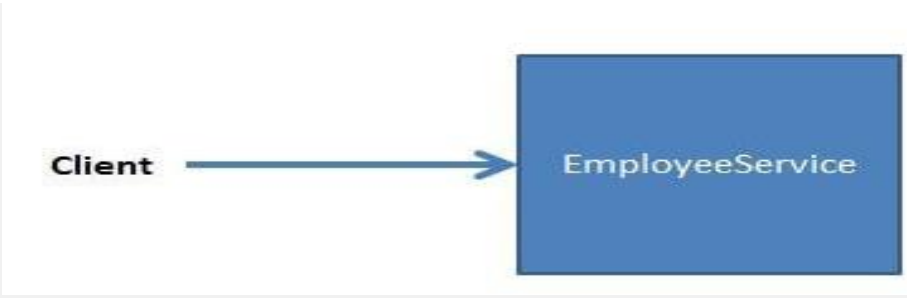

Below diagram represents the propagation of Transaction.

⇨ We will discuss the Client-side example to represent the propagation of transaction in Spring as follows:

- Call using Organization service



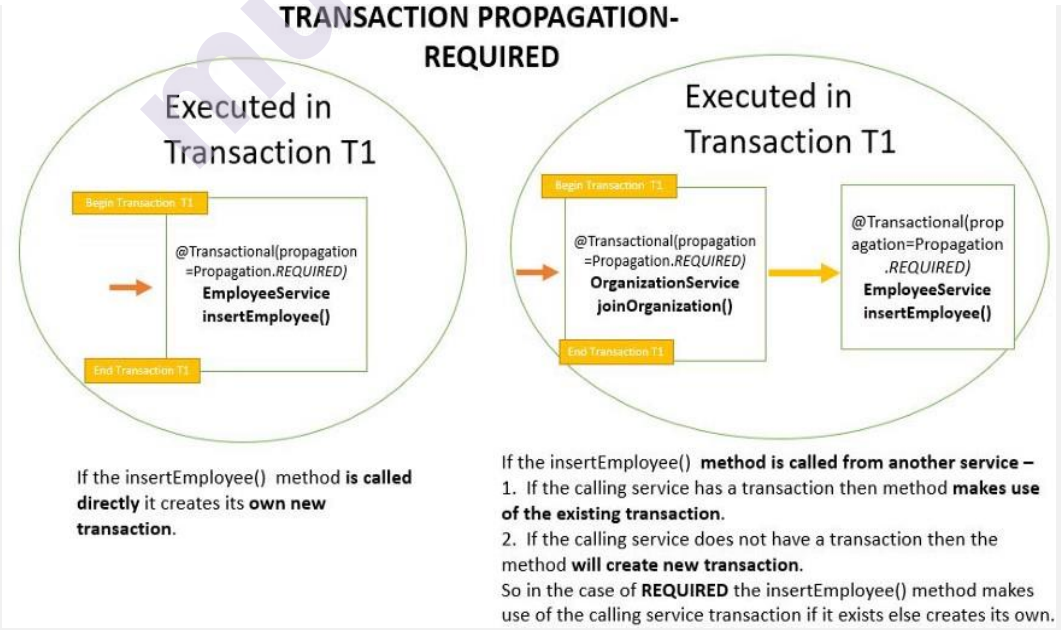

- Call the the Employee Service directly.




- As the Employee Service may also be called directly we will need to use Transaction annotation with Employee Service also. So both the services — Organization Service and the Employee Service will be using Transaction annotation.
- We will be looking at the various propagation scenarios by observing the behaviour of the Organization and Employee service.There are six types of Transaction Propagations-

1) **REQUIRED**
2) **SUPPORTS**
3) **NOT_SUPPORTED**
4) **REQUIRES_NEW**
5) **NEVER**
6) **MANDATORY**

### 13.2.1 Transaction Propagation (Required):



Here both the Organization Service and the Employee Service have the transaction propagation defined as **Required**. This is the default Transaction Propagation.

**Code-**

The Organization Service will be as follows-

```
package com.javainuse.service.impl;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import com.javainuse.model.Employee;
import com.javainuse.model.EmployeeHealthInsurance;
import com.javainuse.service.EmployeeService;
import com.javainuse.service.HealthInsuranceService;
import com.javainuse.service.OrganizationService;
@Service
@Transactional
public class OrganzationServiceImpl implements OrganizationService {
	@Autowired
	EmployeeService employeeService;
	@Autowired
	HealthInsuranceService healthInsuranceService;
	@Override
	public void joinOrganization(Employee employee,
EmployeeHealthInsurance employeeHealthInsurance) {
		employeeService.insertEmployee(employee);
		if (employee.getEmpId().equals("emp1")) {
			throw new RuntimeException("thowing exception to test
transaction rollback");
		}
	healthInsuranceService.registerEmployeeHealthInsurance(employeeHealthI
nsurance);
	}
	@Override
	public void leaveOrganization(Employee employee,
EmployeeHealthInsurance employeeHealthInsurance) {
		employeeService.deleteEmployeeById(employee.getEmpId());
	healthInsuranceService.deleteEmployeeHealthInsuranceById(employeeHea
lthInsurance.getEmpId());
```

```
	}
}
```

The Employee Service will be as follows-

```
package com.javainuse.service.impl;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import com.javainuse.dao.EmployeeDao;
import com.javainuse.model.Employee;
import com.javainuse.service.EmployeeService;
@Service
@Transactional
public class EmployeeServiceImpl implements EmployeeService {
	@Autowired
	EmployeeDao employeeDao;
	@Override
	public void insertEmployee(Employee employee) {
	employeeDao.insertEmployee(employee);
	}
	@Override
	public void deleteEmployeeById(String empid) {
		employeeDao.deleteEmployeeById(empid);
	}
}
```

