**Contents**

[**Java Features**](#Java_Features)

[**Java8 Features**](#Java8_Features)

[**Struts**](#Struts)

[**Kerberos Interoperability**](#Kerberos_Interoperability)

[**WinDbg**](#Windbg)

**[Team Foundation Server](#TFS)**

**[Visual Studio Online](#VS_Online)**

[**JIRA**](#Jira)

[**J2ee Features**](#J2ee_Features)

[**J2ee Application Design**](#J2eeAppDesign)

[**EJB**](#EJB)

[**Servlets**](#Servlets)

[**JSP**](#JSP)

**[XML Parsing](#XMLParsing)**

[**J2ee Project Creation**](#J2ee_Project)

[**JMS**](#JMS)

[**Spring**](#Spring_Framework)

[**Hibernate**](#Hibernate)

[**Integration Strategies and Patterns**](#IntegPattrens)

[**Design Patterns**](#Des_Patterns)

[**J2EE Patterns**](#J2ee_Patterns)

[**Identity Federation**](#Identity_Federation)

[**Load Balancing**](#Load_Balancer)

[**SOAP Web Services**](#SoapWeb)

[**Spring Boot**](#SpringBoot)

[**Securing Micro services**](#SecureMicroService)

[**Persistence In Web Application**](#PersistenceWebApp)

[**Blogging Web Application**](#BloggingWebApp)

[**Creating Web Application**](#CreateWebApp)

**[Testing Web Application](#SpringRestfulWebservice)**

**[Spring Restful Webservice with HATEOAS](#SpringRestfulWebservice)**

**[Web Development using HTML, JSP, JQUERY, Boot strap & CSS](#WebUsingHTML_JSP_CSS)**

[**MEAN Stack Angular Frontend**](#MSAF)

[**Database Migration from SQL to NoSQL**](#DatabaseMigration)

**[Mongo DB](#MongoDB)**

**[SQL Statements](#SQL_Queries)**

**JAVA Features**

1. **Class Loader Subsystem** is responsible for Loading & Initializing classes. It reads **.class file** and creates instance.

* Bootstrap class loader loads core java classes from JAVA\_HOME/jre/lib directory.
* Extension class loader from extensions directories JAVA\_HOME/jre/lib/ext (**Extension path**).
* System or Application class loader i from application class path.

An application delegates to Bootstrap class loader, then Extension class loader and finally Application class loader.

***Application 🡪 Bootstrap class loader -> Extension class loader🡪 Application class loader***

1. **Java Native Interface (JNI)** enables JAVA code to interact with native applications/libraries written in C/C++
2. Threads can be created by **Extending Thread class** or Implementing **Runnable Interface**. Thread class provides basic thread functionality with inbuilt methods like yield(), interrupt() etc. It cannot extend any other class as MI is not supported.
3. A **thread pool** manages the pool of worker threads. Each worker thread implements Runnable & has a service method. **Executors class** in JAVA manages threads in a pool
4. **ExecutorService of concurrent API** replaces need for working with threads directly or manually creating threads. It transparently manages a pool of threads. It also works with Callables & Futures. **Callables** are functional interfaces which return a value. They raphe submitted to executor services like runnables. **Executor** returns a result of type **Future** which can be used to retrieve the actual result at later time. get() method of Future blocks the current thread and waits until the callable completes and then returns actual result. A **timeout** can be specified as parameter to get() to return in case callable takes too long
5. Executors support **batch submitting of multiple callables** as a collection through **invokeAll(),** returning Future for each callable. **invokeAny()** does not return future objects, instead method blocks until the first callable terminates and returns the result.
6. **ScheduledExecutorService** can schedule tasks to run either periodically or once after an amount of time
7. Unlike fixed size thread-pool, **ForkJoinPools** get created for a given parallelism size which by default is the number of available cores of CPU.
8. **Synchronization** is implemented through **monitors** owned by a single thread at a time. Synchronized blocks ensure only one thread accesses critical section at a time. With Method Synchronization**,** a method can be invoked at a time by a thread. A part of method can also be Synchronized

//Sync block

public void run()

    {

**// Only one thread can send a message at a time.**

        synchronized(sender)

        {

          sender.send(msg);

        }

    }

**//Method Sync**

public **synchronized** void send(String msg)

{

}

**//Partial method Sync**

 public void send(String msg)

    {

        synchronized(this)

        {

            System.out.println("Sending\t" + msg );

            try

            {

                Thread.sleep(1000);

            }

            catch (Exception e)

            {

                System.out.println("Thread interrupted.");

            }

            System.out.println("\n" + msg + "Sent");

        }

    }

1. JAVA uses **wait(), notify() and notifyAll() methods** to avoid polling that consumes CPU cycles. Wait() tells the calling thread to give up the lock and go to sleep. Notify() wakes up one single thread that called wait(). notifyAll() wakes up all the threads that called wait() on same object
2. A **CountDownLatch** is used to make sure that **a task waits for other threads** before it performs an operation. This latch is then passed as an argument to Worker threads created by this task. These Worker threads decrement Latch value using its **CountDown() method**
3. **Start() function of thread** creates a **separate call stack for a thread** and then run() method is called by JVM
4. **Thread scheduler** is part of JVM. Any thread in a runnable state can be chosen by scheduler. Scheduling is influenced by methods sleep(), yield(), join() & setPriority(). **Sleep()** gives back control to CPU and blocks specified thread till specified time elapses. **Yield()** does not block a thread, it gives CPU a chance to pick up another thread. **Synchrozied()** is entering a unit of code in which thread cannot be interrupted. No other thread can run this unit of code while current thread is there. **Blocked thread** is one that is waiting for some action like network connection.
5. **Blocking Queues** are useful in a case of **producer consumer pattern**. It is used for passing data from a thread to another. Its **take ()** method tries to take element from queue, if no element there it waits. Same behaviour with put() method. In JAVA Blocking Queue is a part of **util.concurrent** package. A Blocking Queue is created as a list or array

*Private static BlockingQueue<Integer> queue = new* ***ArrayBlockingQueue****<Integer> (10);*

*BlockingQueue<String> queue = new* ***LinkedBlockingQueue****<Integer> (10);*

1. To make an object eligible for **garbage collection**, variables referencing it should **be set to null.** GC can be called at any time once an object is eligible for garbage collection. **Finalize() method** allows for reclaiming native resources which can’t be collected by garbage collector. It is used to clean up system resources such as database connections. Exceptions occurring in finalize are ignored by GC. **JVM can be forced** to call finalize method as well as GC

