Topics :

**COLLECTIONS**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| [Collection](#Collection) | [Arraylist](#Arraylist) | [Linked List](#LinkedList) | [Hashmap](#Hashmap) | [Linked Hashmap](#linkedhashmap) | [Treemap](#Treemap) | [Hashset](#Hashset) | [Hashmap Vs Concurrent Hashmap](#HashmapVsConcHashmap) | [Linked Hashset](#Linkedhashset) | [Treeset](#Treeset) |

**SPRINGBOOT**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| [Restful webservice](#Restfulwebservice) | [Spring Annotations](#SpringAnnotations) | [Spring Profiles](#SpringProfiles) | [Spring Actuator](#SpringActuator) | [Version Control](#versioncontrol) | [Spring MVC Flow](#SpringMVCFlow) | [Spring Exception Handling](#SpringExeptionHandling) | [SOAP API Vs REST API](#SOAPVsREST) |  |  |
|  |  |  |  |  |  |  |  |  |  |

**SPRING ANNOTATIONS**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| [Autowired](#AT_AUTOWIRED) | [Bean](#AT_BEAN) | [component](#AT_COMPONENT) | [componentscan](#AT_COMPONENTSCAN) | [service](#AT_SERVICE) | [repository](#AT_REPOSITORY) | [springbootapplication](#AT_SPRINGBOOTAPP) |
|  |  |  |  |  |  |  |
| [configuration](#AT_CONFIGURATION) | [enableautoconfiguration](#AT_ENABLE_AUTO_CONFIG) | [restcontroller](#AT_REST_CONTROLLER) | [controller](#AT_CONTROLLER) | [lazy](#AT_LAZY) | [required](#AT_REQUIRED) | [primary](#AT_PRIMARY) |
|  |  |  |  |  |  |  |
| value |  |  |  |  |  |  |

**COMPARISON QUESTIONS**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| [Comparator/ Comparable](#comparatorcomparable) | [Association, Aggregation and Compostion](#AssoAggreCompo) | [Abstract / Interface](#AbstractInterface) | [Iterator / List Iterator](#IteratorVsListIterator) | [StringBuffer / StringBuilder](#StringBufferVsStringBuilder) | [FailFast](#Failfast) and [Failsafe](#Failsafe) | [Enummap Vs Enumset](#enummapVsEnumset) | [String Literal Vs String Object](#StringLiteralVsStringObject) | [Immutable Vs Mutable](#Mutable_Immutable) |  |
|  |  |  |  |  |  |  |  |  |  |

**JAVA 8 Concepts**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| [Functional Interface and Lambda Expresssion](#FunctionalInterfaceLambdaExp) | [Stream API](#StreamAPI) |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |

**Threading Concepts**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| [MultiTasking](#MultiTasking) | [MultiThreading](#MultiThreading) | [Thread Scheduler](#ThreadScheduler) |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |

**Micro service Concepts**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| [Microservices](#Microservices) | [Coupling and cohesion](#Coupling) | [Pros and Cons of Monolithic apps](#ProsConsofMonolithicapps) | [Benefits and Drawbacks of Micro](#BenefitsofMicro) | [Blast Radius and Resiliency](#BlastRadius) | [Circuit Breaker Pattern](#CircuitBreakerPattern) | [Ways of Building Microservice](#Waysofbuildingmicroservices) | [DB Event and Message](#DB_Event_Message) | [Distributed Transaction](#Distributed_Transaction) | [Two-phase commit Transactions (2pc)](#twoPC) |
| [Saga Pattern](#Saga_pattern) | [Saga Choreo pattern](#Saga_choreo_pattern) | [Saga Orchestration pattern](#Saga_orches_pattern) | [DB Monolith, SOA, Microservices](#DB_Monolith_SOA_Micro) | [Bounded Context](#Bounded_context) | [Hystrix](#Hystrix) |  |  |  |  |

**OTHER TOPICS**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| [Exception Handling](#ExceptionHandling) | [Design Patterns](#DesignPatterns) | [Final Keyword](#FinalKeyword) | [Encapsulation](#Encapsulation) | [Abstraction](#Abstraction) |  |
|  |  |  |  |  |  |



Diff between collection (interface) > list (i), queue (i) and set (i) and collections (class) -> utility class (some static methods are present)

Collection: - is an interface. Used to represent a group of individual objects into single unit

- List, set, queue are the main sub interfaces

- add(), remove(), clear(), size(), and contains() are the important methods of the Collection interface.

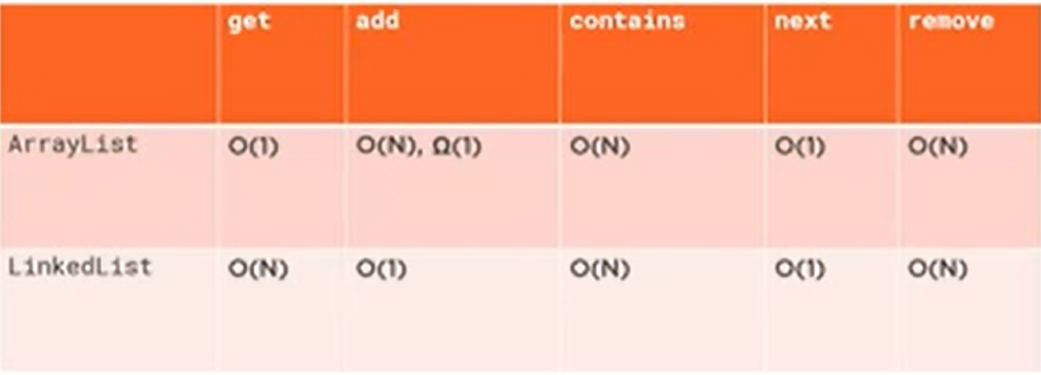
Collections: - is a utility class

- It defines several utility methods like sorting and searching which is used to operate on collection.

- It has all static methods.

- sort(), min(), max() are some example methods

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Collections (right) | Arraylist | LinkedList | Hashmap | Hashtable (not applicable to param) | Treemap | Hashset | Treeset |
| Parameters (down) |
| Duplicates | Allowed | A | Keys – NA  Values – A | NA - Contains unique elements | NA – Contains unique elements | NA | NA |
| Insertion Order | Preserved | P | NP (due to diff hashcode) | NP - The position of the bucket is identified by calling the hashcode() method. | **Maintains Ascending order** | NP | Maintaines Ascending order |
| Heterogeneous objects | Allowed | A | A for both key and values | Synchronized | Non synchronized | A | Non synchronized |
| Null insertion | Allowed | A | A (once for key) and A (any no of time for Value) | NA - Null insertion is not allowed both key and value | NA – for Key A – for values. | A | NA |
| Data Structure | Resizeable/growable array | Doubly linkedlist | Hashtable | - |  | Hashtable |  |
| Implements which interface | List,RandomAccess, cloneable and serializable interface | List, Deque,  Serializable and cloneable interface | Map, Serializable and cloneable interface | - | NavigableMap Serializable and cloneable interface | Serializable and cloneable interface | Sortedset |
| Best, when Freq operation is : | Retrieval | Insertion and Deletion in the middle | Search Operation | - |  | Search Operation | Access and retrieval |
| Default Capacity | 10  Lf: 0.75 |  | 16  Lf: 0.75 | 11  Lf: 0.75 |  | 16  Lf : 0.75 |  |





Arraylist: (implements list)

\_\_\_\_\_\_\_\_\_\_\_

1. Duplicates are allowed

2. Insertion order is preserved

3. Heterogeneous objects are allowed

4. Null insertion is possible

5. Underlying data struc : Resizeable Array / Growable Array

6. implements randomaccess interface - access random element with same speed

General exp:

- Arraylist default capacity is 10. it grows based on load factor and current capacity. The default value of load factor of an ArrayList is 0.75f

- Threshold = (Load Factor) \* (Current Capacity) = 0.75 \* 10 = 7 > This means after filling index 7, the array grows.

- it will internally create a new Arraylist and copy old elements into new Arraylist.

- In Java 8 and later, the new capacity of the ArrayList is calculated to be 50% more than its old capacity.

- new\_capacity = old\_capacity + (old\_capacity >> 1) = 10 + (10 >>1) = 10+5 = 15 (convert 10 to binary value and do right shift)

Linkedlist: (implements list)

\_\_\_\_\_\_\_\_\_\_\_\_

1. Duplicates are allowed

2. Insertion order is preserved

3. Heterogeneous objects are allowed

4. Null insertion is possible

5. underlying data struc : Double linked list

6. implements serializable and clonable interfaces but not randomaccess interface

7. it is a best choice if our frequent operation is insertion or deletion in the middle. And worst choice if it is retrieval operation.

General exp:

-we use linkedlist when we are not sure of the number of elements to be stored or dont have enough contagious memory for initializing the array.

-based on memory location stored in an object - it is of 2 types - singly linked list and doubly linked list

-In Linked List, we don't have to specify the size of the list as a linked list is a dynamic data structure and it automatically changes size when an element is added or removed. Also, the nodes of the linked list are not stored in a contiguous memory location, they are linked to each other with the help of next and previous pointers.

Hashmap: (implements map)

\_\_\_\_\_\_\_\_\_\_

1. Duplicate keys are not allowed but values can be duplicated

2. Insertion order is not preserved - based on hashcode of keys

3. Heterogeneous objects are allowed for both key and value

4. Null is allowed for key (once) and values (any no of times)

5. underlying data struc: Hashtable

6. implements serializable and cloneable interfaces but not randomaccess interface.

7. best choice if frequent operation is search operation.

\*\*note: Every method present in Hashmap is non synchronized (multiple threads are allowed to operate - relative performance high) whereas in hashtable it is synchronized (one thread is allowed to operate - relative performance is low)\*\*

General Exp:

-The initial default capacity of Java HashMap class is 16 with a load factor of 0.75.

Hashtable :

\_\_\_\_\_\_\_\_\_\_\_

1. A Hashtable is an array of a list. Each list is known as a bucket. The position of the bucket is identified by calling the hashcode() method. A Hashtable contains values based on the key.

2. Java Hashtable class contains unique elements.

3. Java Hashtable class doesn't allow null key or value.

4. Java Hashtable class is synchronized.

5. The initial default capacity of Hashtable class is 11 whereas loadFactor is 0.75.

linked hashmap : (implements map)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. child class of hashmap

2. underlying data struc : combination of linkedlist and hashtable

3. insertion order is preserved

4. commonly used in developing cache based applications

Treemap:

\_\_\_\_\_\_

1. Java TreeMap contains values based on the key. It implements the NavigableMap interface and extends AbstractMap class.

Java TreeMap contains only unique elements.

Java TreeMap cannot have a null key but can have multiple null values.

Java TreeMap is non synchronized.

Java TreeMap maintains ascending order.

Hashset: (implements set)

\_\_\_\_\_\_

1. Duplicates are not allowed

2. Insertion order is not preserved

3. Heterogeneous objects are allowed.

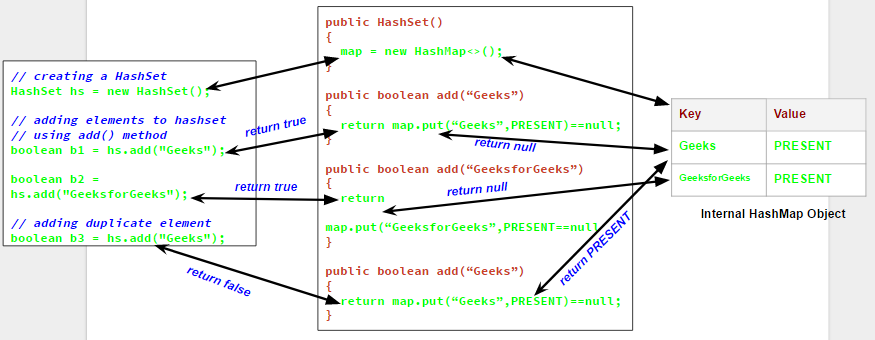
4. Null insertion is possible

5. underlying data struc : Hashtable

6. implements serializable and clonable interfaces but not randomaccess interface.