**Output:**

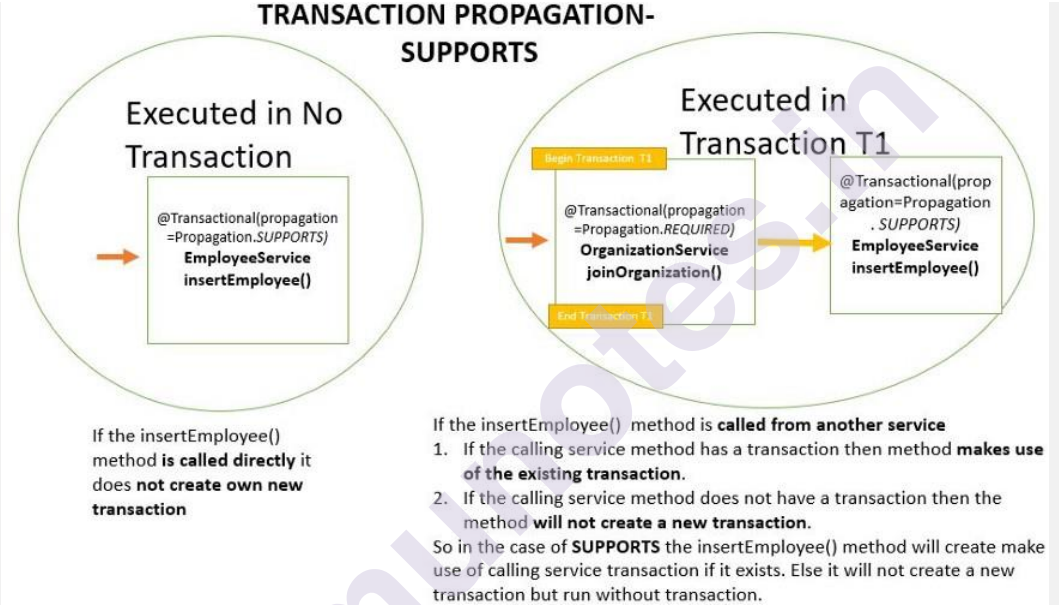
**EmployeeService called using OrganizationService -**

```
main] com.javainuse.SpringBootJdbcApplication  : Started SpringBootJdbcApplication in 3.896 seconds (JVM runni
main] o.s.j.d.DataSourceTransactionManager     : Creating new transaction with name [com.javainuse.service.imp
main] o.s.j.d.DataSourceTransactionManager     : Acquired Connection [HikariProxyConnection@1801021153 wrappin
main] o.s.j.d.DataSourceTransactionManager     : Switching JDBC Connection [HikariProxyConnection@1801021153 w
main] o.s.j.d.DataSourceTransactionManager     : Participating in existing transaction
main] o.s.jdbc.core.JdbcTemplate               : Executing prepared SQL update
main] o.s.jdbc.core.JdbcTemplate               : Executing prepared SQL statement [INSERT INTO employee (empId
main] o.s.j.d.DataSourceTransactionManager     : Participating in existing transaction
main] o.s.jdbc.core.JdbcTemplate               : Executing prepared SQL update
main] o.s.jdbc.core.JdbcTemplate               : Executing prepared SQL statement [INSERT INTO employeeHealthI
main] o.s.j.d.DataSourceTransactionManager     : Initiating transaction commit
main] o.s.j.d.DataSourceTransactionManager     : Committing JDBC transaction on Connection [HikariProxyConnect
main] o.s.j.d.DataSourceTransactionManager     : Releasing JDBC Connection [HikariProxyConnection@1801021153 w
main] o.s.jdbc.datasource.DataSourceUtils      : Returning JDBC Connection to DataSource
ead-2] s.c.a.AnnotationConfigApplicationContext : Closing org.springframework.context.annotation.AnnotationConf
ead-2] o.s.j.e.a.AnnotationMBeanExporter        : Unregistering JMX-exposed beans on shutdown
ead-2] o.s.j.e.a.AnnotationMBeanExporter        : Unregistering JMX-exposed beans
ead-2] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Shutdown initiated...
ead-2] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Shutdown completed.
```

**EmployeeService called directly -**

```
com.javainuse.SpringBootJdbcApplication  : Started SpringBootJdbcApplication in 5.141 seconds (JVM ru
o.s.j.d.DataSourceTransactionManager     : Creating new transaction with name [com.javainuse.service.
o.s.j.d.DataSourceTransactionManager     : Acquired Connection [HikariProxyConnection@809383315 wrapp
o.s.j.d.DataSourceTransactionManager     : Switching JDBC Connection [HikariProxyConnection@809383315
o.s.jdbc.core.JdbcTemplate               : Executing prepared SQL update
o.s.jdbc.core.JdbcTemplate               : Executing prepared SQL statement [INSERT INTO employee (em
o.s.j.d.DataSourceTransactionManager     : Initiating transaction commit
o.s.j.d.DataSourceTransactionManager     : Committing JDBC transaction on Connection [HikariProxyConn
o.s.j.d.DataSourceTransactionManager     : Releasing JDBC Connection [HikariProxyConnection@809383315
o.s.jdbc.datasource.DataSourceUtils      : Returning JDBC Connection to DataSource
s.c.a.AnnotationConfigApplicationContext : Closing org.springframework.context.annotation.AnnotationC
o.s.j.e.a.AnnotationMBeanExporter        : Unregistering JMX-exposed beans on shutdown
o.s.j.e.a.AnnotationMBeanExporter        : Unregistering JMX-exposed beans
com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Shutdown initiated...
com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Shutdown completed.
```

## 11.2.2 Transaction Propagation (Supports):




Here both the Organization Service has the transaction propagation defined as **Required** while Employee Service the transaction propagation is defined as **Supports**.

**Code-**

```
package com.javainuse.service.impl;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import com.javainuse.model.Employee;
import com.javainuse.model.EmployeeHealthInsurance;
import com.javainuse.service.EmployeeService;
import com.javainuse.service.HealthInsuranceService;
import com.javainuse.service.OrganizationService;
```

```
@Service
@Transactional
public class OrganzationServiceImpl implements OrganizationService {
@Autowired
EmployeeService employeeService;
@Autowired
HealthInsuranceService healthInsuranceService;
@Override
public void joinOrganization(Employee employee, EmployeeHealthInsurance
employeeHealthInsurance) {
employeeService.insertEmployee(employee);
if (employee.getEmpId().equals("emp1")) {
throw new RuntimeException("thowing exception to test transaction rollback");
}
healthInsuranceService.registerEmployeeHealthInsurance(employeeHealthInsura
nce);
}
@Override
public void leaveOrganization(Employee employee, EmployeeHealthInsurance
employeeHealthInsurance) {
employeeService.deleteEmployeeById(employee.getEmpId());
healthInsuranceService.deleteEmployeeHealthInsuranceById(employeeHealthIns
urance.getEmpId());
}
}
```