*System.gc()*

*System.runFinalization()*

1. **Finalization queue of JVM:** GC maintains 2 queues - Finalizationqueue (FQ) & F-Reachable queue (FRQ). When an object is created its ref is **also placed in finalization queue**, if it implements finalize() method. Those objects without finalize method are first collected. GC checks FQ for objects that are marked for collection. It moves those object refs to FRQ. After reaching FRQ they are considered as garbage and are ready for finalization. There is another thread that scans FRQ and **calls finalize() method for each object there**. After that these objects references are removed from FRQ. Next time when GC runs, these objects are marked as garbage and collected
2. An Application has a **set of roots** which refer to storage locations pointing to objects on managed heap. App roots include global/static objects, local values and parameters. When GC runs it starts moving to all reachable objects and builds graph like structure. It does it for all roots. Rest objects are considered garbage which GC frees and compacts memory of non-garbage objects. Application remains suspended during GC and compaction
3. When a variable of a class type is declared, **only its reference is created**, while memory is only allocated for it with new().
4. **Generational Garbage Collection:** heap space is partitioned **into young and old generation,** with new objects allocated in **young generation where GC happens frequently**. Objects surviving a threshold number of GCs are moved to Old generation, which is a bigger area on heap with less frequent collections. When **older generation gets full** a complete serial GC happens with **mark sweep and memory compaction**
5. In JAVA **diamond problem** can exist since an Interface can inherit from multiple Interfaces. This can be resolved using super keyword by class implementing this interface. If a class defines a method it is always given priority over method having same signature in an interface. This rule avoids future diamond problems if some default method as in JAVA8 gets added to interface. Already existing class method then gets priority.
6. **BigDecimal** is represents numbers with higher precision than double an is better suited for currency fields.
7. **Adapter** pattern provides interface conversion between systems. A **Decorator** adds extra level of indirection to support intelligent access. It is common with shopping combinations. **Template** pattern provides an outline of a flow and allows customisation of some steps i.e. MFC, other frameworks, Comparator. **Composition** allows switching to different implementation at run-time. Composition Code is more testable than inheritance hierarchy
8. **Memory Leaks in JAVA**: Memory leaks are caused unwanted object references, long living static objects or native resources accessed through JNI. Object added to a collection is not removed i.e. a listener that has got lapsed. Static variables are ones which exist throughout application life cycle, and hence **will never be garbage collected**. A good practice of ensuring memory release is setting **reference to null of static or instance variables** is at point after their use is over.
9. Java has **tools for monitoring JVM** and applications. **Jvisualvm tool** with java1.6 analyses heap dump and displays objects having memory leak. It is a visual tool. Objects displayed on top are leak instances. **Heap dump** is created using **JMap**. **GC log option argument** on command line shows statistics at different times when GC gets run. It gives info on whether program is leaking memory. **JConsole** is a UI tool used to monitor performance of app running on local & remote machines. It takes Process id of application to monitor. For web app process ID of app server is given.

*java verbose GC log option*

*jconsole <processID>*

*jmap -dump:format=b,file=heap.bin<pid>*

*jvisualvm*

1. The **try-with-resources statement** (Java 7) ensures each resource is closed at the end of the statement. JAVA 7 allows to handle **multiple exceptions** in the same catch block

*try (*

*java.util.zip.ZipFile zf =*

*new java.util.zip.ZipFile(zipFileName);*

*java.io.BufferedWriter writer =*

*java.nio.file.Files.newBufferedWriter(outputFilePath, charset)*

*)*

*{*

*///perfrom actions here*

*}*

1. **Vector vs. Array List**: Vector is thread synchronized while ArrayList is not. ArrayList is preferred if requirement is for non-synchronised list which is faster
2. **Concurrent Hash map vs. Synchronized Map & Hash Table:** A Concurrent Hash map has a **segment wise lock feature** while others implement **collection wide lock** and that renders them non scalable in multithreaded environment. Both Hash table & Hash map implement JAVA map interface, but **Hash map is not synchronized** & can store null values.
3. If a reference variable is pointing to class type object it is **strong reference** & object pointed to by it cannot be garbage collected.
4. A **soft reference** is garbage-collected less aggressively only when GC algorithm decides memory is low. They are **useful in caching**, letting objects remain on heap till there is enough space.
5. **Weak Reference** is useful for situation when an object is used multiple times repeatedly during an event, and then, is expected to be used after a long time. Program can recreate underlying object whenever get() method returns null
6. Elements in a **WeakHashMap** can be garbage collected if there are no other strong references to the key object
7. When Java process starts, memory gets assigned to it, part of which is used to build heap space and part in stack space. Memory related methods **from Runtime class** are used to obtain free memory/total memory etc.
8. **equals() method** checks for logical equality of objects, they may not point to same memory space. **hashCode()** method returns an integer hash value for an object. Objects identical to each other should have the same hash code, but not equal objects may also generate hash code.
9. **Compile time constants** are defined as Public static final variables. They are substituted with actual values at compile time. When using constants from third party library, if any change is made in its value in library code, that will not be reflected in application using it. Program should be recompiled
10. A **List** is ordered while **Set** is unordered with no duplicates. **PriorityQueue** guarantees that highest priority elements remains at the head of the queue followed by elements next high priority
11. **Comparator** interface can define custom sorting order based on any field. **Comparable** defines natural order on single field
12. A **volatile variable** is not cached and all read-writes are done from **main memory**. It is useful when developing embedded systems or device drivers where it is required to read or write a memory-mapped hardware device.
13. Declaring an **array as volatile** does not give volatile access to its contents. **AtomicIntegerArray** class implements integer array whose individual fields can be accessed with volatile semantics, through get() /set() methods
14. A thread should always call **wait() in a loop** on a condition, so that it resumes only if condition gets true & not on notify() action of another thread. Unlike wait(),**sleep()** does not release lock
15. **False sharing** applies when one thread impacts performance of other thread by modifying independent variables that share the same cache line. **A cache line** ranges from 32-256 contiguous bytes in size. Different processor cores can pull bits of data from main memory into same cache line. CPU cores invalidate memory chunks at the level of cache lines. Even if threads are accessing different fields of an instance on cache line, it is invalidated every time, as cache subsystem flags whole line as dirty
16. Java 8 introduced **@Contended annotation** which instructs the JVM to add 128 bytes before and after the ‘hot’ annotated field. This makes it unlikely that adjacent field will end up in the same cache line as hot field.
17. **JOL (Java Object Layout) toolbox** analyses object layout schemes in JVMs. This toolbox uses Unsafe, JVMTI, and Serviceability Agent (SA) to decode object layout with more accuracy than tools relying on heap dumps. **Sun.misc.Unsafe**, provides an alternative to low-level programming using a Java API (apart from JNI). Unsafe class is intended to be only used by core Java classes

**Java Virtual Machine Tool Interface (JVMTI)** allows a program to inspect the state and control execution applications running in JVM. It is used by debuggers or profilers. **Serviceability Agent (SA)** helps debugging Java applications at both Java and JVM level. It can debug live Java processes and crash dump files.