7. best choice if frequent operation is search operation.

How duplicates are not allowed :



Gen Exp :

-HashSet is an unordered Collection which extends the AbstractSet class and implements the Set interface.

-HashSet uses Hash Table for internal storage of its elements on the basis of their hashcodes.

-default capacity of HashSet is 16 and the load factor is 0.75.

-we can convert hashset to list using HashSet(Collection c) constructor.

linked hashset : (implements set)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. contains unique elements only like HashSet.

2. provides all optional set operations and permits null elements.

3. is non-synchronized.

4. maintains insertion order.

Treeset: (implements sortedset where sortedset implements set)

\_\_\_\_\_

1. contains unique elements only like HashSet.

2. access and retrieval times are quite fast.

3. doesn't allow null elements.

4. is non-synchronized.

5. maintains ascending order.

-------------------------------------------------------------------------------------------------------------------

Diff between hashmap and concurrent hashmap

* the HashMap is a non-synchronized and non-Thread safe, while the ConcurrentHashMap is a synchronized and Thread-safe collection class. Though the ConcurrentHashMap can not match the synchronization level of Hashtable, it performs well for most of the practical cases.
* The HashMap can be synchronized using the **Collection.syncronizedMap;** It returns a collection that is almost equal to Hashtable.
* The synchronized HashMap is less scalable than the ConcurrentHashMap.
* In the multi-threaded environment, The ConcurrentHashMap has improved performance than Synchronized HashMap.
* In the single-threaded environment, The HashMap is slightly better than ConcurrentHashMap.
* In HashMap, if one thread is iterating the object and the other thread wants to modify the objects, we will get a **ConcurrentModificationException** runtime exception. But, in ConcurrentHashMap, one thread can perform modification while the other thread is running.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

ConcurrentModification Exception : FailFast Case

[ConcurrentModificationException - Scaler](https://www.scaler.com/topics/concurrentmodificationexception-java/)

**ConcurrentModificationException :** The concurrent modification exception occurs in Java if we try to modify any object concurrently, although we do not have permission to modify it. Usually, when we are working with Java Collection classes, this type of exception might occur.

For instance, we have a collection list in our code, and some thread is iterating over it, whereas some other thread is trying to modify its value, during the process of iteration, then in this type of scenario we might get this ConcurrentModificationException. This is because the iteration outcome becomes undefined with it. This type of exception is thrown by some implementation of the Iterator class, which also includes all those general-purpose implementations of the Iterator given by the JRE. We call those Iterators **fail-fast** Iterators, which instantly throw an exception whenever they run into a problem rather than having to deal with the collection's unpredictable behavior in the future.

Fail-Safe iterators don’t throw any exceptions if a collection is structurally modified while iterating over it. This is because, they operate on the clone of the collection, not on the original collection and that’s why they are called fail-safe iterators. Iterator on CopyOnWriteArrayList, ConcurrentHashMap classes are examples of fail-safe Iterator.

------------------------------------------------------------------------------

Enumset and enummap

see active mq and mq related concepts

-----------------------------------------------------------------------------------------------------------------------------------------

Multi Tasking:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

- Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved in two ways:

Process-based Multitasking (Multiprocessing) and Thread-based Multitasking (Multithreading).

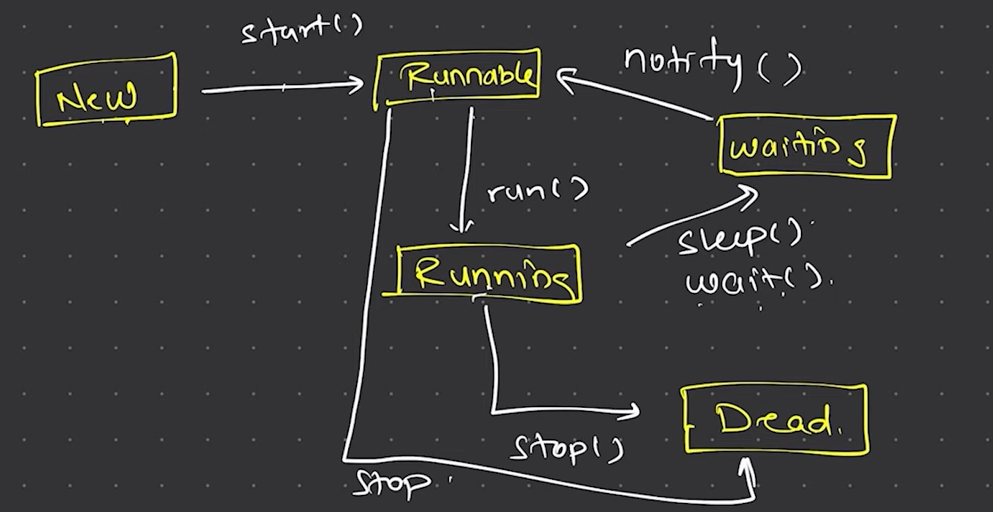
Multi threading:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

- process of executing multiple threads simultaneously.

- Java provides Thread class to achieve thread programming. Thread class provides constructors and methods to create and perform operations on a thread. Thread class extends Object class and implements Runnable interface.

- In Java, a thread always exists in any one of the following states. These states are: New, Active, Blocked / Waiting, Timed Waiting, Terminated



- Four ways of creating a thread

- By extending thread class

- By implementing Runnable interface.

- Using thread class: Thread(String Name)

- Using thread class: Thread(Runnable r, String name)

Start a Thread :

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Thread.start() method will newly create a thread. The new Thread starts > The thread moves to runnable state > when thread gets chance to execute it's target run method will run > it will go to running state

Constructors:

\_\_\_\_\_\_\_\_\_\_\_\_\_

Thread()

Thread(String name)

Thread(Runnable r)

Thread(Runnable r, String name)

Thread Scheduler:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

- A component of Java that decides which thread to run or execute and which thread to wait is called a thread scheduler in Java.

- A thread is only chosen by a thread scheduler if it is in the runnable state. However, if there is more than one thread in the runnable state, it is up to the thread scheduler to pick one of the threads and ignore the other ones. There are some criteria that decide which thread will execute first. There are two factors for scheduling a thread i.e. Priority (1(low) to 10(high)) and Time of arrival(this will be considered if priority is same)

- The thread scheduler selects the thread that has the highest priority, and the thread begins the execution of the job. If a thread is already in runnable state and another thread (that has higher priority) reaches in the runnable state, then the current thread is pre-empted from the processor, and the arrived thread with higher priority gets the CPU time.

- When two threads (Thread 2 and Thread 3) having the same priorities and arrival time, the scheduling will be decided based on FCFS algorithm. Thus, the thread that arrives first gets the opportunity to execute first.

ThreadPoolExecutor :

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

The ThreadPoolExecutor is an extensible thread pool implementation with lots of parameters and hooks for fine-tuning.

The main configuration parameters are corePoolSize, maximumPoolSize and keepAliveTime.

The pool consists of a fixed number of core threads that are kept inside all the time. It also consists of some excessive threads that may be spawned and then terminated when they are no longer needed.

The corePoolSize parameter is the number of core threads that will be instantiated and kept in the pool. When a new task comes in, if all core threads are busy and the internal queue is full, the pool is allowed to grow up to maximumPoolSize.

Sleep method (2 methods) :

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

public static void sleep(long mls) throws InterruptedException // This is native method

public static void sleep(long mls, int n) throws InterruptedException //non-native

mls - time in milliseconds

n - additional time for a thread tso sleep - 0 to 999999

Whenever the Thread.sleep() methods execute, it always halts the execution of the current thread.

Whenever another thread does interruption while the current thread is already in the sleep mode, then the InterruptedException is thrown.

Advantages of Java Multithreading :

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1) It doesn't block the user because threads are independent and you can perform multiple operations at the same time.

2) You can perform many operations together, so it saves time.

3) Threads are independent, so it doesn't affect other threads if an exception occurs in a single thread.

Some important points :

1. We call thread.start() method to invoke run() method. But, when we call the run() method directly, it will be called as a normal object call.

2. When the join() method is invoked, the current thread stops its execution and the thread goes into the wait state. The current thread remains in the wait state until the thread on which the join() method is invoked has achieved its dead state. If interruption of the thread occurs, then it throws the InterruptedException.

------------------------------------

Hystrix (microservices) is a library that controls the interaction between microservices to provide latency and fault tolerance. Additionally, it makes sense to modify the UI to let the user know that something might not have worked as expected or would take more time.

--------------------------------------------------------------------------------------------------------------------------------------------

Comparator and comparable - Comparable and Comparator both are interfaces and can be used to sort collection elements.

- Comparable is used to sort a single element in a list but in comparator we can sort multiple elements.

- Comparable provides (only compareTo() method) whereas comparator provides (compare() method and other methods)

- Comparable is in java.lang package but comparator is in java.util package

- Comparable affects the original class (Actual class is modified) but comparator does not affect the original class (Actual class is not modified).

- Comparable is a functional interface whereas comparator is not.

------------------------------------------------------------------------------------------------------------------------

Association, aggregation and composition in oops concepts

In Java, two types of Association are possible:

1. IS-A Association
2. HAS-A Association
3. Aggregation
4. Composition – part of

Aggregation - "has a" relationship

public class Employee{

Address address;

}

Association - relationship between objects - i.e one to one, many to many mapping

Composition - Child cannot exist independent of parent.

Aggregation vs Composition

1. Dependency: Aggregation implies a relationship where the child can exist independently of the parent. For example, Bank and Employee, delete the Bank and the Employee still exist. whereas Composition implies a relationship where the child cannot exist independent of the parent. Example: Human and heart, heart don’t exist separate to a Human

2. Type of Relationship: Aggregation relation is “has-a” and composition is “part-of” relation.

3. Type of association: Composition is a strong Association whereas Aggregation is a weak Association.

-----------------------------------------------------------------------------------------------------------

Exception handling - diff types of exception handling - compile time exception (checked exception) (sql exception, i/o exception) - runtime exception (unchecked exception) (nullpointer, arrayoutofbounds, arithmetic exception). we can create custom exceptions.

Exception handling - Throw/Throws and try/catch and finally block

Throw - used throw an exception explicitly in the code, inside the function or the block of code.

we can only propagate unchecked exception

The throw keyword is followed by an instance of Exception to be thrown.

We are allowed to throw only one exception at a time i.e. we cannot throw multiple exceptions.

Throws - used in the method signature to declare an exception which might be thrown by the function while the execution of the code.

we can declare both checked and unchecked exceptions. However, the throws keyword can be used to propagate checked exceptions only.

The throws keyword is followed by class names of Exceptions to be thrown.

We can declare multiple exceptions using throws keyword that can be thrown by the method.

Exception propagation (chain propagation) - refer more

-----------------------------------------------------------------------------------

Design patterns in java - creational pattern (Singleton pattern) , Structural pattern (Adapter pattern) , behavioral pattern (iterator pattern)

--------------------------------------------------------------------------------------

final keyword:

Final keyword is a non-access modifier where it can be used in class, method and variable.

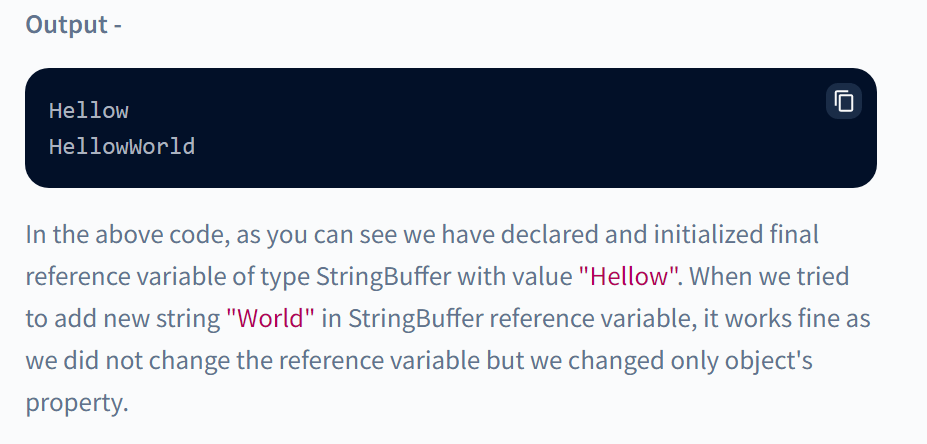
Final variable ---> we cannot change the value of final variable inside the same class or outside the class.