The Employee Service will be as follows-

```
package com.javainuse.service.impl;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import com.javainuse.dao.EmployeeDao;
import com.javainuse.model.Employee;
import com.javainuse.service.EmployeeService;
@Service
@Transactional(propagation=Propagation.SUPPORTS)
public class EmployeeServiceImpl implements EmployeeService {
@Autowired
EmployeeDao employeeDao;
@Override
public void insertEmployee(Employee employee) {
employeeDao.insertEmployee(employee);
}
```

```
@Override
public void deleteEmployeeById(String empid) {
employeeDao.deleteEmployeeById(empid);
}
}
```

**Output:**

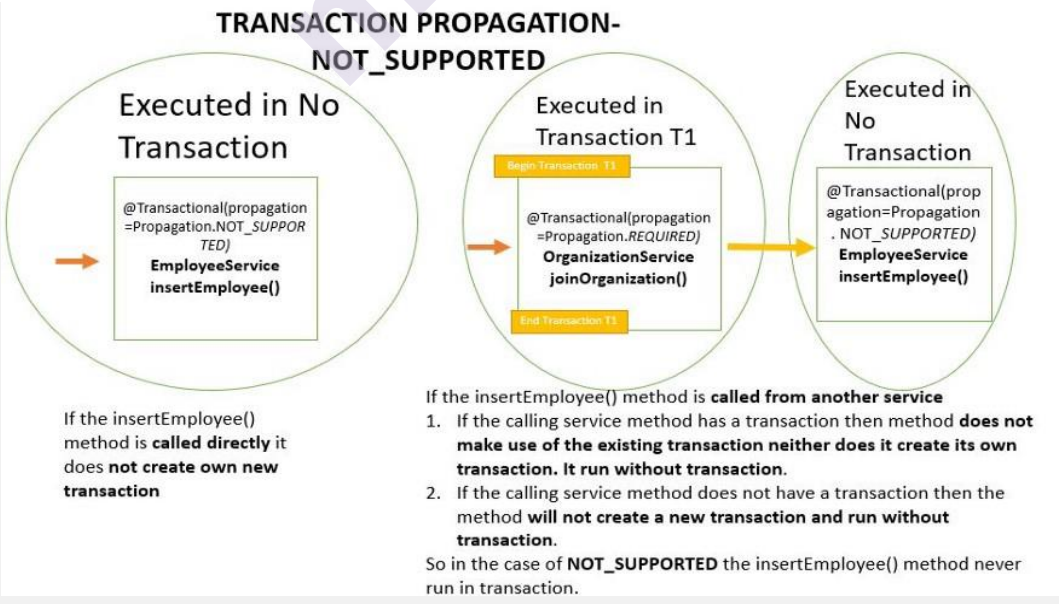
**EmployeeService called using OrganizationService -**

```
main] com.javainuse.SpringBootJdbcApplication  : Started SpringBootJdbcApplication in 4.361 seconds (JVM running f
main] o.s.j.d.DataSourceTransactionManager     : Creating new transaction with name [com.javainuse.service.impl.Or
main] o.s.j.d.DataSourceTransactionManager     : Acquired Connection [HikariProxyConnection@1134202713 wrapping co
main] o.s.j.d.DataSourceTransactionManager     : Switching JDBC Connection [HikariProxyConnection@1134202713 wrapp
main] o.s.j.d.DataSourceTransactionManager     : Participating in existing transaction
main] o.s.jdbc.core.JdbcTemplate               : Executing prepared SQL update
main] o.s.jdbc.core.JdbcTemplate               : Executing prepared SQL statement [INSERT INTO employee (empId, em
main] o.s.j.d.DataSourceTransactionManager     : Participating in existing transaction
main] o.s.jdbc.core.JdbcTemplate               : Executing prepared SQL update
main] o.s.jdbc.core.JdbcTemplate               : Executing prepared SQL statement [INSERT INTO employeeHealthInsur
main] o.s.j.d.DataSourceTransactionManager     : Initiating transaction commit
main] o.s.j.d.DataSourceTransactionManager     : Committing JDBC transaction on Connection [HikariProxyConnection@
main] o.s.j.d.DataSourceTransactionManager     : Releasing JDBC Connection [HikariProxyConnection@1134202713 wrapp
main] o.s.jdbc.datasource.DataSourceUtils      : Returning JDBC Connection to DataSource
```

**EmployeeService called directly -**

```
main] com.javainuse.SpringBootJdbcApplication  : Started SpringBootJdbcApplication in 4.361 seconds (JVM running f
main] o.s.j.d.DataSourceTransactionManager     : Creating new transaction with name [com.javainuse.service.impl.Or
main] o.s.j.d.DataSourceTransactionManager     : Acquired Connection [HikariProxyConnection@1134202713 wrapping co
main] o.s.j.d.DataSourceTransactionManager     : Switching JDBC Connection [HikariProxyConnection@1134202713 wrapp
main] o.s.j.d.DataSourceTransactionManager     : Participating in existing transaction
main] o.s.jdbc.core.JdbcTemplate               : Executing prepared SQL update
main] o.s.jdbc.core.JdbcTemplate               : Executing prepared SQL statement [INSERT INTO employee (empId, em
main] o.s.j.d.DataSourceTransactionManager     : Participating in existing transaction
main] o.s.jdbc.core.JdbcTemplate               : Executing prepared SQL update
main] o.s.jdbc.core.JdbcTemplate               : Executing prepared SQL statement [INSERT INTO employeeHealthInsur
main] o.s.j.d.DataSourceTransactionManager     : Initiating transaction commit
main] o.s.j.d.DataSourceTransactionManager     : Committing JDBC transaction on Connection [HikariProxyConnection@
main] o.s.j.d.DataSourceTransactionManager     : Releasing JDBC Connection [HikariProxyConnection@1134202713 wrapp
main] o.s.jdbc.datasource.DataSourceUtils      : Returning JDBC Connection to DataSource
```

### 11.2.3 Transaction Propagation (Not_Supports):



Here for the Organization Service we have defined the transaction propagation as **REQUIRED** and the Employee Service have the transaction propagation defined as **NOT_SUPPORTED**