1. A **Disruptor** sends messages between threads in the most efficient manner. It can be used as alternative to queue, but also shares features with SEDA. **LMAX Disruptor** frameworks, provides a high-performance inter-thread messaging library
2. **Staged event-driven architecture (SEDA)** approach to software architecture decomposes complex, event-driven application into stages connected by queues. It decouples event and thread scheduling from application logic and avoids overhead associated with thread-based concurrency models (locking, unlocking, polling for locks)
3. **Busy spin technique** is used to wait for events without freeing CPU. This is done to avoid losing data in CPU cache, which is lost if the **thread gets paused and resumed in some other core**. It is useful if waiting is for a very little time
4. **Thread-local variables** are thread’s own copy of variables not shared with other threads. They can cause a memory leak in Java application if not taken away once its work is done. In webservers this is important as thread outlives any application variables
5. When a String is created **using new (), it is put on heap** & when created as a **literal, it is put in String pool**. String created on heap can be added to String pool through **intern().** StringBuffer and StringBuilder provide mutable means of manipulating string. **Methods of StringBuffer are thread synchronized** while StringBuilder (introduced later) is not.
6. Any alteration to **Immutable objects** result in a new object e.g. String, Integer, and other wrapper class.
7. A string is defined as immutable, to **prevent external tool from modifying memory location** which could contain sensitive information like contents of password file. With String as Immutable if such operation is attempted, a new block will get created.
8. **Java NIO (New IO)** is alternative to standard Java IO and Java Networking API's. It enables working with **channels and buffers**, performing non-blocking IO. Data is read data from channel to buffer while thread can do something else. A selector object can monitor multiple channels for **events** (connection opened, data arrived etc.)
9. **Non-direct byte buffers** are a wrapper around byte array and reside in Java Heap memory. **Direct byte** buffer is outside JVM and memory is not allocated from the heap. They are not affected by Garbage Collection. JVM performs native IO operation directly into buffer performing high-speed IO operation.
10. **Multithreaded code best practices**: assign a name to thread to help in debugging. Minimise the scope of **synchronization to critical section**, instead of entire method. Volatile is preferred over synchronization. **Concurrency utilities** are preferred over wait()/notify() i.e. BlockingQueue, CountDownLatch and Semaphore. Concurrent collection preferred over synchronized collection i.e. ConcurrentHashMap,
11. **Serializable interface** allows Java classes to be serialized to transmitted over network persist on disk. It uses default serialization of JVM. **Externalizable** allows full control over Serialization process
12. In JAVA, **Serialization** converts an object into binary format that can be persisted or transferred over network. A class can be made Serializable by **implementing Serializable interface** which is a marker interface. **Marker interface** does not have any methods. However WriteObject() method of output stream internally checks if object is of type Serializable. It then writes object else throws an exception. During serialization, a **SerialVersionUID gets stamped on object** when it gets serialized. It controls version of object and is used to recover object state on de- serialization

*Save obj = new Save();*

*File f = new File(out.txt);*

*FileOutputStream fos = new FileOutputStream(f);*

*ObjectOutputStream oos = new ObjectOutputStream(fos);*

***oos.writeObject(obj);***

1. Serialization is supported through **methods ReadObject() and WriteObject() of Object class**, however for these methods to work class should implement marker interface. All classes **aggregated/composed by a Serializable class** should implement Serializable interface. Serialization is based on recursive calls through reflection of all member variables (excluding transient) of super class as instance variables. Also **deserialization does not call a constructor**, so initializations in constructor may not be useful in deserialization
2. Externalization implements **Serialization through Externalizable interface**. It provides more control on what instance variables to serialize. Externalizable has **writeExternal() and readExternal() methods**, which must be implemented by class. If writeObject() method is called for object implementing Externalizable, it results in an internal call to writeExternal

*Employee obj = new Employee(); //Employee() implements* ***Externalizable interface***

*File f = new File(out.txt);*

*FileOutputStream fos = new FileOutputStream(f);*

*ObjectOutputStream oos = new ObjectOutputStream(fos);*

***oos.writeObject(obj); //result in call to Employee.writeExternal()***

1. **Peer to Peer Cluster**: A Simple architecture in which nodes can join in or leave cluster anytime, through algorithms adding or removing them from network. **Peer Discovery** Mechanism through J-Groups, UDP Multicast, RMI, JMS etc. It support below configurations
   1. **Full Replication** where each peer node maintains a **full copy of cache**, with updates broadcast to other nodes. It is **Good for write once read multiple applications** but impacts scalability and becomes slower if application writing frequency increases, as each write triggers multiple peer updates
   2. **Distributed Hash Table** where each node is **responsible for a range of keys** and though backup copies are there, full replication never happens. It has **better performance with cache update operations** but read operations are slow due to network overhead. It can be useful in **building services** such as - Cooperative web caching, Distributed file system, Domain name services, Instant messaging multicast, peer to peer file sharing, content Distribution systems
2. In **a distributed cache**, Data is stored in a JVM & then distributed across multiple JVMs. Distribution happens using custom protocol between nodes hosting JVMs. There are 2 types of distributed caches – a Replicated cache & a Partition cache. A **combination of both caches** can be **used by an Application**
3. In **Replicated caches** a PUT operation **write copies to all JVMs** & GET reads from any JVM. Replicated caches do not scale for writes but have a **good read performance**. It is used with small datasets with **lesser writes** but lots of reads
4. In **Partitioned caches**, data is stored in distributed **partitions based on Hash of key**. It makes 2 copies, a primary on partition determined by hash key and backup data as per algorithm chosen. GET retrieves primary data. Partitioned caches scale for write operations to capacity of cluster.

**Java 8 Features**

1. Ability to utilize **Multi-core** processors and **Functional Programming features**
2. Anonymous inner classes are replaced by **Lambda expressions** in Java8**.**

*( ) -> System.out.println("Hello World");*

***//Transaction below is an i/f with single method***

*Transaction obj = (int amount) -> {*

*System.out.println(“Withdrawl = “ + amount);*

*}*

1. Lambda expression doesn’t allow **re-declaration of local** variables in enclosing scope, though this is permitted by anonymous classes. There variables are **effectively Final** for lambda. But they can be used inside lambda expression

*int localVar = 1;*

*//String str = "Hello"; ---will generate compile error*

*Consumer<String> consumer1 = str -> {*

***//int localVar = 2; ---will generate compile error***

*Int x = localVar; //will work*

*System.out.println ("X= " + str);*

*};*

1. Instance variables can be re-declared inside lambda and can be referenced through **this keyword**
2. Java Interfaces have been extended to support **default methods** to account for changing interface signatures**.** Interfaces can be extended without breaking implementation. Default methods have **own implementation** and can also be extended by class

public interface Interface1 {

void method1(String str);

**default** **void log(String str)**{

System.out.println("I1 logging::"+str);

}

}

1. Interface **static methods** are similar in behaviour to default ones except they offer a functionality which should not be overridden
2. A class implementing multiple interfaces having one or more default **methods with** **same signature** is forced by compiler to implement those methods.
3. **@FunctionalInterface** can be applied to interface having single abstract method, but may contain multiple **default or static methods**. Interface can contain other abstract methods i.e. equals(), hashCode() etc. belonging to Object class.
4. **Java.util.function package** contains Functional Interfaces suiting requirements for lambda expressions such as **Predicate<T>** which can be used for comparator.