Final method ----> cannot override a final method but can be overloaded.

Final class ----> cannot extend but can be instantiated multiple times

Blank final variable ---> not initialized at time of declaration, but can be initialized inside constructor and as well as initialized once anywhere inside the same class. But cannot be reassigned.

Final reference variable ---> When we used final for any object reference, we can change the object properties.

Note : A field declared **static and final** is also called a "constant".

------------------------------------------------------------------------------------

Static Keyword :

To efficiently manage our program memory.

If we need to access the variables, methods defined inside the class without creating an instance of the class, we use static keyword.

**Static variable** – when we declare a static variable in a class, a single copy of that variable will be made available at class level and shared across all of its objects.

If any instance of a class modifies the value of a static variable, the change is reflected across all instances of the class.

It is a class-level variable, memory allocation of such variables only happens once when the class is loaded in the memory.

Static variables and methods are stored in heap memory.

**Static Block** – used to set the static member initial value

It is called before the main function.

**Static Method** – Any static member of the class can be accessed before any objects of that class are created with the use of class name.

Static method can view and edit the value of the static member variable.

We cannot access a non static data member or a non static method inside a static method. Instead we can call those non static variables/methods by creating an object of the class and calling them.

We cannot use this and super keyword inside a static method.

Static methods can be overloaded but cannot be overridden.

**Static inner class** - We can use static keyword in class level, only for inner class. Where we can access static variable of outer class inside the non static method of static inner class.

----------------------------------------------------

Abstract and interface (Add more content)

Abstract : 0 to 100% abstraction --- methods declared in abstract class needs to be implemented but additionally other methods also can be used.

-- Does not support multiple inheritance

--**Abstraction** is a process of hiding the implementation details and showing only functionality to the user.

* An abstract class must be declared with an abstract keyword.
* It can have abstract and non-abstract methods.
* It cannot be instantiated.
* It can have [constructors](https://www.javatpoint.com/java-constructor) and static methods also.
* It can have final methods which will force the subclass not to change the body of the method.

Scenarios:

1. Abstract class has Abstract method declaration. But implementation is given by extending abstract class.
2. Abstract methods cannot provide implementation in the same abstract class whereas non abstract methods can provide implementation in the same abstract class.
3. If we have multiple classes extending the abstract class, the run method will be run for which the object is instantiated.
4. If there is an abstract method in a class, that class must be abstract.
5. The abstract class can also be used to provide some implementation of the interface. In such case, the end user may not be forced to override all the methods of the interface.

Interface : 100% abstraction --- whatever methods declared in interface class those needs to be implemented compulsory -- Supports multiple inheritance

- every variable should be declared final in the interface class (internally added)

- The Java compiler adds public and abstract keywords before the interface method. Moreover, it adds public, static and final keywords before data members.

There are mainly three reasons to use interface. They are given below.

* It is used to achieve abstraction.
* By interface, we can support the functionality of multiple inheritance.
* It can be used to achieve loose coupling.

It cannot be instantiated just like the abstract class.

Since Java 8, we can have default and static methods in an interface.

Since Java 9, we can have private methods in an interface.

If a class implements multiple interfaces, or an interface extends multiple interfaces, it is known as multiple inheritance.

An interface which has no member is known as a marker or tagged interface, for example, [Serializable](https://www.javatpoint.com/serialization-in-java), Cloneable, Remote, etc. They are used to provide some essential information to the JVM so that JVM may perform some useful operation.

---------------------------------------------------------

Iterator interface Vs ListIterator interface

An Iterator is an interface in Java and we can traverse the elements of a list in a forward direction whereas a ListIterator is an interface that extends the Iterator interface and we can traverse the elements in both forward and backward directions. An Iterator can be used in these collection types like List, Set, and Queue whereas ListIterator can be used in List collection only. The important methods of Iterator interface are hasNext(), next() and remove() whereas important methods of ListIterator interface are add(), hasNext(), hasPrevious() and remove().

### **What are** **Mutable Objects**

The mutable objects are objects whose value can be changed after initialization. We can change the object's values, such as field and states, after the object is created. For example, **[Java.util.Date](https://www.javatpoint.com/java-util-date),**[**StringBuilder**](https://www.javatpoint.com/StringBuilder-class)**, [StringBuffer](https://www.javatpoint.com/StringBuffer-class)**, etc.

### **What are Immutable Objects**

The immutable objects are objects whose value can not be changed after initialization. We can not change anything once the object is created. For example, **primitive objects** such as [int](https://www.javatpoint.com/int-keyword-in-java), [long](https://www.javatpoint.com/long-keyword-in-java), [float](https://www.javatpoint.com/float-keyword-in-java), [double](https://www.javatpoint.com/double-keyword-in-java), **all**[**legacy classes**](https://www.javatpoint.com/legacy-class-in-java)**,**[**Wrapper class**](https://www.javatpoint.com/wrapper-class-in-java)**,**[**String class**](https://www.javatpoint.com/methods-of-string-class), etc.

-------------------------------------------------------------------

StringBuffer Vs StringBuilder

1. String is immutable whereas StringBuffer and StringBuilder are mutable classes.

2. The StringBuilder class was introduced as of Java 5 and the main difference between the StringBuffer and StringBuilder is that StringBuilder’s methods are not thread safe (not synchronized). StringBuffer is threadsafe (synchronized)

3. StringBuffer is less efficient, StringBuilder is more efficient.

(See string literals concept)

-----------------------------------------------------------

Encapsulation in Java is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit. In encapsulation, the variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class. Therefore, it is also known as data hiding.

To achieve encapsulation in Java :

1. Declare the variables of a class as private.

2. Provide public setter and getter methods to modify and view the variables values.

Benefits of Encapsulation :

1. The fields of a class can be made read-only or write-only.

2. A class can have total control over what is stored in its fields.

Polymorphism :

Poly means many and morphism means forms. So Many Forms.

Two types of polymorphisms

* Compile Time polymorphisms (Static polymorphism) -🡪 method overloading
* Run time polymorphisms (dynamic polymorphism) 🡪 method overriding

1. EnumMap: EnumMap is a specialized implementation of the Map interface for enumeration types. It implements the Map interface and extends AbstractMap in Java.

EnumMap is much faster than HashMap.

EnumMap class is a member of the Java Collections Framework.

EnumMap is an ordered collection maintained in the natural order of their keys.

All keys of each EnumMap instance must be keys of the same enum type.

EnumMap doesn’t allow inserting a null key if we try to insert the null key, it will throw NullPointerException.

EnumMap internally represented as arrays for better performance.

2. EnumSet: EnumSet is a specialized implementation of the Set interface for enumeration types. It implements the Set interface and extends AbstractSet in Java.

EnumSet class is a member of the Java Collections Framework and it is not synchronized.

All the elements in an EnumSet must come from a single enumeration type that is specified when the set is created either explicitly or implicitly.

EnumSet is much faster than HashSet.

EnumSet doesn’t allow inserting null object if we try to insert the null object, it will throw NullPointerException.

String Literal Vs String Object

String str = “Dhanush”;

This is string literal. When you declare string like this, you are actually calling intern() method on String. This method **references internal pool** of string objects. If there already exists a string value “Dhanush”, then str will reference of that string and no new String object will be created.

String str = new String(“Dhanush”);

This is string object. In this method JVM is forced to create a new string reference, even if “GeeksForGeeks” is in the reference pool.

Therefore, if we compare performance of string literal and string object, string object will always take more time to execute than string literal because it will construct a new string every time it is executed.

**Design Patterns**

A design pattern is a well-described solution to a common software problem.

Some of the benefits of using design patterns are:

* Already defined and provide an industry-standard approach to solving a recurring problem, so it saves time if we sensibly use the design pattern.
* Reusability that leads to more robust and highly maintainable code. It helps in reducing the total cost of ownership (TCO) of the software product.
* Easy to understand and debug. It leads to faster development and new members of the team understand it easily.

Java design patterns are divided into three categories - creational, structural, and behavioral design patterns.

In creational, we will discuss three important patterns. 1. Singleton design pattern. 2. Factory design pattern 3. Abstract Factory design pattern 4. Prototype pattern.

**Singleton design pattern:**

* Singleton pattern restricts the instantiation of a class and ensures that only one instance of the class exists in the Java Virtual Machine.
* The singleton class must provide a global access point to get the instance of the class.
* Singleton pattern is used for logging, drivers objects, caching, and thread pool.

To implement a singleton pattern, we have different approaches, but all of them have the following common concepts.

* Private constructor to restrict instantiation of the class from other classes.
* Private static variable of the same class that is the only instance of the class.
* Public static method that returns the instance of the class, this is the global access point for the outer world to get the instance of the singleton class.

There are different approaches to singleton pattern.

|  |  |
| --- | --- |
| Eager initialization  (private Constructor, private static final instance variable initialized, global access point returns instance) |  |
| Static Block Initialization  (private Constructor,  private static final instance variable not initialized,  static block initializes instance variable with try catch,  global access point returns instance) |  |
| Lazy initialization  (private Constructor,  private static final instance variable not initialized,  global access point has if condition for instance null or not. If YES, initialize the instance variable or else return the null instance) |  |
| Thread safe singleton  (private Constructor,  private static final instance variable not initialized,  global access point (with synchronized) has if condition for instance null or not. If YES, initialize the instance variable or else return the null instance) – this will create performance issue while multiple threads enter.  So, using double lock checking by using synchronized block inside if condition and initializing instance variable. |  |

**Prototype Design Pattern:**

Prototype pattern is used where object creation is a costly process and we have a similar object already. Prototype pattern uses cloning to copy original object to new object and modify according to our needs. Here we wont create a new object, but clone the object.

Here, class employees implements cloneable and has a clone method overridden which returns a copy of the object. In PrototypePatternTest class, we call clone method and cast to employees class and get the same copy of instance.

A screenshot of a computer program

Description automatically generatedA screenshot of a computer program

Description automatically generated

A blue background with white text

Description automatically generated

**Factory Design Pattern:**

If we have a superclass and multiple sub classes and based on input, we need to return one of the sub-class. In this scenario, we can use factory design pattern.

Computer abstract class – it has 3 abstract methods getRam, getHDD and getCPU

PC extends Computer - it has all arg constructor, getter and setter.

Server extends Computer – it has all arg constructor, getter and setter.

ComputerFactory class – which has getComputer Method

A computer code on a blue background

Description automatically generated

We can keep ComputerFactory class as singleton or we can keep the getComputer (factory method) as static.

Based on input parameters, different subclass is created and returned.

TestFactory class – Simple test client program which uses above factory design pattern implementation.

A computer code on a blue background

Description automatically generated

Output :

A close-up of a number

Description automatically generated

Advantages:

* Provides approach to code for interface rather than implementation.
* Removes instantiation of the actual implementation classes from client code.
* Abstraction between implementation and client classes through inheritance.

**Abstract Factory Design pattern:**

Here we will have Factory class for each subclass and a AbstractFactory interface which is implemented by Subclass factories.

The ComputerFactory class has getComputer (factory method) which calls the abstractfactory method inside AbstractFactory class.

While calling, we call the getComputer method from ComputerFactory class which inturn calls the AbstractFactory interface implementation classes (PCFactory and ServerFactory). So it is called Factory of Factories.

<https://www.digitalocean.com/community/tutorials/abstract-factory-design-pattern-in-java>

Builder Design Pattern:

Three major issues from factory and abstract factory design patterns are solved here.

* Too many arguments to be passed from client program to factory class and it is hard to maintain the order of the argument.
* Optional parameters are to be sent as null forcefully.
* Complexity of object creation and factory classes.