**Code-**

The Organization Service will be as follows-

```
package com.javainuse.service.impl;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import com.javainuse.model.Employee;
import com.javainuse.model.EmployeeHealthInsurance;
import com.javainuse.service.EmployeeService;
import com.javainuse.service.HealthInsuranceService;
import com.javainuse.service.OrganizationService;
@Service
@Transactional
public class OrganzationServiceImpl implements OrganizationService {
@Autowired
EmployeeService employeeService;
@Autowired
HealthInsuranceService healthInsuranceService;
@Override
public void joinOrganization(Employee employee, EmployeeHealthInsurance
employeeHealthInsurance) {
employeeService.insertEmployee(employee);
if (employee.getEmpId().equals("emp1")) {
throw new RuntimeException("thowing exception to test transaction rollback");
}
healthInsuranceService.registerEmployeeHealthInsurance(employeeHealthInsura
nce);
}
@Override
public void leaveOrganization(Employee employee, EmployeeHealthInsurance
employeeHealthInsurance) {
employeeService.deleteEmployeeById(employee.getEmpId());
healthInsuranceService.deleteEmployeeHealthInsuranceById(employeeHealthIns
urance.getEmpId());
}
}
```

The Employee Service will be as follows-

```
package com.javainuse.service.impl;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import com.javainuse.dao.EmployeeDao;
import com.javainuse.model.Employee;
import com.javainuse.service.EmployeeService;
@Service
@Transactional(propagation=Propagation.NOT_SUPPORTED)
public class EmployeeServiceImpl implements EmployeeService {
@Autowired
EmployeeDao employeeDao;
@Override
public void insertEmployee(Employee employee) {
employeeDao.insertEmployee(employee);
}
@Override
public void deleteEmployeeById(String empid) {
employeeDao.deleteEmployeeById(empid);
}
}
```

**Output**

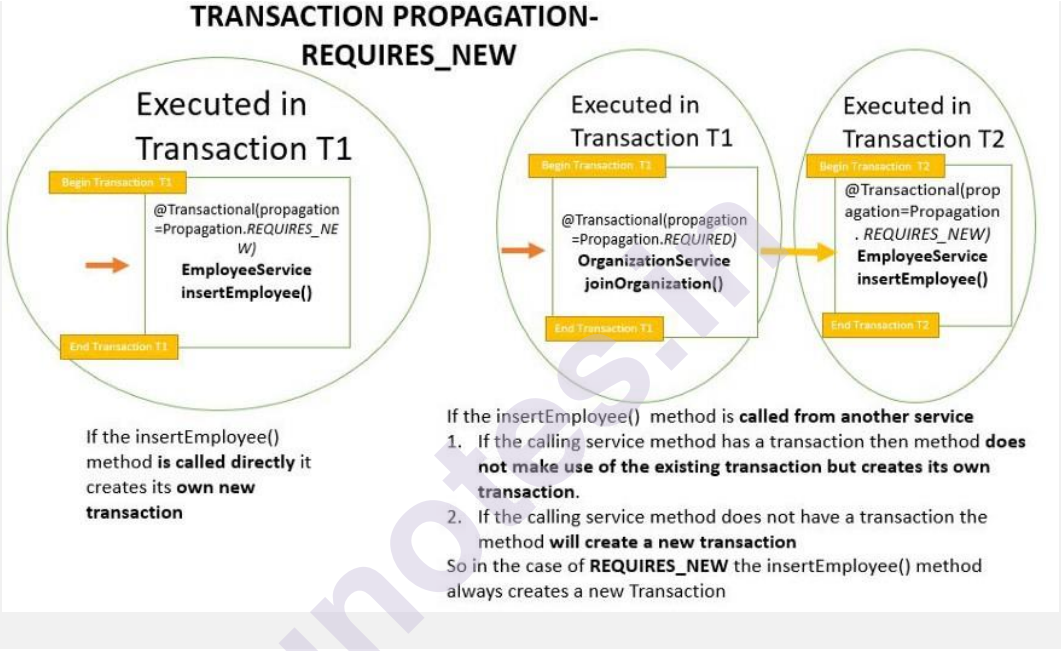
**EmployeeService called using OrganizationService -**

```
  main] com.javainuse.SpringBootJdbcApplication  : Started SpringBootJdbcApplication in 3.926 seconds (JVM running
  main] o.s.j.d.DataSourceTransactionManager     : Creating new transaction with name [com.javainuse.service.impl.
  main] o.s.j.d.DataSourceTransactionManager     : Acquired Connection [HikariProxyConnection@1134202713 wrapping
  main] o.s.j.d.DataSourceTransactionManager     : Switching JDBC Connection [HikariProxyConnection@1134202713 wra
  main] o.s.j.d.DataSourceTransactionManager     : Suspending current transaction
  main] o.s.jdbc.core.JdbcTemplate               : Executing prepared SQL update
  main] o.s.jdbc.core.JdbcTemplate               : Executing prepared SQL statement [INSERT INTO employee (empId,
  main] o.s.jdbc.datasource.DataSourceUtils      : Fetching JDBC Connection from DataSource
  main] o.s.jdbc.datasource.DataSourceUtils      : Registering transaction synchronization for JDBC Connection
  main] o.s.jdbc.datasource.DataSourceUtils      : Returning JDBC Connection to DataSource
  main] o.s.j.d.DataSourceTransactionManager     : Resuming suspended transaction after completion of inner transa
  main] o.s.j.d.DataSourceTransactionManager     : Participating in existing transaction
  main] o.s.jdbc.core.JdbcTemplate               : Executing prepared SQL update
  main] o.s.jdbc.core.JdbcTemplate               : Executing prepared SQL statement [INSERT INTO employeeHealthIns
  main] o.s.j.d.DataSourceTransactionManager     : Initiating transaction commit
  main] o.s.j.d.DataSourceTransactionManager     : Committing JDBC transaction on Connection [HikariProxyConnectio
  main] o.s.j.d.DataSourceTransactionManager     : Releasing JDBC Connection [HikariProxyConnection@1134202713 wra
  main] o.s.jdbc.datasource.DataSourceUtils      : Returning JDBC Connection to DataSource
read-2] s.c.a.AnnotationConfigApplicationContext : Closing org.springframework.context.annotation.AnnotationConfig
read-2] o.s.j.e.a.AnnotationMBeanExporter        : Unregistering JMX-exposed beans on shutdown
read-2] o.s.j.e.a.AnnotationMBeanExporter        : Unregistering JMX-exposed beans
read-2] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Shutdown initiated...
read-2] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Shutdown completed.
```