1. **Exception handling** in lambdas is done through a wrapper around lambda
2. **Closure with Lambdas**: In a closure compiler keeps track of value of a variable, & uses it later when function is executed out of scope.
3. **this reference** doesn’t work directly with lambda expressions as it would work with inner class
4. **Stream API** Allows taking advantage of multiple CPU cores & writing concise code. **Streams** can **process** large amount of data**.** Each collection has introduced a **stream** and a **parallelstream method.** Stream API offers **Database like operations** i.e. Group by, Order by etc., Lazy operations, Parallel operations, Internal iteration & Pipelining
5. **Collectors class** of streams provide methods to group and aggregate data in a stream based on criteria, specified using **Predicate or Function** interfaces

*A predicate class used to group employees less than 40 years of age*

*Predicate<Employee> pc1 = (Employee e)-> e.getAge() <40;*

*A Function class returning an employee name in upper case. This will get used in mapper method of collector*

*Function<Employee, String > fUpper = new Function(){*

*@Override*

*public Object apply(Object t) {*

*// TODO Auto-generated method stub*

*Employee e = (Employee)t;*

*return e.getName().toUpperCase();*

*}*

*};*

*Group employees on State and print their name in upper case*

*System.out.println(employees.stream().collect(Collectors.groupingBy((Employee e) -> e.getState(),*

*Collectors.mapping(fUpper, toList()))));*

1. Collector class has **methods** **toList(), toSet(), toMap(), toCollection()** etc. Method toCollection () can convert output to multiple data structures i.e. ArrayList, Stack, Vector, Hash Map etc.
2. A Predicate is a Functional interface with method test, taking in object and returning a Boolean. Additionally it has default methods for AND, OR & NEGATE operations. Both predicates and Functions can be passed around as parameters through lambda expressions.
3. A Function is a Functional interface with a method apply, taking in object and returning an Object. It has default methods compose and andThen
4. An **IntFunction** is specialization of Function, taking in an Integer and returning an object

1. **Print operation** on any instance invokes **method toString()** if its implemented in object, else Object class implementation of toString() is called
2. Java7 supports **external iteration** over a collection through for loop

*List<Integer> list = Arrays.asList(1,2,3,4,5,6,7,8,9,10,11,12);*

*List<Integer> outList = new ArrayList<> ();*

*For (int X: list)*

*outList.Add(X);*

*For (int X: outList)*

*System.out.println(X);*

X represents element in list instead of index

1. Java8 supports **internal iteration** on collections through forEach

*List<Double> outList = new ArrayList<> ();*

*list.forEach(X -> outList.Add(X);*

*outList.ForEach (System.out.println);*

1. Streams can be generated from Collections i.e. List, Map etc. , Literal Strings, Arrays and using methods i.e. generate, iterate, build etc. **Stream operations** i.e filter, sort, distinct, limit etc. return a stream and can be chained

*map.values().stream().filter(x -> x.contains("e")).sorted().forEach(System.out::println);*

*creates a stream on map, filters, sorts and prints resultant stream*

*"a1bc3d4efg567h8i9".chars().filter(x -> x > 65)*

*.forEach( x -> System.out.println(Character.toChars(x)));*

*Converts a literal string into stream*

*Stream.generate(() -> Math.random()).limit(10).forEach(System.out::println);*

*Generates a stream of 10 random numbers*

*Stream.iterate(0, x->x\*2 + 1).limit(20).forEach(System.out::println);*

*Generates a stream using iterate*

1. **Map operation** on stream transforms an element into something else

*List <Person> list = new ArrayList<>();*

*list.stream ().map (p -> "Name = " + p.getName() + " Lastname = " + p.getLastName())*

1. **Stream operation allMatch** works like a filter but it short circuits wherever condition on stream element returns false. Other methods **noneMatch** returns false wherever condition on stream element returns true & **anyMatch** returns true wherever condition on a stream element returns true short circuiting remaing elements
2. **Supplier interface** works within a Factory pattern creating objects of any generic type.

*public class Factory*

*{*

*public <T> T getGenericObject(Supplier<? extends T> supplier)*

*{*

*T object = supplier.get();*

*return object;*

*}*

*}*

*Factory factory = new Factory();*

*Supplier <? extends Shape> s1 = new Supplier()*

*{*

*@Override*

*public Object get()*

*{*

*return Shape.CreateCircle();*

*}*

*};*

***///Using Supplier as anonymous class***

*Shape circle = factory.getObject(s1);*

***///Using supplier as lambda***

*Vehicle suv = factory.getGenericObject(()->Vehicle.CreateSUV());*

1. Java 8 allows **repeatable annotations** with different property on a class method. In JAVA 7 similar function was achieved through different annotation classes

*@AttributeOverrides({*

*@AttributeOverride(name = "street", column = @Column(name="HOME\_STREET")),*

*@AttributeOverride(name = "city", column = @Column(name="CITY\_STREET"))*

*})*

*Address address;*

This can be replaced with

*@AttributeOverride(name = "street", column = @Column(name="HOME\_STREET"))*

*@AttributeOverride(name = "city", column = @Column(name="CITY\_STREET"))*

*Address address*

1. **ForkJoinPool framework** is supported in Java7 & Java8. In Java7 ForkJoinPool object should be instantiated and **new ForkJoinPool().** In Java8 it is returned via static method **ForkJoinPool.commonPool().** Any job submitted to **ForkJoinPool** is of type **RecusrsiveTask<T> with** overridden Compute method where processing is performed. Parallelism is achieved on recursive calls to the job through fork(), with sub task being assigned a separate thread
2. **Optional** is introduced to avoid **null checks & NullPointerException**. It forces developer to think of handling null conditions. IsPresent() method of Optional class is used to check if there is an object inside Optional class. **Functional chaining** is performed with Optional class through its methods map(), ofNullable() & ifPresent().