Implementation is as follows :

* First of all you need to create a static nested class and then copy all the arguments from the outer class to the Builder class. We should follow the naming convention and if the class name is Computer then builder class should be named as ComputerBuilder.
* Java Builder class should have a public constructor with all the required attributes as parameters.
* Java Builder class should have methods to set the optional parameters and it should return the same Builder object after setting the optional attribute.
* The final step is to provide a build() method in the builder class that will return the Object needed by client program. For this we need to have a private constructor in the Class with Builder class as argument.

<https://www.digitalocean.com/community/tutorials/builder-design-pattern-in-java>

Builder pattern is used in StringBuffer and StringBuilder

SOAP Vs Rest :

## REST: representational state transfer

REST is a set of architectural principles attuned to the needs of lightweight [web services](https://www.redhat.com/en/topics/cloud-computing/what-are-cloud-services) and [mobile applications](https://www.redhat.com/en/topics/mobile). Because it's a set of guidelines, it leaves the implementation of these recommendations to developers.

When a request for data is sent to a REST API, it’s usually done through hypertext transfer protocol (commonly referred to as HTTP). Once a request is received, APIs designed for REST (called RESTful APIs or RESTful web services) can return messages in a variety of formats: HTML, XML, plain text, and JSON. JSON (JavaScript object notation) is favored as a message format because it can be read by any programming language (despite the name), is human- and machine-readable, and is lightweight. In this way, RESTful APIs are more flexible and can be easier to set up.

An application is said to be RESTful if it follows 6 architectural guidelines. A RESTful application must have:

1. A client-server architecture composed of clients, servers, and resources.
2. [Stateless](https://www.redhat.com/en/topics/cloud-native-apps/stateful-vs-stateless) client-server communication, meaning no client content is stored on the server between requests. Information about the session’s state is instead held with the client.
3. Cacheable data to eliminate the need for some client-server interactions.
4. A uniform interface between components so that information is transferred in a standardized form instead of specific to an application’s needs. This is [described by Roy Fielding](https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm), the originator of REST, as “the central feature that distinguishes the REST architectural style from other network-based styles.”
5. A layered system constraint, where client-server interactions can be mediated by hierarchical layers.
6. Code on demand, allowing servers to extend the functionality of a client by transferring executable code (though also reducing visibility, making this an optional guideline).

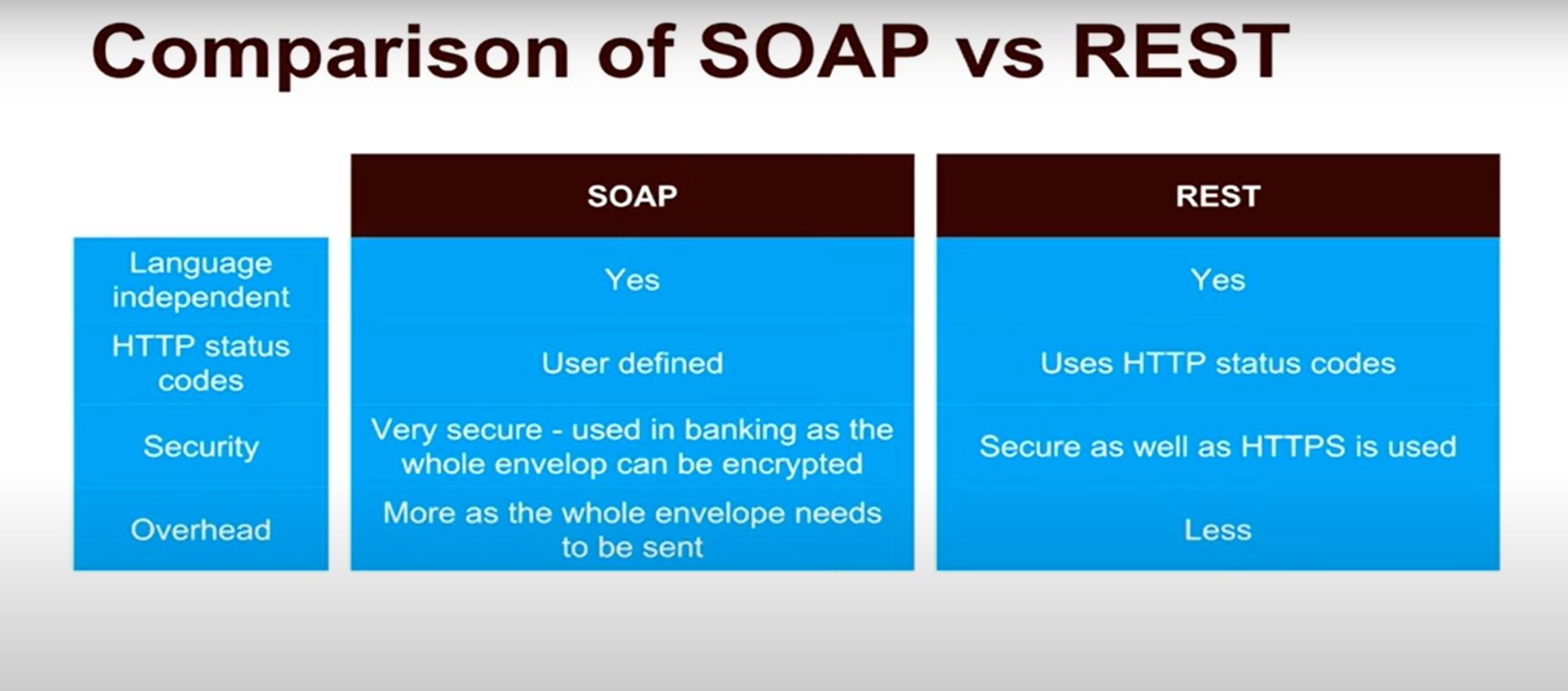
## SOAP: simple object access protocol

SOAP is a standard protocol that was first designed so that applications built with different languages and on different platforms could communicate. Because it is a protocol, it imposes built-in rules that increase its complexity and overhead, which can lead to longer page load times. However, these standards also offer built-in compliances that can make it preferable for enterprise scenarios. The built-in compliance standards include [security](https://www.redhat.com/en/topics/security/api-security), atomicity, consistency, isolation, and durability (ACID), which is a set of properties for ensuring reliable database transactions.

Common web service specifications include:

* **Web services security (WS-security)**: Standardizes how messages are secured and transferred through unique identifiers called tokens.
* **WS-ReliableMessaging**: Standardizes error handling between messages transferred across unreliable IT infrastructure.
* **Web services addressing (WS-addressing)**: Packages routing information as metadata within SOAP headers, instead of maintaining such information deeper within the network.
* **Web services description language (WSDL)**: Describes what a web service does, and where that service begins and ends.

When a request for data is sent to a SOAP API, it can be handled through any of the application layer protocols: HTTP (for web browsers), SMTP (for email), TCP, and others. However, once a request is received, return SOAP messages must be returned as XML documents—a markup language that is both human- and machine-readable. A completed request to a SOAP API is not cacheable by a browser, so it cannot be accessed later without resending to the API.





-------------------------------------------------------------------------------------------------------------------------------

JAVA 8

\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Functional Interfaces and Lambda Expressions :

- If you notice the above interface code, you will notice @FunctionalInterface annotation.

- Functional interfaces are a new concept introduced in Java 8.

- An interface with exactly one abstract method becomes a Functional Interface.

- Since functional interfaces have only one method, lambda expressions can easily provide the method implementation.

We just need to provide method arguments and business logic. For example, we can write above implementation using lambda expression as:

Example 1 :

Runnable r1 = () -> {

System.out.println("My Runnable");

};

Example 2 :

Interface1 i1 = (s) -> System.out.println(s);

i1.method1("abc");

Stream API's :

A new java.util.stream has been added in Java 8 to perform filter/map/reduce like operations with the collection.

Stream API will allow sequential as well as parallel execution.

In Collections and usually with Big Data, we use streams for filtering out them based on some conditions.

Collection interface has been extended with stream() and parallelStream() default methods to get the Stream for sequential and parallel execution.

package com.journaldev.java8.stream;

import java.util.ArrayList;

import java.util.List;

import java.util.stream.Stream;

public class StreamExample {

public static void main(String[] args) {

List<Integer> myList = new ArrayList<>();

for(int i=0; i<100; i++) myList.add(i);

//sequential stream

Stream<Integer> sequentialStream = myList.stream();

//parallel stream

Stream<Integer> parallelStream = myList.parallelStream();

//using lambda with Stream API, filter example

Stream<Integer> highNums = parallelStream.filter(p -> p > 90);

//using lambda in forEach

highNums.forEach(p -> System.out.println("High Nums parallel="+p));

Stream<Integer> highNumsSeq = sequentialStream.filter(p -> p > 90);

highNumsSeq.forEach(p -> System.out.println("High Nums sequential="+p));

}

}

Refer : https://www.digitalocean.com/community/tutorials/java-8-stream

--------------------------------------------------------------------

Restful webservice with springboot :

1. REST stands for REpresentational State Transfer.

2. REST is an architectural approach, not a protocol.

3. 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) (eg: /hello which we give inside @requestmaapping)

4. 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. If we have 1,2,3 fields and we wanted to change field number 2, we need to send all 3 fields data. Or else , the entire record would be changed.

POST: It creates a new resource.

DELETE: It deletes the resource.

PATCH: It also updates existing resource. But here, we don’t have to send all 3 fields data. Instead we can send field number 2 (which to be changed). I.E Fields that need to be updated by the client, only that field is updated without modifying the other field.

--------------------------------------------------------------------------

Spring Annotations :

@Required - it applies to bean class setter methods. - it will throw bean intialization exception, if we declare a setter method as @required and not setting its value.

--------------------

@Controller - The @Controller annotation is used to indicate the class is a Spring controller. This annotation can be used to identify controllers for Spring MVC or Spring WebFlux.

@RestController - simplify the creation of RESTful web services. It's a convenient annotation that combines @Controller and @ResponseBody, which eliminates the need to annotate every request handling method of the controller class with the @ResponseBody annotation.

@Service - This annotation is used on a class. @Service marks a Java class that performs some service, such as executing business logic, performing calculations, and calling external APIs. This annotation is a specialized form of the @Component annotation intended to be used in the service layer.

@Repository - This annotation is used on Java classes that directly access the database. The @Repository annotation works as a marker for any class that fulfills the role of repository or Data Access Object.

*Normally when we interact with DB, the exceptions occur are checked exceptions which should be handled in java files. We use @repository and use springDataAccess class to make it unchecked so that we don’t need to handle them.*

@component (Stereotype) - Marks a java class as a bean or component so that we can create instance of that class whenever needed using the (\*below code - ApplicationContext)\*. The below code creates an instance of the Gamerunner class which is marked @component on the class.

@Bean - Marks methods in java class. similar to @component, it also creates instances of bean classes but also return objects to IOC container.

- It is typically used with @configuration annotation in class level.

Note : @Component is a class-level annotation, but @Bean is at the method level, so @Component is only an option when a class's source code is editable. @Bean can always be used, but it's more verbose. @Component is compatible with Spring's auto-detection, but @Bean requires manual class instantiation.

ConfigurableApplicationContext context = SpringApplication.run(LearnSpringFrameworkApplication.class, args);

GameRunner runner = context.getBean(bean.class)

----------------------

@Autowired - Autowiring is injecting object dependency implicitly.

- Process of wiring dependencies for a spring Framework

- Three types - By property, by setter method, by constructor

- Constructor based was the best one - Reason is: when we have multiple dependencies, with one constructor based autowiring, objects are ready to be used.

- Visit : http://javainsimpleway.com/autowiring-in-spring/

Note : if there are more number of bean instances / no bean instances - spring will throw unsatisfiedDependencyException

@Primary - we will use this annotation whenever we have multiple bean classes for a autowiring (interface). so one of the bean classes can be annotated as @primary to be autowired.

Note: if we mark two or more classes as @primary, NoUniqueBeanDefinitionException will be thrown.