**EmployeeService called directly -**

```
    main] com.javainuse.SpringBootJdbcApplication  : Started SpringBootJdbcApplication in 4.146 seconds (JVM runni
    main] o.s.jdbc.core.JdbcTemplate               : Executing prepared SQL update
    main] o.s.jdbc.core.JdbcTemplate               : Executing prepared SQL statement [INSERT INTO employee (empId
    main] o.s.jdbc.datasource.DataSourceUtils      : Fetching JDBC Connection from DataSource
    main] o.s.jdbc.datasource.DataSourceUtils      : Registering transaction synchronization for JDBC Connection
    main] o.s.jdbc.datasource.DataSourceUtils      : Returning JDBC Connection to DataSource
Thread-2] s.c.a.AnnotationConfigApplicationContext : Closing org.springframework.context.annotation.AnnotationConf
Thread-2] o.s.j.e.a.AnnotationMBeanExporter        : Unregistering JMX-exposed beans on shutdown
Thread-2] o.s.j.e.a.AnnotationMBeanExporter        : Unregistering JMX-exposed beans
Thread-2] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Shutdown initiated...
Thread-2] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Shutdown completed.
```

### 11.2.4 Transaction Propagation (Requires_New):




Here for the Organization Service we have defined the transaction propagation as **REQUIRED** and the Employee Service have the transaction propagation defined as **REQUIRES_NEW**

**Code-**

The Organization Service will be as follows-

```
package com.javainuse.service.impl;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import com.javainuse.model.Employee;
import com.javainuse.model.EmployeeHealthInsurance;
import com.javainuse.service.EmployeeService;
import com.javainuse.service.HealthInsuranceService;
import com.javainuse.service.OrganizationService;
@Service
```

**@Transactional**

```
public class OrganzationServiceImpl implements OrganizationService {
@Autowired
EmployeeService employeeService;
@Autowired
HealthInsuranceService healthInsuranceService;
@Override
public void joinOrganization(Employee employee, EmployeeHealthInsurance
employeeHealthInsurance) {
employeeService.insertEmployee(employee);
if (employee.getEmpId().equals("emp1")) {
throw new RuntimeException("thowing exception to test transaction rollback");
}
healthInsuranceService.registerEmployeeHealthInsurance(employeeHealthInsuran
ce);
}
@Override
public void leaveOrganization(Employee employee, EmployeeHealthInsurance
employeeHealthInsurance) {
employeeService.deleteEmployeeById(employee.getEmpId());
healthInsuranceService.deleteEmployeeHealthInsuranceById(employeeHealthInsu
rance.getEmpId());
}
}
```

The Employee Service will be as follows-

```
package com.javainuse.service.impl;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import com.javainuse.dao.EmployeeDao;
import com.javainuse.model.Employee;
import com.javainuse.service.EmployeeService;
@Service
@Transactional(propagation=Propagation.REQUIRES_NEW)
public class EmployeeServiceImpl implements EmployeeService {
@Autowired
EmployeeDao employeeDao;
```

```
@Override
public void insertEmployee(Employee employee) {
employeeDao.insertEmployee(employee);
}
@Override
public void deleteEmployeeById(String empid) {
employeeDao.deleteEmployeeById(empid);
}
}
```

**Output:**

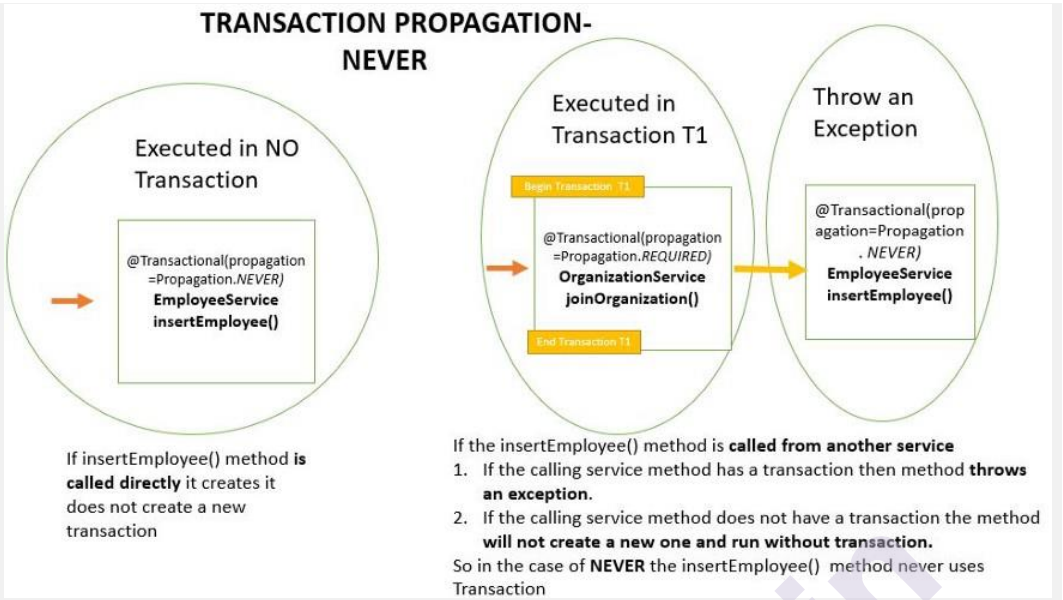
**EmployeeService called using OrganizationService -**

```
main] com.javainuse.SpringBootJdbcApplication  : Started SpringBootJdbcApplication in 4.647 seconds (JVM running
main] o.s.j.d.DataSourceTransactionManager     : Creating new transaction with name [com.javainuse.service.impl.
main] o.s.j.d.DataSourceTransactionManager     : Acquired Connection [HikariProxyConnection@1338958728 wrapping
main] o.s.j.d.DataSourceTransactionManager     : Switching JDBC Connection [HikariProxyConnection@1338958728 wra
main] o.s.j.d.DataSourceTransactionManager     : Suspending current transaction, creating new transaction with n
main] o.s.j.d.DataSourceTransactionManager     : Acquired Connection [HikariProxyConnection@1521568953 wrapping
main] o.s.j.d.DataSourceTransactionManager     : Switching JDBC Connection [HikariProxyConnection@1521568953 wra
main] o.s.jdbc.core.JdbcTemplate               : Executing prepared SQL update
main] o.s.jdbc.core.JdbcTemplate               : Executing prepared SQL statement [INSERT INTO employee (empId,
main] o.s.j.d.DataSourceTransactionManager     : Initiating transaction commit
main] o.s.j.d.DataSourceTransactionManager     : Committing JDBC transaction on Connection [HikariProxyConnectio
main] o.s.j.d.DataSourceTransactionManager     : Releasing JDBC Connection [HikariProxyConnection@1521568953 wra
main] o.s.jdbc.datasource.DataSourceUtils      : Returning JDBC Connection to DataSource
main] o.s.j.d.DataSourceTransactionManager     : Resuming suspended transaction after completion of inner transa
main] o.s.j.d.DataSourceTransactionManager     : Participating in existing transaction
main] o.s.jdbc.core.JdbcTemplate               : Executing prepared SQL update
main] o.s.jdbc.core.JdbcTemplate               : Executing prepared SQL statement [INSERT INTO employeeHealthIns
main] o.s.j.d.DataSourceTransactionManager     : Initiating transaction commit
main] o.s.j.d.DataSourceTransactionManager     : Committing JDBC transaction on Connection [HikariProxyConnectio
main] o.s.j.d.DataSourceTransactionManager     : Releasing JDBC Connection [HikariProxyConnection@1338958728 wra
main] o.s.jdbc.datasource.DataSourceUtils      : Returning JDBC Connection to DataSource
ead-2] s.c.a.AnnotationConfigApplicationContext : Closing org.springframework.context.annotation.AnnotationConfig
ead-2] o.s.j.e.a.AnnotationMBeanExporter       : Unregistering JMX-exposed beans on shutdown
ead-2] o.s.j.e.a.AnnotationMBeanExporter       : Unregistering JMX-exposed beans
ead-2] com.zaxxer.hikari.HikariDataSource      : HikariPool-1 - Shutdown initiated...
ead-2] com.zaxxer.hikari.HikariDataSource      : HikariPool-1 - Shutdown completed.
```