*public Integer sum(Optional<Integer> a, Optional<Integer> b)*

*private Optional<Manager> manager = Optional.empty();*

*Optional<Manager> manager1 = Optional.of(new Manager(“John”));*

*Integer count= getSecondWord.andThen(getLetterCount).apply(“John”);*

*System.out.println(count)*

***//The above expressions are chained through Optional as below***

*Optional.ofNullable(getSecondWord.apply(“John”)).map(getLetterCount)*

*.ifPresent(System.out::println);*

1. JAVA compiler can **interpret types from context**. Java7 does type inference through **<> operator**. There is no need to specify parameters on RHS of expression

Map <Integer, String> map = new HashMap <> (); //doesn’t require Integer, String arguments again

In JAVA8 **type inference** is provided with lambdas. Lambdas are assigned to a variable of Functional Interface implementation. To transfer lambda expression to a function as a parameter, a function should take a functional Interface as parameter. Based on method signature in functional Interface compiler is able to infer types

public Interface Pricer {

public List<Trade> skew(List<Trade> trades, PriceSkewer skewer);

}

(trades, skewer) ->

{

///function logic - **compiler auto infers types of trader & skewer** based on Functional interface Pricer

}

1. **Spliterator** in Java8 enables parallel operation on a collection. It is **an alternative to fork join framework** for parallelism. By default a collection framework is not thread safe. Spliterator splits a collection, partitioning some elements as another Spliterator, dividing data into sub sized units for parallel processing. **TrySplit() method** of Spliterator divides elements in half. A custom implementation of Spliterator is required for a different behaviour

**Struts**

1. Struts are an **MVC based framework for developing web applications.** It comprises JSP, Servlet, Custom tags, message resources etc. in single package
2. The main components for struts framework are
   1. **Action Servlet**: controller class responsible for handling web request
   2. **Action class:** for business logic
   3. Action form: a java bean representing **forms and mapping with action class**
   4. Action Mapping: maps between object and action
   5. Action Forward: forwards result from controller to destination
3. Exceptions can be **handled programmatically** using try-catch or in **declarative way in XML** file within **<global-exception>** tag. Key refers to key present in Message.properties file, Type is class of exception, Path is page where control follows

<exception >  
  
      key="stockdataBase.error.invalidCurrencyType"  
  
      path="/AvailbleCurrency.jsp"  
  
      type="Stock.account.illegalCurrencyTypeException">

</exception>

1. Validation in struts is performed through a **validator framework comprising of XML configuration files**. These files are placed in WEB-INF folder to make them available in class path
   1. **Validator-rules.xml:** contains default struts validator definitions
   2. **Validation.xml:** file containing details regarding the validation routines applied to form beans

<!--  Validator plugin -->  
<plug-in className="org.apache.struts.validator.ValidatorPlugIn">  
  <set-property  
  property="pathnames"  
   value="/WEB-INF/validator-rules.xml,/WEB-INF/validation.xml"/>  
</plug-in>

1. **DispatchAction** class groups a **set of related functions into a single action**, eliminating need to create actions for each function. It selects a method to execute depending on request parameter value configured in the XML file
2. **LookupDispatchAction** class subclasses DispatchAction that dispatches to subclass mapped to execute method. This is **useful with form having multiple submit buttons with same name**. It looks into resource bundle file to map key name to method name
3. DynaActionForm is a subclass of ActionForm allowing creation of form beans with dynamic sets of properties, making form bean declarative in struts-config.XML
4. Struts uses ApplicationResources.properties and struts-config.xml configuration files between controller and model
5. Struts **work flow**
   1. Client request comes from **JSP to action servlet**, that **checks from struts-config.xml** for **form bean related to that JSP** and delegates work to validate method of that form bean
   2. The form bean **performs validations** and **returns to action servlet**. In case of Action Errors, action servlet returns to original JSP. In case of no errors, action servlet passes control to appropriate request class
   3. The Action class performs necessary business logic and forwards to next appropriate action (another JSP). In practice Action class calls a method on a business service class that performs business logic
6. Struts XML contains **mappings of input URL to action classes** defined similar to servlet definition in web.xml. By default **execute() method** **of action class** gets executed to serve request. Struts XML include a DOCTYPE and struts\_X.X.dtd definition. XML will have a **starting <struts> tag**, within which mapping is done inside <package> tag. <action> tag contains name of request in browser, while class represent actual JAVA class representing action. Action class may need to show different JSPs based on result of method execution. Execute method returns a **success/failure code**, which is then mapped to a JSP.

*<struts>*

*<package name=”default” extends= “struts-default”>*

*<action name=”getTutorial” class=”TutorialAction”>*

*<result name=”success”>/success.jsp</result>*

*<result name=”failure”>/failure.jsp</result>*

*</action>*

*</package>*

*</struts>*

1. To enable struts, **struts filter is added to web.xml** such that all requests now pass through this filter before hitting any servlet, as in snippet below.

*<filter>*

*<filter-name>struts2</filter-name>*

*<filter-class> org.apache.struts2.dispatcher.filter.StrutsPrepareAndExecuteFilter*

*</filter-class>*

*</filter>*

*<filter-mapping>*

*<filter-name>struts2</filter-name>*

*<url-pattern>/\*</url-pattern>*

*</filter-mapping>*

1. **Jars required in JAVA App using struts**: Struts-core.jar, ognl.jar, xwork-core.jar, commons-fileupload, commons-logging, commons-logging-api, commons-io, freemarker, java, javaassist, commons-lang. In order to make struts application work**, struts libraries should also be available in web app class path**. This is done in eclipse through "deployment assembly" option under project properties
2. In order to **transfer values from Action class to JSP**, struts-tag library is used within JSP file mapped to Action class. **<s:property> tags** are used within to access Action class member variables

<%@ taglib prefix="s" uri="/struts-tags" %>

<s:property value=”bestTutorialSite” />

1. Each **request to a servlet is in a separate thread** and having member objects store business service data can cause race condition. Business service data is stored in **session objects & request objects** associated with each client request which is accessible in JSPs through tags. Request object is used for data that is to be simply passed on to JSP and may not be used again. To store data for entire session i.e. user login info, request object is not suitable, and such data is saved in user session
2. With Struts there is a **separate instance of Action class for each request**, so a separate request object is not needed. Action object behaves as a model object. Framework saves **action object on to value stack**. JSP has a struts tag that accesses Value stack to get action data. Value stack behaves as a virtual object, isolating all member variables from action objects it stores
3. Passing parameters to Action class: parameter passed in URL is accessible as member variable in Action class. **Parameter transfer is done through interceptors** that set member value of Action class through setter Method. Interceptors utilize chain of responsibility pattern
4. **Struts form** is simpler to represent than Html form using tags from struts-tags library