--------------------------

@SpringBootApplication - This annotation is used on the application class while setting up a Spring Boot project. The class that is annotated with the @SpringBootApplication must be kept in the base package. The one thing that the @SpringBootApplication does is a component scan. But it will scan only its sub-packages. As an example, if you put the class annotated with @SpringBootApplication in the com.example, then @SpringBootApplication will scan all its sub-packages, such as com.example.a, com.example.b, and com.example.a.x.

- It is a combination of @Configuration, @EnableAutoConfiguration, and @ComponentScan annotations

@Configuration - A Java class annotated with @Configuration is a configuration by itself and will have methods to instantiate and configure the dependencies.

@EnableAutoConfiguration - This annotation is usually placed on the main application class. The @EnableAutoConfiguration annotation implicitly defines a base “search package”. This annotation tells Spring Boot to start adding beans based on classpath settings, other beans, and various property settings.

***@ComponentScan - This annotation is used with the @Configuration annotation to allow Spring to know the packages to scan for annotated components.***

***@EnableAutoConfiguration: enable Spring Boot’s auto-configuration mechanism***

***@ComponentScan: enable @Component scan on the package where the application is located (see the best practices)***

***@Configuration: allow to register extra beans in the context or import additional configuration classes***

Note : if a class(marked @component) cannot be found in the current package (where the springboot application run class) where we are trying to get a class (with context.getBean(java.class)), then NosuchBeanDefinitionException will be thrown

@Lazy - used for initializing a bean lately - bean is created and intialized only after when it is first requested - we can also use this annotation on @configuration which means that methods which marked as @Bean will be lazily initialized.

@Scope – To specify a bean’s lifecycle. There are different scopes for a bean. They are Singleton, Prototype, request, Session

Default scope is Singleton. Which means only one instance of the bean is created for the entire application.



A white background with black text

Description automatically generated

In Singleton, Even if we create two objects for the class annotated as @scope, both will refer to same bean instance.

@Transactional – This annotation helps in rollback of unneeded persistence in DB. For example, we have a user and address classes and we wanted to save both user and address only if both are assigned some data and don’t store anything, if anyone of them is not assigned data.

For that we use @EnableTransactionManagement annotation in main class of springboot application and use @Transactional annotation above the method required.

A computer screen shot of a program

Description automatically generated A computer screen shot of a program code

Description automatically generated

---------------------------

TO BE CONTINUED TOMORROW.

------------------------------------------------------------------------------------

Spring MVC Flow :

1. Request goes to Dispatcher servlet (front controller)
2. Dispatcher Servlet consults Handler Mapping to call appropriate controller
3. Controller takes the request and calls appropriate service methods (based on GET or POST method)
4. The service method will set model data based on the defined business logic and returns view name to Dispatcher servlet
5. Dispatcher servlet will communicate with ViewResolver to pickup the defined view for the request.
6. Once view is finalized, The Dispatcher servlet passes the model data to the view to finally show on screen.

Dependency Injection - Identify beans (whosever has @component annotation), their dependency and wire them together (Inversion of control - we dont create instances, instead spring creates objects for us)

Spring beans - An object managed by the spring Framework.

IOC Container - When we launch application context, we typically launch an IOC Container. Which manages the lifecycle of bean and dependcies

- Application context is the implementation of IOC container.

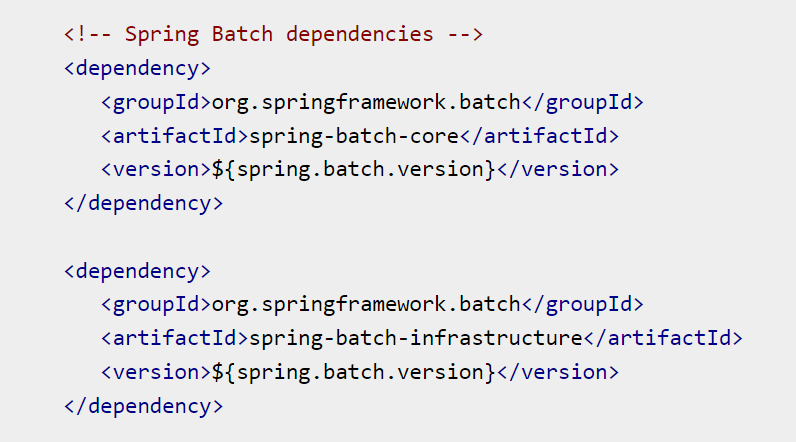
- 2 types - Application context (Advanced) and BeanFactory (Basic) (Rare used)

Flow : Controller > BusinessService(sum) > DataService (Data)

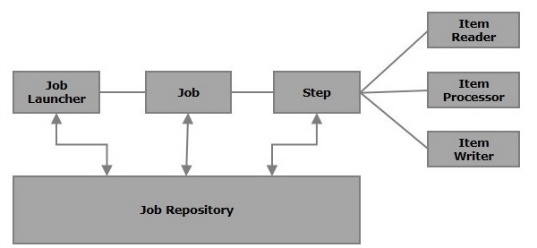
version control: In spring mvc, when we use different versions of dependency jars, we face configuration problems. But in Springboot, the version control is done through parent class version. By mentioning parent class version, we don't need to mention any version for dependency jar in maven. Spring will take care of downloading the jars based on parent version which are pre defined versions.

Spring Batch :

Spring batch is a lightweight framework which is used to develop Batch Applications that are used in Enterprise Applications.



Components of spring Batch :



Job - This tag is used to define/configure the job of the SpringBatch. It contains a set of steps and it can be launched using the JobLauncher. This tag has 2 attributes - Id and restartable

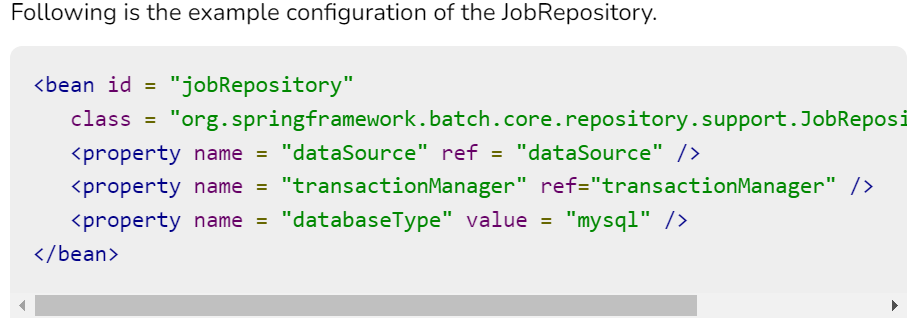
Step - This tag is used to define/configure the steps of a SpringBatch job. It has three attributes – Id, next and parent



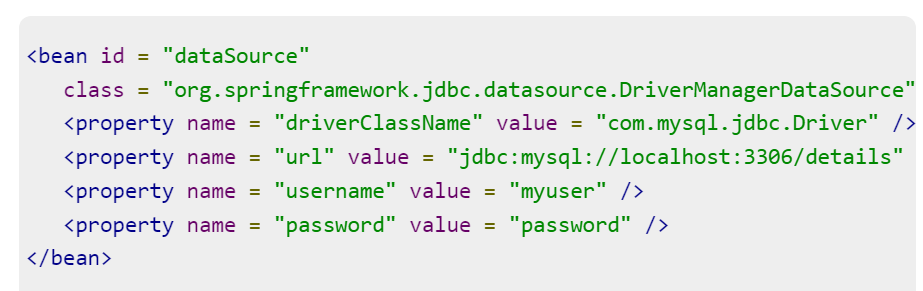
This tag is used to define/configure a chunk of a tasklet. It has four attributes – reader, writer, processor, commit-interval



JobRepository - The JobRepository Bean is used to configure the JobRepository using a relational database. This bean is associated with the class of type org.springframework.batch.core.repository.JobRepository. it has three attributes – dataSource, transactionManager, databaseType

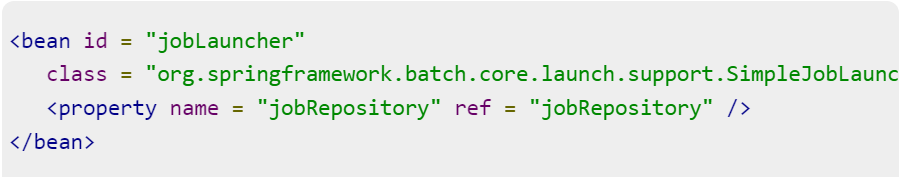


DataSource - he datasource bean is used to configure the Datasource. This bean is associated with the class of type org.springframework.jdbc.datasource.DriverManagerDataSource. it has four attributes – driverClassName, url, username, password



TransactionManager - The TransactionManager bean is used to configure the TransactionManager using a relational database. This bean is associated with the class of type org.springframework.transaction.platform.TransactionManager.

JobLauncher - The JobLauncher bean is used to configure the JobLauncher. It is associated with the class org.springframework.batch.core.launch.support.SimpleJobLauncher (in our programs). This bean has one property named jobrepository, and it is used to specify the name of the bean which defines the jobrepository.



An Item Reader reads data into the spring batch application from a particular source, whereas an Item Writer writes data from Spring Batch application to a particular destination.

Spring Batch provides an Interface ItemReader and ItemWriter. Both implement these interfaces respectively.

An Item processor is a class which contains the processing code which processes the data read in to the spring batch. If the application reads n records the code in the processor will be executed on each record. The interface ItemProcessor<I,O> represents the processor.

A chunk is a child element of the tasklet. It is used to perform read, write, and processing operations. We can configure reader, writer, and processors using this element, within a step as shown below.

-------------------------------------------------------------------------------------

Spring AOP – Aspect Oriented Programming – See only Tutorials point to get the idea.

[Tutorialspoint link for Spring AOP](https://www.tutorialspoint.com/springaop/springaop_application1.htm)

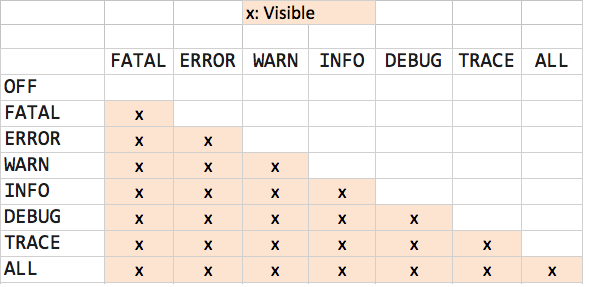
---------------------------------------------------------------------------------------

Manage App Config using Spring Profiles - For different environments we can configure different application.properties

application.properties > spring.profiles.active =dev

application-dev.properties > logging.level = trace

Different Logging Levels:



-----------------------------------------------------------------------------------

Monitor Apps using Spring Actuator - It provides a number of end points -> beans, health, metrics, mappings

Configure in application.properties as > management.endpoints.web.exposure = \*

Springboot - Exception handling

@ResponseStatus - allows us to modify the HTTP status of our response. It can be applied in the following places:

On the exception class itself

Along with the @ExceptionHandler annotation on methods

Along with the @ControllerAdvice annotation on classes

@ExceptionHandler - The @ExceptionHandler annotation gives us a lot of flexibility in terms of handling exceptions.

For starters, to use it, we simply need to create a method either in the controller itself or in a @ControllerAdvice class

and annotate it with @ExceptionHandler

@ExceptionHandler(NoSuchElementFoundException.class)

@ResponseStatus(HttpStatus.NOT\_FOUND)

public ResponseEntity<String> handleNoSuchElementFoundException(

NoSuchElementFoundException exception

) {

return ResponseEntity

.status(HttpStatus.NOT\_FOUND)

.body(exception.getMessage());

}

@ControllerAdvice - Controller advice classes allow us to apply exception handlers to more than one or all controllers in our application

If we want to selectively apply or limit the scope of the controller advice to a particular controller, or a package, we can use the properties provided by the annotation:

@ControllerAdvice("com.reflectoring.controller"): we can pass a package name or list of package names in the annotation’s value or basePackages parameter. With this, the controller advice will only handle exceptions of this package’s controllers.

@ControllerAdvice(annotations = Advised.class): only controllers marked with the @Advised annotation will be handled by the controller advice.