**EmployeeService called directly -**

```
main] com.javainuse.SpringBootJdbcApplication  : Started SpringBootJdbcApplication in 4.647 seconds (JVM running
main] o.s.j.d.DataSourceTransactionManager     : Creating new transaction with name [com.javainuse.service.impl.
main] o.s.j.d.DataSourceTransactionManager     : Acquired Connection [HikariProxyConnection@1338958728 wrapping
main] o.s.j.d.DataSourceTransactionManager     : Switching JDBC Connection [HikariProxyConnection@1338958728 wra
main] o.s.j.d.DataSourceTransactionManager     : Suspending current transaction, creating new transaction with n
main] o.s.j.d.DataSourceTransactionManager     : Acquired Connection [HikariProxyConnection@1521568953 wrapping
main] o.s.j.d.DataSourceTransactionManager     : Switching JDBC Connection [HikariProxyConnection@1521568953 wra
main] o.s.jdbc.core.JdbcTemplate               : Executing prepared SQL update
main] o.s.jdbc.core.JdbcTemplate               : Executing prepared SQL statement [INSERT INTO employee (empId,
main] o.s.j.d.DataSourceTransactionManager     : Initiating transaction commit
main] o.s.j.d.DataSourceTransactionManager     : Committing JDBC transaction on Connection [HikariProxyConnectio
main] o.s.j.d.DataSourceTransactionManager     : Releasing JDBC Connection [HikariProxyConnection@1521568953 wra
main] o.s.jdbc.datasource.DataSourceUtils      : Returning JDBC Connection to DataSource
main] o.s.j.d.DataSourceTransactionManager     : Resuming suspended transaction after completion of inner transa
main] o.s.j.d.DataSourceTransactionManager     : Participating in existing transaction
main] o.s.jdbc.core.JdbcTemplate               : Executing prepared SQL update
main] o.s.jdbc.core.JdbcTemplate               : Executing prepared SQL statement [INSERT INTO employeeHealthIns
main] o.s.j.d.DataSourceTransactionManager     : Initiating transaction commit
main] o.s.j.d.DataSourceTransactionManager     : Committing JDBC transaction on Connection [HikariProxyConnectio
main] o.s.j.d.DataSourceTransactionManager     : Releasing JDBC Connection [HikariProxyConnection@1338958728 wra
main] o.s.jdbc.datasource.DataSourceUtils      : Returning JDBC Connection to DataSource
ead-2] s.c.a.AnnotationConfigApplicationContext : Closing org.springframework.context.annotation.AnnotationConfig
ead-2] o.s.j.e.a.AnnotationMBeanExporter       : Unregistering JMX-exposed beans on shutdown
ead-2] o.s.j.e.a.AnnotationMBeanExporter       : Unregistering JMX-exposed beans
ead-2] com.zaxxer.hikari.HikariDataSource      : HikariPool-1 - Shutdown initiated...
ead-2] com.zaxxer.hikari.HikariDataSource      : HikariPool-1 - Shutdown completed.
```

### 11.2.5 Transaction Propagation (Never):




Here for the Organization Service we have defined the transaction propagation as **REQUIRED** and the Employee Service have the transaction propagation defined as **NEVER**s

**Code-**

The Organization Service will be as follows-

```
package com.javainuse.service.impl;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import com.javainuse.model.Employee;
import com.javainuse.model.EmployeeHealthInsurance;
import com.javainuse.service.EmployeeService;
import com.javainuse.service.HealthInsuranceService;
import com.javainuse.service.OrganizationService;
@Service
@Transactional
public class OrganzationServiceImpl implements OrganizationService {
@Autowired
EmployeeService employeeService;
@Autowired
HealthInsuranceService healthInsuranceService;
@Override
public void joinOrganization(Employee employee, EmployeeHealthInsurance
```

```
employeeHealthInsurance) {
employeeService.insertEmployee(employee);
if (employee.getEmpId().equals("emp1")) {
throw new RuntimeException("thowing exception to test transaction rollback");
}
healthInsuranceService.registerEmployeeHealthInsurance(employeeHealthInsuran
ce);
}
@Override
public void leaveOrganization(Employee employee, EmployeeHealthInsurance
employeeHealthInsurance) {
employeeService.deleteEmployeeById(employee.getEmpId());
healthInsuranceService.deleteEmployeeHealthInsuranceById(employeeHealthInsu
rance.getEmpId());
}
}
```