*<s:form action="tutorials/getTutorial">*

*<s:textfield key="language" />*

*<s:submit/>*

*<s:form />*

1. Best Practices while designing forms using struts
   1. **Hardcoded strings** can be eliminated through use of **Action interface** provided by Struts framework. It contains definitions of common strings i.e. Success, Failure etc. Action class can implement this interface though not mandatory
   2. Struts allow **splitting struts.xml into multiple XML files** for larger application. Other XML files can be included in struts.xml through include directive
   3. **Use interceptors** for common tasks across application like logging. For a custom interceptor to be used in multiple actions, create interceptor stack for that
   4. Instead of directly mapping URL in XML files based on success or failure, have an **alias mapping to allow flexibility for changing URL name**. This can be done by a Dummy Action that redirects to actual URL

<package name=”search” namespace=”/” extends= “struts-default”>

<action name=”searchForm”>

<result>/searchForm.jsp</result>

</action>

</package>

1. Wildcards allow **single action mapping with multiple action names**. In case of conflict struts checks mapping from top to bottom, and first matched action is executed. **More specific actions should be specified first**. Also whatever matches in wildcard can be substituted in mapped URL. Number indicates a wildcard, for multiple wildcards consecutive numbers will represent them from left to right. Wildcard substitution can also work in similar way for Action classes

<package name=”search” namespace=”/” extends= “struts-default”>

<action name=”searchForm”>

<result>/searchForm.jsp</result>

</action>

<action name=”search\*”>

<result>/search{1}.jsp</result>

</action>

</package>

1. When **integrating spring with struts** operator "new" is not used to create an object. Rather use spring application context to get bean which is auto injected

ApplicationContext context = new ClassXmlApplicationContext(“/WEBINF/spring-config.xml”);

CustomerService service = context.getBean(CustomerService.class);

1. Spring provides a **listener class ContextLoaderListener**, which can be configured in web.xml. ContextLoaderListener will be called by tomcat container at time of deployment and un-deployment of application. Parameters to this listener are specified within <context-param> in web.xml. Using this approach avoids creation of ApplicationContext every time to get a Bean, rather it can be accessed through servletcontext

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/struts-config.xml</param-value>

</context-param>

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

<listener>

1. **Copy dependencies plugin of maven:** at time of package it will gather all dependencies & their dependencies in a specified output directory

<execution>

<id>copy-dependencies</id>

<phase>package</phase>

<goals>

<goal> copy-dependencies</goal>

</goals>

<configuration>

<outputDirectory>[ output directory ]</outputDirectory>

<overWriteReleases>false</ overWriteReleases>

<overWriteSnapshots>false</ overWriteSnapshots>

<overWriteIfViewer>true</overWriteIfViewer>

</configuration>

</execution>

1. Important **configurable struts constants** are
   1. **struts.devMode**: useful when developing application, provides extra logging info and debugging
   2. **struts.convention.result.path**: configures location of result pages from default at {WEBAPP-ROOT}/{Namespace}/
   3. **struts.action.extension**: configures URL suffix, changing name from **default .action**
   4. **struts.custom.i18n.resources**: global resource bundle for i18n support.

constant name="struts.devMode" value="true"></constant>

<constant name="struts.action.extension" value="action,do"></constant>

<constant name="struts.custom.i18n.resources" value="global"></constant>

<constant name="struts.convention.result.path" value="/"></constant>

1. Interceptor for i18n support is com.opensymphony.xwork2.interceptor.I18nInterceptor configured in struts-default package with name “i18n”
2. Struts classes can implement **Action interface as well as ActionSupport class**. Action interface provides constants i.e. SUCCESS, ERROR, LOGIN etc. ActionSupport class **additionally implements** Action, Validateable, ValidationAware, TextProvider and LocaleProvider interfaces. It enables **overriding validate() method** to include field level validation in action classes
3. Struts2 API provides **Aware interfaces** that can be implemented to **access Servlet API components** such as Request, Response and Session. Important Aware interfaces are SessionAware, ApplicationAware, ServletRequestAware and ServletResponseAware
4. Struts2 provides **execAndWait interceptor** for long running action classes that returns an intermediate response page to the client and after processing is finished final response is returned
5. **Double form submission** problem is handled through **token interceptor** defined in struts-default package
6. **log4j API** can be included in struts2 through log4j configuration file in the WEB-INF/classes directory.
7. Struts2 tags can be categorized as **Data tags, Control tags and UI tags**. Data tags include property, set, push, bean, action, include, i18n and text tag. Control tags are used for manipulation/navigation of data from a collection. They include if-elseif-else, iterator, append, merge, sort, subset and generator tag. UI tags are used to **generate HTML markup language**, binding HTML form data to action classes properties, type conversion, validation and i18n support. **Important UI tags** are textfield, password, textarea, checkbox, select, radio and submit tag
8. **Custom Type Converter class** in Struts2 can be created by extending **OGNL expression language**. Interface com.opensymphony.xwork2.conversion should be implemented by Type converter classes. StrutsTypeConverter is provided by Struts 2 API as default implementation of TypeConverter interface
9. **Interceptor life cycle** includes calls to init(), destroy() and intercept() methods. intercept() is the method called every time client request comes through the interceptor. Interceptors are Singleton classes and Struts2 initialize a new thread to handle each request. An interceptor stack helps grouping together multiple interceptors in a package. struts-default package creates basicStack and defaultStack. **Custom interceptor** is implemented using com.opensymphony.xwork2.interceptor.Interceptor interface and then defined in struts.xml package

<package name="user" namespace="/" extends="struts-default">

<interceptors>

<interceptor name="authentication"

class="com.journaldev.struts2.interceptors.AuthenticationInterceptor">

</interceptor>

<interceptor-stack name="authStack">

<interceptor-ref name="authentication"></interceptor-ref>

<interceptor-ref name="defaultStack"></interceptor-ref>

</interceptor-stack>

</interceptors>

<default-interceptor-ref name="authStack"></default-interceptor-ref>

</package>

1. Struts2 provides built in support for **file upload through fileUpload** Interceptor providing options to set the maximum size of a file and file types that can be uploaded

<struts>

<constant name="struts.multipart.maxSize" value="104857600" />

<package name="user" namespace="/" extends="struts-default">

<action name="upload">

<result>/UploadFile.jsp</result>

</action>

<action name="UploadFile" class=**"com.journaldev.struts2.actions.UploadFileAction"**>

<param name="filesPath">myfiles</param>

<result name="success">/UploadFileSuccess.jsp</result>

<result name="input">/UploadFile.jsp</result>

<interceptor-ref name="defaultStack">

<param name="**fileUpload.maximumSize**">10485760</param>

<param name="**fileUpload.allowedTypes**">text/plain,image/jpeg</param>

</interceptor-ref>

</action>

</package>

</struts>

1. Struts2 provides exception handling through **mapping specific exceptions** **to result pages** at the global package level as well as action level.