JPA and Hibernate Difference :

JPA defines the Specification. It is an API. But Hibernate is one of the implementations of JPA.

Using Hibernate directly would result in a lock in to Hibernate (There are other implementations such as Toplink)

when we use <property name="hibernate.hbm2ddl.auto">update</property> and drop tables if any, before using the below strategies

1. for generationtype.identity @GeneratedValue(strategy = GenerationType.IDENTITY)

it will create table (only for entity class) with Auto increment checked in DB. then for every insert, DB will generate auto incremented id. Mainly it helps in mapping the Primary key from DB to Java Object after the save/persist method is called.

2. for generationtype.auto @GeneratedValue(strategy = GenerationType.AUTO)

it will create table for entity and sequence table ( tablename\_sequence) and inserts the first row with id 1. The nextval will be in the sequence table. everytime it will fetch (select query is triggered) it from db and based on the value, our code increments it and uses the fetched id in the insert query of next object.

Note: Here, Auto increment will not be checked in DB side whenever we create the table.

3. for generationtype.sequence @GeneratedValue(strategy =GenerationType.SEQUENCE,generator = "book\_gen" ) @SequenceGenerator(name = "book\_gen",initialValue = 4,allocationSize = 1, sequenceName = "book\_Seq")

it will create table for entity and sequence table (with table name as book\_Seq) and inserts the initial value of 4 in the book\_seq table. Our code gets the value 4 from db and then updates the book\_Seq table with value 5. then, we insert the record into our entity table with id value 4.

4.for generationtype.table @GeneratedValue(strategy=GenerationType.TABLE,generator=”book\_gen”) @TableGenerator(name=”book\_gen”, valueColumnName=”book\_seq”,allocationSize=1)

It will create table for entity and sequence table (with table name as book\_gen) and inserts default value of 0 in the sequence table. Then if we tried to add data, we will fetch the 0 from book\_seq table and increment it by one (since allocationsize is 1) and assign it to the id field which is getting inserted and as well as update the book\_seq value from 0 to 1

A screenshot of a computer

Description automatically generated

Basically the difference between sequence and table is minor difference. In sequence, the value is getting fetched and assigned to the id field first and then getting incremented (only the sequence table next\_val will only get incremented). But in Table, we fetch the value and increment it and assign it to the id field (both sequence table value and id value gets incremented at the same time).

Know about @GeneratedValue annotation in this link 🡪 <https://codippa.com/generatedvalue-annotation-spring-boot-hibernate/>

| **#** | **SQL** | **HQL** |
| --- | --- | --- |
| 1. | SQL is database-oriented table query. | HQL is object-oriented query. |
| 2. | SQL manipulates data stored in tables and modifies its rows and columns. | HQL is concerned about objects and its properties. |
| 3. | SQL is concerned about the relationship that exists between two tables. | HQL considers the relation between two objects. |
| 4. | Native SQL is usually faster. | Non-native HQL is usually slower. |
| 5. | SQL offers complex interface to new users. | HQL provides user-friendly interface. |
| 6. | SQL stands for Structured Query Language. | HQL stands for Hibernate Query Language. |

MICROSERVICES :

**Coupling** – is a degree of interdependence of software modules, components or services.

High Coupling - Too much dependency

Types of coupling – Content coupling, common coupling, control coupling, routine coupling, data coupling, type use coupling, stamp coupling, import coupling, external coupling.

Content Coupling – One class can access the private members of another class

Common Coupling – Two classes access the same shared data i.e global variables and static properties

Control Coupling – When a function controls the flow of another function

Routine Coupling – When one function calls another function without passing any parameters.

Data Coupling – When two systems share the same database

Type use coupling – when a member or property of class B is of type class A

Stamp coupling – When in class B a method has a parameter of type class A

Import Coupling – When a library is imported into another program.

External Coupling – When communicating with an external program.

Cohesion – Is a degree to which all elements of a module, software or code are directed towards performing a single task.

* + High cohesion among the elements is GOOD
  + Example of low cohesion : Helper class

Microservice – Microservices are an architectural and organizational approach to software development where software is composed of small independent services that communicate with each other (one way is through API) in a loosely coupled manner.

* + With Monolithic architecture, all processes are tightly coupled and run as a single service.
  + But in Microservice, each process is made into a single service and they are executed independently, which solves the problems of monolithic applications.

PROS and CONS of Monolithic Apps :

GOOD:

1. Simplicity : Monolithic apps are easier to build, test and deploy.
2. Cross cutting concerns : One piece of code can handle monitoring, logging, security etc.
3. Performance : Different classes or functions communicate directly and share the same memory so the application runs faster compared to microservices where services communicate through a network.

BAD:

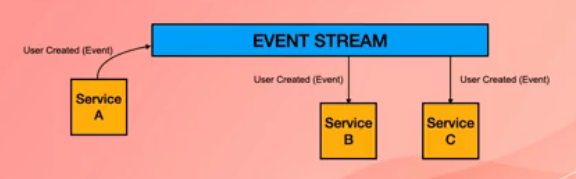
1. Reliability: A fatal error in any part of the application will crash the entire system.
2. Updates and Deployment: Even for a small change the entire app has to be re-deployed.
3. Tech stack: The entire application has to be developed with one single tech stack.

Alternatives of Monolith Apps :

1. Component Based Monolith Apps : App is made of multiple binary files (DLL files in windows). All libraries run as part of a single process and app.
2. Service Oriented Architecture : Application is broken down to services. Services communicate via SOAP or Restful APIs. Normally same techstack is used. Services use a shared database.
3. Microservice Architecture : Services are independent and can be used in any project or application. SOAP APIs are deprecated. Microservice can be called in an asynchronous manner. No shared database. Can mix and match techstacks.

Benefits of Microservices :

1. Microservices can be deployed independently
2. Microservices can be scaled out independently.
3. Microservices offer better fault tolerance.
4. A smaller code base allows new team members learn the code easier and quicker.
5. Different microservices can be built with different technology stacks.
6. Microservices can communicate asynchronously (using apache kafka – by publishing and consuming events) and not by direct API calls.



Drawbacks of Microservices :

1. Microservices-based systems are more complex to design and build
2. Requires changes in how teams and organizations work (More teams to manage, often more developers are required, teams have to collaborate more efficiently)
3. Often more expensive to build.
4. Diagnosing problems become difficult.
5. Testing becomes more challenging.

Micro services relate to business :

1. Business prefer to use cloud i.e AWS as opposed to on-prem infrastructure to avoid a large capex
2. Micro services are a great fit for cloud
3. Micro services enable businesses to better implement agile
4. Micro services allow businesses to out-source part of their development effort, and get to market quicker

Name some design patterns and tools we use in developing microservices :

1. Event Streaming – Apache Kafka
2. Message Broker – Apache Active MQ
3. Containers & Dockers – Kubernetes or Docker container
4. No-SQL Database –
5. Restful API –
6. API Gateway –

Blast Radius & Resiliancy :

Blast Radius is the degree to which the entire system is affected if a micro service falls or shutsdown.

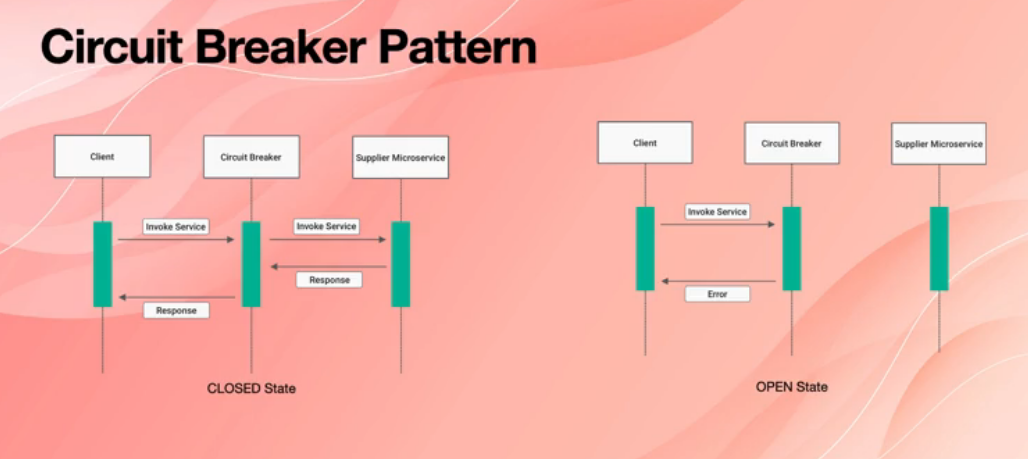
1. In a monolithic application the blast radius is 100% (If part of application encounters a fatal error or if the database becomes unavailable (due to misconfiguration, overloading etc) or breaks)

In micro services architecture we aim at reducing the blast radius with patterns of resilience.

1. Independent database for each microservice
2. Timeouts (the timeout for which the other microservice needs to respond or else we need to handle it)
3. Backoff strategy (retries) (the number of times it should try to communicate when api call failure happens)
4. Circuit breaker pattern
5. Bulkhead pattern
6. Fallback pattern (If a microservice cannot handle a request, it hands it over to other microservice (backup – not meant to have full functionality) to handle
7. Asynchronous communication (no point-to-point calls)

Circuit Breaker Pattern (Applied to API (point to point) calls and not asynchronous calls via event streaming) :

1. We set a threshold for failing API calls i.e 3. Meaning if a microservice fails for more than three times while trying to handle the request, then it will reject api calls.
2. If failing API calls exceed the threshold we reject the API calls
3. Circuit breakers are micro services themselves and act as proxy.
4. Circuit breakers can have three states : Open (doesn’t allow api calls), Closed ( allows api calls), Half-Open (may or may not allow api calls).



For Closed State : The client needs to make an api call to the supplier micro service. The inbetween circuit breaker passes the request to supplier and returns the value back to clients from supplier. So it acts like a proxy.

For Open state : When client makes an api call , the circuit breaker decides that supplier is not able to respond. So it returns an error to client and client needs to handle the error.

Note : If we have one circuit breaker paired microservice, its going to double the number of microservices. If we have one centralized circuit breaker service, its going to have single point of failure. For this reason, sometimes microservices can have circuit breaker built into them.

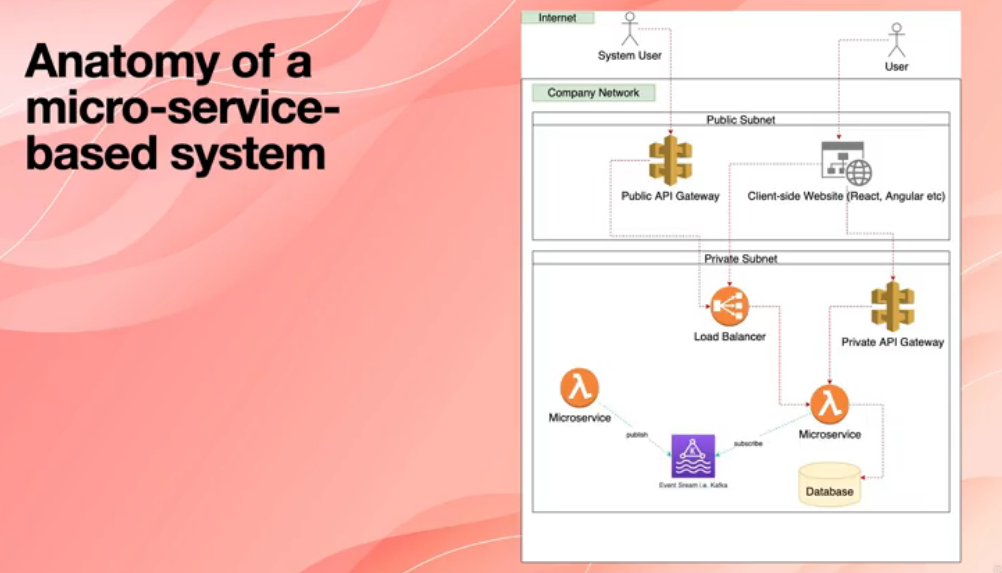
For payment services where we have more number of transactions, we would limit them by having a separate circuit breaker microservice. So that the supplier doesn’t get overloaded.