The Employee Service will be as follows-

```
package com.javainuse.service.impl;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import com.javainuse.dao.EmployeeDao;
import com.javainuse.model.Employee;
import com.javainuse.service.EmployeeService;
@Service
@Transactional(propagation=Propagation.NEVER)
public class EmployeeServiceImpl implements EmployeeService {

@Autowired
EmployeeDao employeeDao;
@Override
public void insertEmployee(Employee employee) {
employeeDao.insertEmployee(employee);
}
@Override
public void deleteEmployeeById(String empid) {
employeeDao.deleteEmployeeById(empid);
}
}
```

## Output:

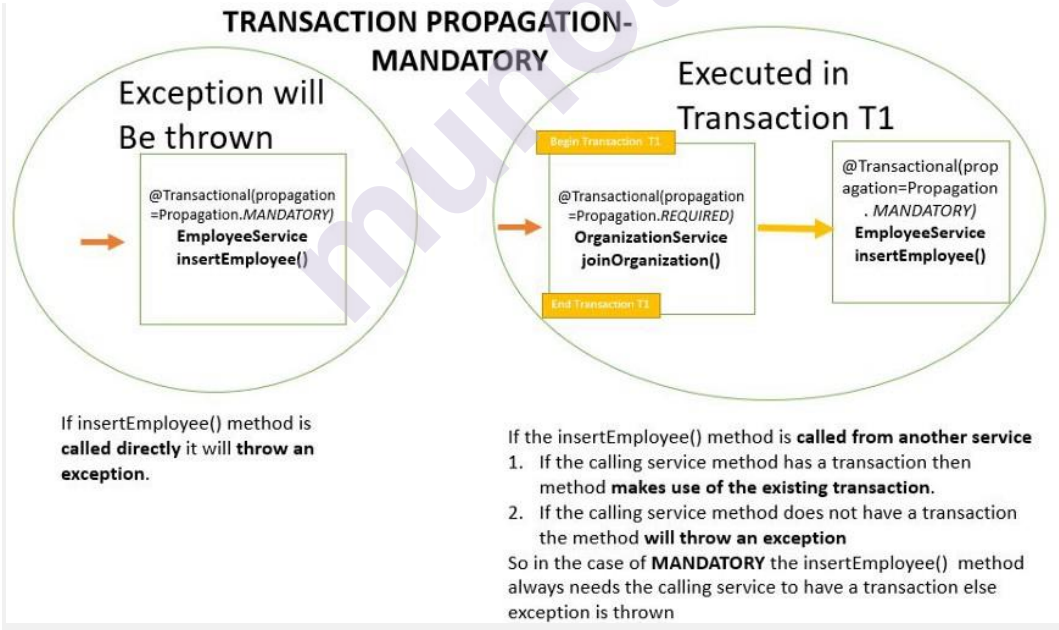
### EmployeeService called using OrganizationService -

```
    main] com.javainuse.SpringBootJdbcApplication  : Started SpringBootJdbcApplication in 4.409 seconds (JVM running
    main] o.s.jdbc.core.JdbcTemplate               : Executing prepared SQL update
    main] o.s.jdbc.core.JdbcTemplate               : Executing prepared SQL statement [INSERT INTO employee (empId,
    main] o.s.jdbc.datasource.DataSourceUtils      : Fetching JDBC Connection from DataSource
    main] o.s.jdbc.datasource.DataSourceUtils      : Registering transaction synchronization for JDBC Connection
    main] o.s.jdbc.datasource.DataSourceUtils      : Returning JDBC Connection to DataSource
Thread-2] s.c.a.AnnotationConfigApplicationContext : Closing org.springframework.context.annotation.AnnotationConfig
Thread-2] o.s.j.e.a.AnnotationMBeanExporter        : Unregistering JMX-exposed beans on shutdown
Thread-2] o.s.j.e.a.AnnotationMBeanExporter        : Unregistering JMX-exposed beans
Thread-2] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Shutdown initiated...
Thread-2] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Shutdown completed.
```

### EmployeeService called directly -

```
main] com.javainuse.SpringBootJdbcApplication  : Started SpringBootJdbcApplication in 3.907 seconds (JVM running for 5.268)
main] o.s.j.d.DataSourceTransactionManager     : Creating new transaction with name [com.javainuse.service.impl.OrganzationS
main] o.s.j.d.DataSourceTransactionManager     : Acquired Connection [HikariProxyConnection@727860268 wrapping com.mysql.cj.
main] o.s.j.d.DataSourceTransactionManager     : Switching JDBC Connection [HikariProxyConnection@727860268 wrapping com.mys
main] o.s.j.d.DataSourceTransactionManager     : Initiating transaction rollback
main] o.s.j.d.DataSourceTransactionManager     : Rolling back JDBC transaction on Connection [HikariProxyConnection@72786026
main] o.s.j.d.DataSourceTransactionManager     : Releasing JDBC Connection [HikariProxyConnection@727860268 wrapping com.mys
main] o.s.jdbc.datasource.DataSourceUtils      : Returning JDBC Connection to DataSource
action.IllegalTransactionStateException: Existing transaction found for transaction marked with propagation 'never'
bstractPlatformTransactionManager.handleExistingTransaction(AbstractPlatformTransactionManager.java:406)
bstractPlatformTransactionManager.getTransaction(AbstractPlatformTransactionManager.java:354)
or.TransactionAspectSupport.createTransactionIfNecessary(TransactionAspectSupport.java:474)
or.TransactionAspectSupport.invokeWithinTransaction(TransactionAspectSupport.java:289)
or.TransactionInterceptor.invoke(TransactionInterceptor.java:98)
iveMethodInvocation.proceed(ReflectiveMethodInvocation.java:186)
pProxy$DynamicAdvisedInterceptor.intercept(CglibAopProxy.java:688)
eImpl$$EnhancerBySpringCGLIB$$90f42036.insertEmployee(<generated>)
viceImpl.joinOrganization(OrganzationServiceImpl.java:25)
```

### 11.2.6 Transaction Propagation (Mandatory):




Here for the Organization Service we have defined the transaction propagation as **REQUIRED** and the Employee Service have the transaction propagation defined as **MANDATORY**