<package name="user" namespace="/" extends="struts-default">

<global-results>

<result name="exception">/exception.jsp</result>

<result name="runtime\_exception">/runtime\_exception.jsp</result>

<result name="error">/error.jsp</result>

</global-results>

1. **ModelDriven Interface** implementation by an Action class gives ability to transfer the form data into the object automatically. Struts exposes member variables of action class to value stack. Action class should **implement ModelDriven Interface** totell struts that action class has a model that struts should populate. Struts also needs to be told of the model object to populate and this is done through implementing method **getModel()** of ModelDriven interface

*Public class LoginAction extends ActionSupport extends ModelDriven <User> {*

*private User user = new User();*

*…..*

*public User getModel(){*

*return user;*

*}*

*}*

1. **ActionInvocation interface** represents the execution **state of an action**. It holds the action and interceptors objects, can be used to get the instance of ValueStack, ActionProxy, ActionContext, Result etc
2. Struts 2 provide an **alternative to using XML** to configure application by using standard naming conventions and annotations. Instead of defining Actions in struts.xml, Action classes can well be annotated with equivalent annotations

@Results({

@Result(name="success", location="/success.jsp"),

@Result(name="input", location="/index.jsp")

})

public class Employee extends ActionSupport{

private String name;

private int age;

@Action(value="/empinfo")

public String execute()

{

return SUCCESS;

}

@RequiredFieldValidator( message = "The name is required" )

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

@IntRangeFieldValidator(message = "Age must be in between 28 and 65",

min = "29", max = "65")

public int getAge() {

return age;

}

public void setAge(int age) {

this.age = age;

}

}

1. **Object-Graph Navigation Language (OGNL)** is a powerful expression language used to **retrieve objects on Value Stack** through referencing in JSP. It helps in data type conversion i.e. converting parameters coming in form of strings in an HTTP request to java types in java bean. It also converts java type to string when response is sent. OGNL offers expression language to access values inside a bean, and can also invoke methods on java classes through expression language. Struts 2 automatically stores **action objects onto value stack** that acts as a container of objects. It internally has an **objects stack & a context map** (contains various maps like parameters, requests, sessions, app maps etc). OGNL searches object map for a property by default. To search a context map # should be prefixed to search expression

*<s:property value="[0].name"/>*

*<s:property value="name"/>*

The above expressions are identical, trying to fetch name **property from top object on value stack**. If property is non-existent it keeps searching next objects till property is found or end of value stack is reached. To start searching a property from top object index need not be specified

*<s:property value="#request.name"/>*

*<s:property value="#request.user.name"/>*

Searches for name property from request map & from user object on request map in context map of value stack

*<s:property value="#parameters.count [0]"/>*

Retrieves value of first parameter on parameters map in context map of value stack

**Kerberos Interoperability**

1. **Kerberos deployment** is referred to as **realm** and is equivalent to **windows domain.** Realm comprises of KDC with applications/services using Kerberos. End users belong to a specific Kerberos realm and authenticate to that realm. Applications/services using Kerberos protocol to authenticate to network are **Kerberized Applications, having a** **credentials cache** for storing authentication info. User’s Kerberos credentials are managed through utilities – kinit, kdestroy, klist, kpasswd
2. **Credentials cache** is used **by application to store tickets** for accessing network resources after authenticating. **KDC network service** accepts requests for tickets from clients and grants tickets to them. **Kerberos implementation on distributed network** may have a seriesof slave servers to provide authentication services, which also provide KDC service having recent copy of **Key Database (KDB).** Changes to KDB are done at master KDC using kadmin utility. Kprop utility synchronizes master KDB with replicas
3. Kerberos implementation approaches and scenarios
   1. Windows client authentication to windows KDC : **Native approach** to access Windows resources, **One way trust or service account approach** for non-windows resources
   2. Windows client authentication to Non windows KDC: **Two way trust** for Windows resources, **Client configuration** for non-Windows resources
   3. Non Windows client authentication to Non windows KDC: **Two way trust** for Windows resources, **Native approach** to non-Windows resources
   4. Non Windows client authentication to windows KDC: **Client configuration** for Windows resources, **One way trust or service account** for Windows resources
4. **Native approach** involves matched Kerberos technology, with **no interoperability challenge and additional configuration**/implementation for Kerberos functionality. A user when added to Active directory on windows automatically uses Kerberos for network authentication
5. With **One way trust approach** between domain and realm, tickets generated in one realm will be recognized by resources in other. If a Kerberos realm on a UNIX server trusts Windows 2000 domain, it would allow windows users to log on to the domain. One way trust is important with **Windows laptop users wanting to access** **Windows print and file sharing services along with Kerberized services on UNIX servers**. Windows client locates its domain and authenticates to its KDC that issues a TGT to client. A **trust configured between windows domain and Kerberos realm** allows authenticated clients to access Kerberized resources on unix. Resources in Kerberos realm will decide access rights to resources based on authorization methods implemented for those resources. Alternatively **a service account can be created for UNIX based service in windows domain**, enabling UNIX service to be accessed in same way as other windows based services.
6. With **Two way trust** additionally users logged into Kerberos reaslm can access resources in Windows 2000 domain
7. An **Active directory service account** can be used to support Non windows Kerberized services in Windows domain. A UNIX based service is made accessible to Kerberos client in windows domain through service account
8. **Implementation for Automobile manufacturer**: Auto Company has deployed Windows platform for management of partner accounts. Its order processing application for is UNIX based, which is accessed by dealers and employees. KInit utility in UNIX Kerberos client is configured to request tickets from windows domain as its primary KDC. On Kerberos login to windows domain, TGT is returned and stored in credentials cache of Unix client. Also Windows KDC will provide authentication credentials to resources in Kerberos realm on UNIX
9. **Implementation for Insurance Company:** Agents use Windows laptops and need to connect to corporate networks having database applications on Unix server. **Windows clients need to be supported in non-windows Kerberos environment** without introducing windows domain. Windows based Kerberos **client can be configured to request TGT/resource tickets from non-windows KDC** and store in its credential cache, after which client presents credentials to Kerberized application on UNIX. Mapping is required between local users on windows to user account on Kerberos realm. It can be **many to one** where multiple windows users can be **mapped to a guest user** on UNIX or **One to one** where resource access gets controlled based on local user credentials
10. **Implementation for Brokerage house:** TheCompany has provided windows server and professional machines for their analysts to take advantage on windows features i.e. print/file services and also access Unix based resources. Analysts authenticate on non-windows Kerberos realm. Windows client need to **be configured to log on to** Kerberos realm. **One way trust is set up** between Windows domain and Kerberos realm as account domain i.e. Windows domain trusts Kerberos realm. Each **Kerberos principal** must have corresponding **windows proxy account**. Analyst **logs on to** **Kerberos realm** through Kerberos principal name and get tickets to access resources on realm. To access windows domain resource **logged in user is mapped to proxy windows domain account** and issued ticket from windows KDC. Synchronization is required between account info on Kerberos realm and windows domain
11. **Implementation for Brokerage house:** In above setup traders using Unix workstations require access to Unix based trading application as well as application services on windows server. **One way trust is set up** between Windows domain and Kerberos realm and each Unix account must have a mapped windows domain account.
12. To synchronize **active directory with Kerberos realm database** AD supports ADSI (AD service interfaces) & LDAP C-API used on UNIX platform. Basic sync routines to add/modify/delete user should be implemented. **Account mapping** between Kerberos realm and windows domain is contained in **AltSecuritId property**, One to One mapping allows maximum flexibility in administration. Sync choices include modification of KProp, or modification of KAdmin.
13. **KProp service** can be modified to contact **windows based Kpropd service** that runs as a trusted member of kprop service on Kerberos realm that is master source for user accounts. This requires **parsing KDB dump format** to create/update accounts and delete those not present in current dump. When kpropd receives an update, changes are applied to active directory through secure update package
14. Sync can be externally implemented as a batch script through **Kadmin listprinc service** that lists principals in KDB. It dumps current list of principals to a file, compare with previous sync, updating if required. **To automate procedure**, create a keytab file for an authorized principal and then kinit using this file. List and sort principals through Unix sort command. Comparison file contains users, hosts & services, only users need to be kept. **Secure update package** then needs to be called to update to active directory