For other microservices, we can have built in mechanism for circuit breaking.

Ways of building microservices :

1. Single-tenant micro services – One micro service on one virtual server.
2. Containerised micro services – Each micro service runs as a docker container. (with help of orchestrator Ex: Kubernetes)
3. Serverless – Small micro services that run without a server in the cloud i.e Amazon lambda in AWS.
4. Spring Boot in Java (use on prem servers or cloud too)

MICROSERVICE ARCHITECTURE :



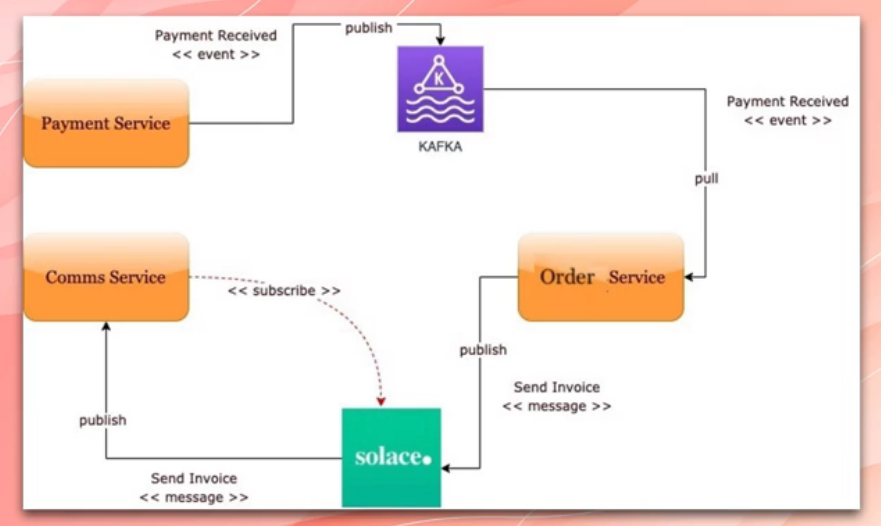
Difference between Event and Message :

Event :

1. Has already happened
2. Happened in the past
3. Order cannot be changed
4. Handled by event streaming platforms such as Apache kafka

Message : (It is called command in microservice perspective)

1. We ask this to be done
2. Order and priority can change
3. Can be sent via API calls or a message broker
4. Handled by message brokers such as Apache Active MQ, Rabbit MQ, Solace

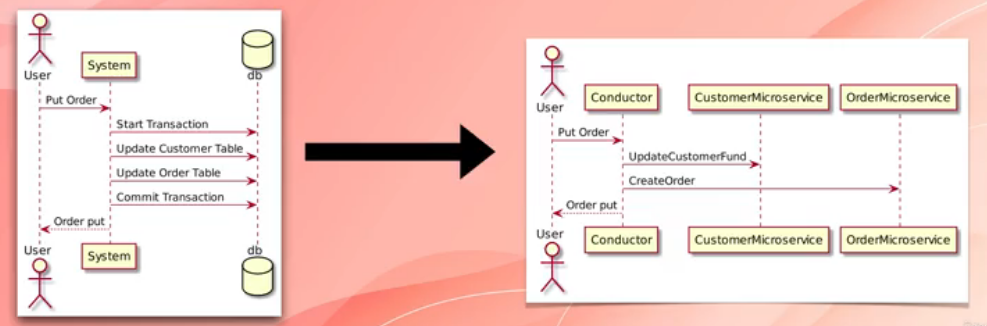


We have 3 services – payment, order and communication service

If a customer pays for the item he purchased, an payment received event happens and it is getting published in apache kafka. Order service subscribes to it and receives that event. Order service wants to send an invoice to the customer. So it sends a message to the message broker solace abou the invoice. Communication service subscribes to it and receives that message.

Distributed Transaction :

1. In micro service based architecture, each service has its own database
2. We need to make sure transaction (insert, update, delete) are ATOMIC across all services
3. Managing transactions across multiple services is Distributed Transactions



There are two patterns for handling distributed transactions

1. Two-phase commit Transactions (2pc) (not recommended)
2. Saga pattern

**Two-phase commit Transactions (2pc) :**

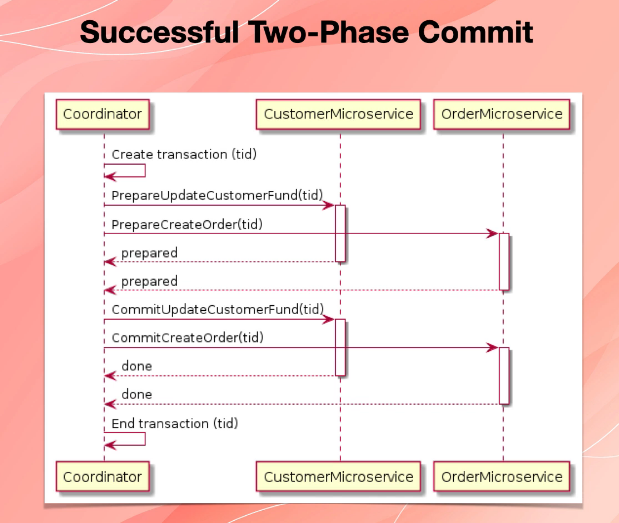
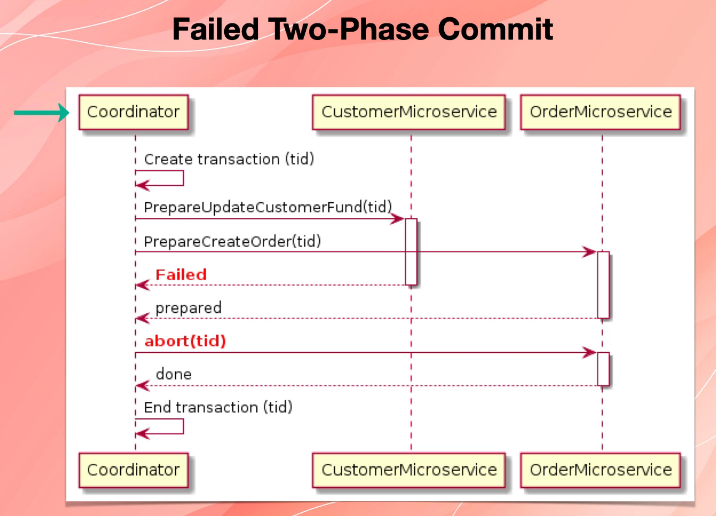
Two-phase commit protocol (or 2PC) is a mechanism for implementing a transaction across different software components (multiple databases, message queues etc.)

One of the important participants in a distributed transaction is the transaction coordinator. The distributed transaction consists of two steps:

* Prepare phase — during this phase, all participants of the transaction prepare for commit and notify the coordinator that they are ready to complete the transaction
* Commit or Rollback phase — during this phase, either a commit or a rollback command is issued by the transaction coordinator to all participants

The problem with 2PC is that it is quite slow compared to the time for operation of a single microservice.

Coordinating the transaction between microservices, even if they are on the same network, can really slow the system down, so this approach isn't usually used in a high load scenario.

For More Depth explanation, Visit : <https://www.baeldung.com/cs/saga-pattern-microservices>

Problems with 2PC :

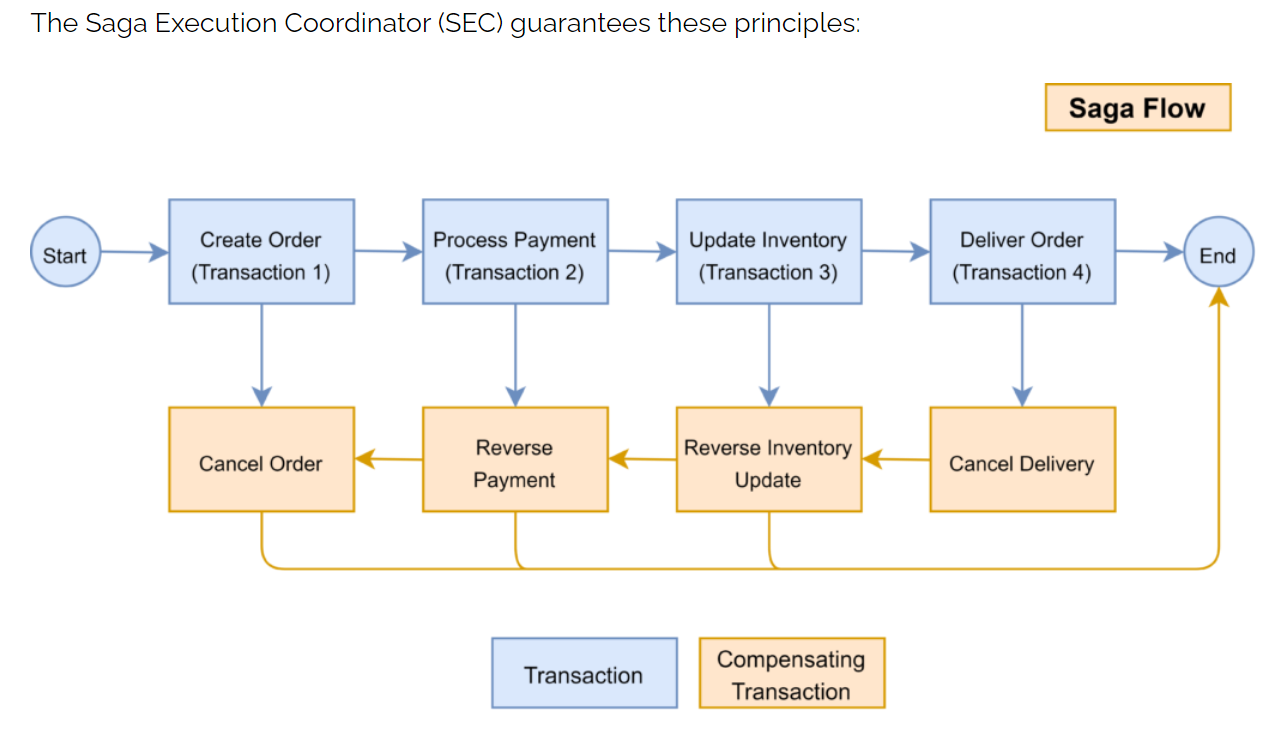
* The onus of the transaction is on the coordinator node, and it can become the single point of failure.
* All other services need to wait until the slowest service finishes its confirmation. So, the overall performance of the transaction is bound by the slowest service.
* The two-phase commit protocol is slow by design due to the chattiness and dependency on the coordinator. So, it can lead to scalability and performance issues in a microservice-based architecture involving multiple services.
* Two-phase commit protocol is not supported in NoSQL databases. Therefore, in a microservice architecture where one or more services use NoSQL databases, we can’t apply a two-phase commit.

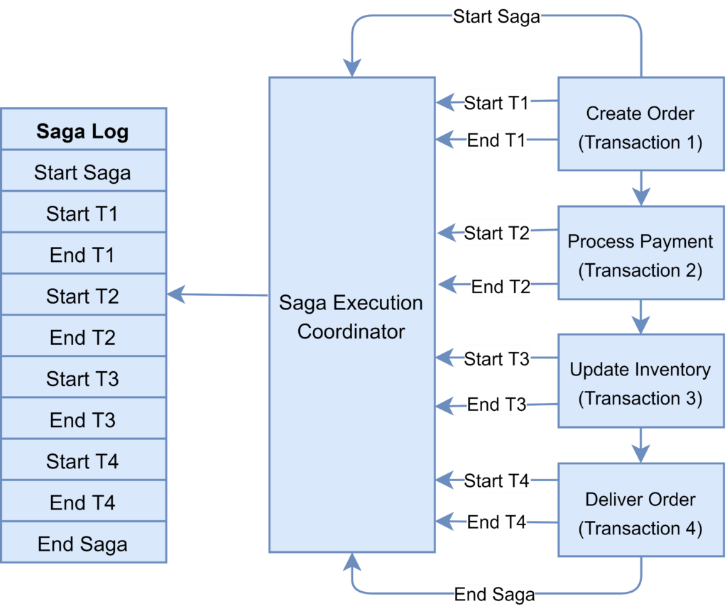
**Saga Pattern :**

The Saga architecture pattern provides transaction management using a sequence of local transactions.