**Code-**

The Organization Service will be as follows-

```
package com.javainuse.service.impl;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import com.javainuse.model.Employee;
import com.javainuse.model.EmployeeHealthInsurance;
import com.javainuse.service.EmployeeService;
import com.javainuse.service.HealthInsuranceService;
import com.javainuse.service.OrganizationService;
@Service
@Transactional
public class OrganzationServiceImpl implements OrganizationService {
@Autowired
EmployeeService employeeService;
@Autowired
HealthInsuranceService healthInsuranceService;
@Override
public void joinOrganization(Employee employee, EmployeeHealthInsurance
employeeHealthInsurance) {
employeeService.insertEmployee(employee);
if (employee.getEmpId().equals("emp1")) {
throw new RuntimeException("thowing exception to test transaction rollback");
}
healthInsuranceService.registerEmployeeHealthInsurance(employeeHealthInsuran
ce);
}
@Override
public void leaveOrganization(Employee employee, EmployeeHealthInsurance
employeeHealthInsurance) {
employeeService.deleteEmployeeById(employee.getEmpId());
healthInsuranceService.deleteEmployeeHealthInsuranceById(employeeHealthInsu
rance.getEmpId());
}
}
```

The Employee Service will be as follows-

```
package com.javainuse.service.impl;
import org.springframework.beans.factory.annotation.Autowired;
```

```
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import com.javainuse.dao.EmployeeDao;
import com.javainuse.model.Employee;
import com.javainuse.service.EmployeeService;
@Service
@Transactional(propagation=Propagation.MANDATORY)
public class EmployeeServiceImpl implements EmployeeService {
@Autowired
EmployeeDao employeeDao;
@Override
public void insertEmployee(Employee employee) {
employeeDao.insertEmployee(employee);
}
@Override
public void deleteEmployeeById(String empid) {
employeeDao.deleteEmployeeById(empid);
}
}
```

**Output:**

**EmployeeService called using OrganizationService -**

```
   main] com.javainuse.SpringBootJdbcApplication  : Started SpringBootJdbcApplication in 3.83 seconds (JVM running f
   main] o.s.j.d.DataSourceTransactionManager     : Creating new transaction with name [com.javainuse.service.impl.C
   main] o.s.j.d.DataSourceTransactionManager     : Acquired Connection [HikariProxyConnection@1599674462 wrapping c
   main] o.s.j.d.DataSourceTransactionManager     : Switching JDBC Connection [HikariProxyConnection@1599674462 wrap
   main] o.s.j.d.DataSourceTransactionManager     : Participating in existing transaction
   main] o.s.jdbc.core.JdbcTemplate               : Executing prepared SQL update
   main] o.s.jdbc.core.JdbcTemplate               : Executing prepared SQL statement [INSERT INTO employee (empId, e
   main] o.s.j.d.DataSourceTransactionManager     : Participating in existing transaction
   main] o.s.jdbc.core.JdbcTemplate               : Executing prepared SQL update
   main] o.s.jdbc.core.JdbcTemplate               : Executing prepared SQL statement [INSERT INTO employeeHealthInsu
   main] o.s.j.d.DataSourceTransactionManager     : Initiating transaction commit
   main] o.s.j.d.DataSourceTransactionManager     : Committing JDBC transaction on Connection [HikariProxyConnectior
   main] o.s.j.d.DataSourceTransactionManager     : Releasing JDBC Connection [HikariProxyConnection@1599674462 wrap
   main] o.s.jdbc.datasource.DataSourceUtils      : Returning JDBC Connection to DataSource
hread-2] s.c.a.AnnotationConfigApplicationContext : Closing org.springframework.context.annotation.AnnotationConfigA
hread-2] o.s.j.e.a.AnnotationMBeanExporter        : Unregistering JMX-exposed beans on shutdown
hread-2] o.s.j.e.a.AnnotationMBeanExporter        : Unregistering JMX-exposed beans
hread-2] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Shutdown initiated...
hread-2] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Shutdown completed.
```

**EmployeeService called directly -**

```
524 --- [           main] com.javainuse.SpringBootJdbcApplication  : Started SpringBootJdbcApplication in 4.329 seconds (JVM running for 5.718)
springframework.transaction.IllegalTransactionStateException: No existing transaction found for transaction marked with propagation 'mandatory'
transaction.support.AbstractPlatformTransactionManager.getTransaction(AbstractPlatformTransactionManager.java:364)
transaction.interceptor.TransactionAspectSupport.createTransactionIfNecessary(TransactionAspectSupport.java:474)
transaction.interceptor.TransactionAspectSupport.invokeWithinTransaction(TransactionAspectSupport.java:289)
transaction.interceptor.TransactionInterceptor.invoke(TransactionInterceptor.java:98)
aop.framework.ReflectiveMethodInvocation.proceed(ReflectiveMethodInvocation.java:186)
aop.framework.CglibAopProxy$DynamicAdvisedInterceptor.intercept(CglibAopProxy.java:688)
e.impl.EmployeeServiceImpl$$EnhancerBySpringCGLIB$$789d1868.insertEmployee(<generated>)
BootJdbcApplication.main(SpringBootJdbcApplication.java:46)
524 --- [       Thread-2] s.c.a.AnnotationConfigApplicationContext : Closing org.springframework.context.annotation.AnnotationConfigApplicatior
524 --- [       Thread-2] o.s.j.e.a.AnnotationMBeanExporter        : Unregistering JMX-exposed beans on shutdown
524 --- [       Thread-2] o.s.j.e.a.AnnotationMBeanExporter        : Unregistering JMX-exposed beans
524 --- [       Thread-2] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Shutdown initiated...
524 --- [       Thread-2] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Shutdown completed.
```

## 11.3 Conclusion

In this chapter we looked at how to manage transactions in Spring stored programs, allowing us to group together related database changes, applying them all or aborting them all as a single logical unit. Implementing transactions using stored programs is a fairly natural choice, since a stored program can encapsulate complex transaction logic into a single database call, providing good separation between database and application logic.

## 11.4 List of References

https://docs.spring.io/spring-framework/docs/2.5.x/reference/transaction.htmls

https://medium.com/@rameez.s.shaikh/spring-boot-transaction-tutorial-understanding-transaction-propagation-ad553f5d85d4

❖❖❖❖❖