A local transaction is the unit of work performed by a Saga participant. Every operation that is part of the Saga can be rolled back by a compensating transaction. Further, the Saga pattern guarantees that either all operations complete successfully or the corresponding compensation transactions are run to undo the work previously completed.

In the Saga pattern, a compensating transaction must be idempotent and retryable. These two principles ensure that we can manage transactions without any manual intervention.





There are two approaches to implement the Saga pattern: choreography and orchestration

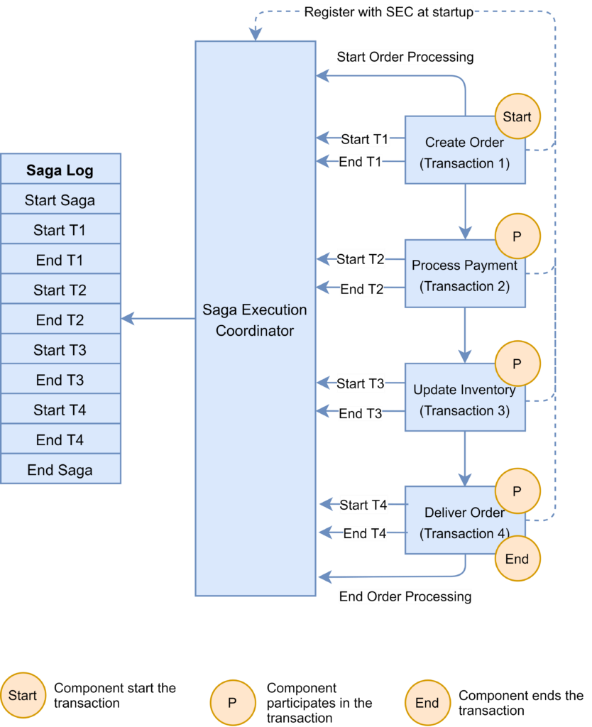
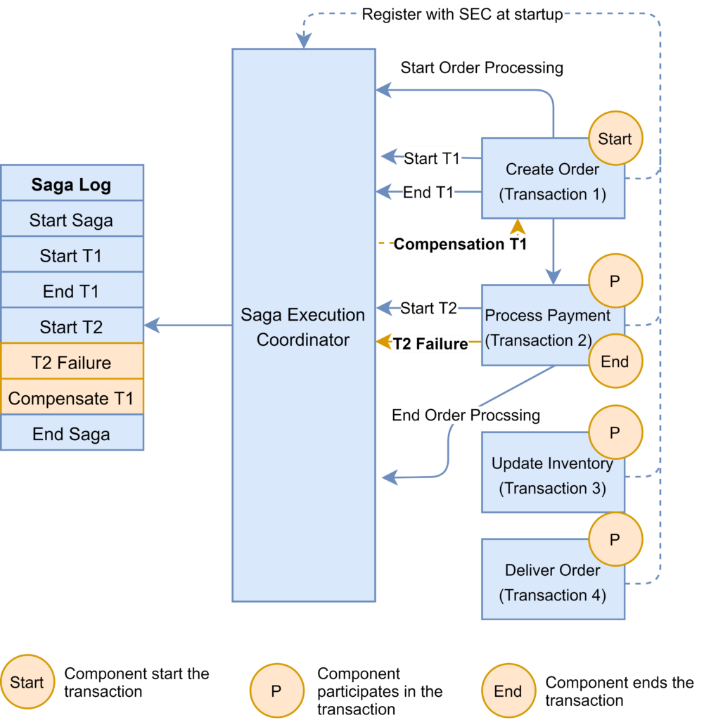
**Saga Choreography Pattern:**

In the Saga Choreography pattern, each microservice that is part of the transaction publishes an event that is processed by the next microservice.

To use this pattern, we need to decide if the microservice will be part of the Saga. Accordingly, the microservice needs to use the appropriate framework to implement Saga. In this pattern, the Saga Execution Coordinator is either embedded within the microservice or can be a standalone component.

In the Saga, choreography flow is successful if all the microservices complete their local transaction, and none of the microservices reported any failure.

The following diagram demonstrates the successful Saga flow for the online order processing application:

In the event of a failure, the microservice reports the failure to SEC, and it is the SEC’s responsibility to invoke the relevant compensation transactions.

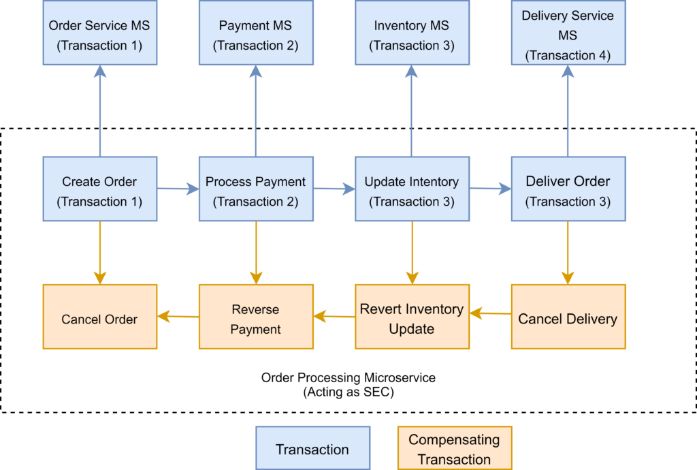
In this example, the Payment microservice reports a failure, and the SEC invokes the compensating transaction to unblock the seat. If the call to the compensating transaction fails, it is the SEC’s responsibility to retry it until it is successfully completed. Recall that in Saga, a compensating transaction must be idempotent and retryable.

Different Frameworks available to implement choreography pattern : Axon saga, Eclipse Microprofile LRA, Eventuate Tram saga, seata.

**Saga Orchestration Pattern :**

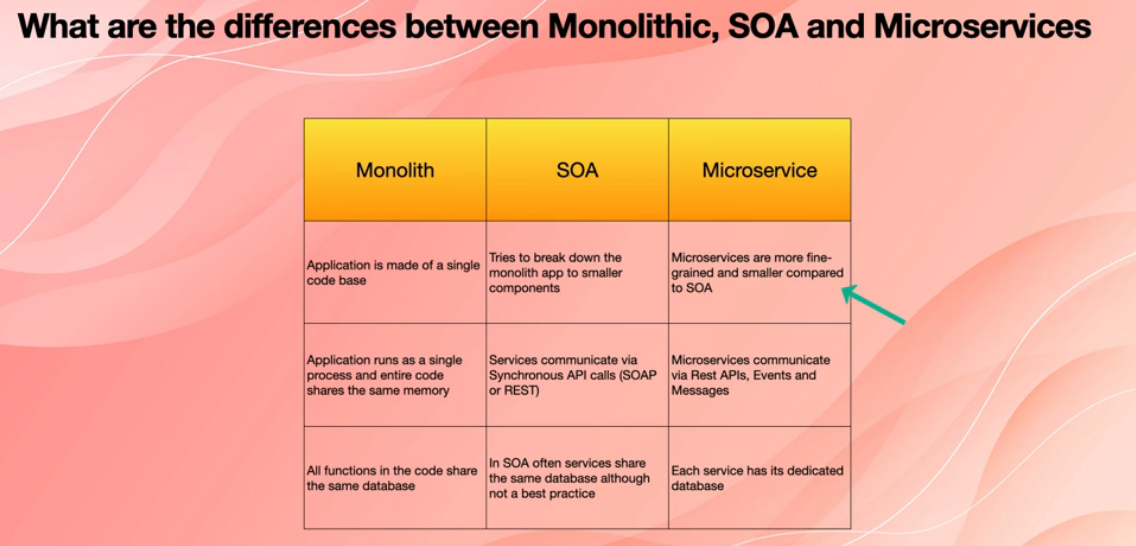
In the Orchestration pattern, a single orchestrator is responsible for managing the overall transaction status.

If any of the microservices encounter a failure, the orchestrator is responsible for invoking the necessary compensating transactions:



Frameworks available to implement orchestrator pattern : Camunda and Apache Camel

Difference between Monolithic apps, Service oriented apps (SOA), Microservices



**Bounded Context :**

Bounded context is a concept in Domain Driven Design (DDD)

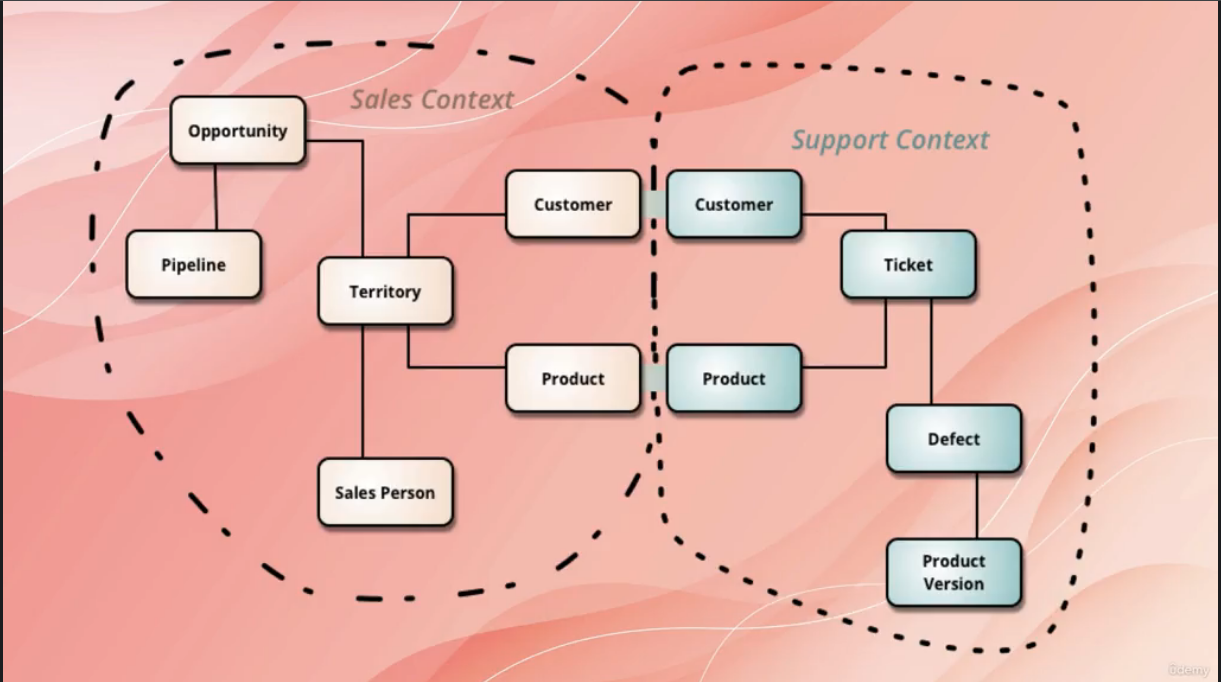
DDD is a framework of analysing and modelling large problems, models and teams

The entire problem is the domain

A domain model is the representation of a real thing in the world. For Example : User, Package, Drone

A bounded context is simply the boundary within the domain where a particular domain model applies.

Normally one microservice represents one bounded context.



SQL IMP QUERY :

**To find nth highest salary**

SQL> select top 1 salary from (

Select top n salary from table order by salary desc

) as SAL order by salary asc;

In order to calculate the second highest salary use rownum < 3

In order to calculate the third highest salary use rownum < 4

Important points for sql questions

* The primary key will not accept NULL values whereas the Unique key can accept NULL values.
* A table can have only one primary key whereas there can be multiple unique keys on a table.
* A Clustered index is automatically created when a primary key is defined whereas a Unique key generates the non-clustered index.
* A Primary Key can be a Unique Key, but a Unique Key cannot be a primary key.

https://sqlrelease.com/sql-server-tutorial/types-of-keys