**WinDbg tool**

1. Windbg is more powerful than VS2010, though later versions of VS have integrated features of WinDbg. It is a **light weight tool, can be copied to prod environment**. Provides dump analysis in user mode & kernel mode
2. An application can be started in WinDbg or it can be attached to running app. It supports kernel mode debugging using 2 machines, target also being VM

**Team Foundation Server (TFS)**

1. TFS enables all team members having different roles to **collaborate**, i.e. a CEO can connect through browser to check project status and reports. PM can connect to TFS through excel, developer can use VSTS while tester can use MTM
2. TFS is a central server product, a **web app installed on IIS**, uses SQL server reporting services. Any data team members push get stored in SQL server. **SharePoint** enables TFS access through browser. Share point & reporting services are part of advanced TFS installation

**Visual Studio Online**

1. VS Online is **a platform to plan, build, deploy, and manage apps** using services available to entire team. Comprises of other services also on top of TFS. Enables lightweight browser based code editing, automated build and application insights that give info on app availability, performance, traffic and other questions
2. Its online interface facilitates **creating work items, queries, charts and adding them to project dashboard**. Project user stories & work items can be added through backlog tool.
3. Its **integration with excel** enables querying TFS sever and fetching information i.e. user stories matching a criteria and also updating respective details on TFS server

**JIRA**

1. JIRA project is collection of issues. A Project is created with Name and Key, and key is appended to each project issue. Issues are grouped within components that are sub sections of project (UI, DB, Server, Bug etc.). Jira workflow is set of status and transitions an issue goes through in its life cycle

**J2EE Features**

1. J2EE technology includes **components and services**. Components are **servlets, JSP, EJB.** Services include **JMS, JTS, JAAS, JNDI, Java Mail**. J2ee is used for web apps **(WAR) & enterprise apps (EAR)**. EARs contain EJBs that are not in WAR files. EJB3.X adaptability is increasing as it is lightweight w.r.t previous versions
2. J2EE components are **functional units** assembled into a J2EE application. J2EE specifications include frontend which are **Application clients, applets or web clients**, Servlets and JSP which are **web components**, enterprise beans which are **business components.** It also includes **Resource adapter components** provided by Enterprise Information System (EIS) and tool vendors enabling connection to data store.
3. **A J2EE web module** can contain JSP files, Servlet classes, images, html files & deployment descriptor **all packaged as a WAR** while an **EJB modules** containing enterprise java beans class files and EJB deployment descriptor are **packed as JAR files** with .jar extension. **Both these modules are packaged in an EAR** file to be deployed on Application Server. EAR files also additionally contains **Application client Module** i.e. class files and an application client deployment descriptor and **Resource adapter modules** i.e. Java interfaces, classes, native libraries, documentation, resource adapter deployment descriptor
4. Container Service is **interface between component and platform specific support** for component. Before execution, components need to be assembled in a J2EE module and deployed in a container with configured settings ie security transaction mgmt, JNDI lookups & remote connectivity, allowing J2ee components behave differently within same J2ee application. It also provides **Management of non-configurable services** i.e. EJB/Servlet life cycles, DB resource pooling, platform API access and Container managed data persistence that can be overridden in EJB implementation. **J2EE containers** types comprise **EJB container** managing EJB execution, **Web container** managing JSP and servlet components, **App client container** managing App client components on client system & **Applet container** managing applet execution in web browser
5. **J2ee XML APIs** **translates application data** from XML data stream, **sent over XML transport protocols**, such as SOAP. XML Application formats include WSDL for describing network services, UDDI/ebXML used by businesses for publishing information on net about products/web services etc.
6. J2EE provides **role based access control (RBAC)**, for instance a user role or a manager role can have access to certain functions on application, with **access control being defined in web.xml** providing xml declaration on **what resources to protect an how to protect**, most common way being form authentication.
7. **J2EE application debugging** support
   1. Server Log **located at <J2EE\_HOME>/domains/domain1/logs/server.log** contains output from app server & applications
   2. Java Platform Debugger Architecture **(JPDA)** enables configuring Application Server to communicate debugging information via a socket
8. **J2EE API Collection**
   1. **EJB** **technology** handles DB access, no direct call to JDBC required
   2. **Java Servlet** Technology lets define HTTP based servlet classes
   3. **JSP** Technology allows servlet code in text/html based files
   4. **JMS API** provides message standard for J2ee components to send, receive messages
   5. **Java Transaction API** provides interface for demarcating transactions
   6. **Java Mail API** provides interface for email notifications
   7. **Java beans Activation Framework** is used by Java beans to determine type of data, discover operations available on it
   8. **Java API for XML** supports XML processing using DOM/SAXP/XSLT
   9. **Java API for XML base RPC** allows clients to make XML based RPC over internet, using SOAP and HTTP
   10. SOAP with Attachments API for Java **(SAAJ)** is low level API on which JAX RPC depends
   11. **Java API for XML Registries** provides access to business and general-purpose registries over the web
   12. **J2EE Connector Architecture** enables creating **resource adapters** for access to EIS that can be plugged in to any J2EE product. Allows interacting with underlying resource manager in EIS. Different resource adapter for each DB
   13. **JDBC API** enables invoking SQL commands from Java programming language methods
   14. **Java Naming and Directory Interface (JNDI)** provide naming and directory functionality and methods for performing standard directory operations.
   15. Java Authentication and Authorization Service **(JAAS)** enables application to authenticate/authorize user groups
   16. JSP Tag Library (JSTL) contains **core functions common to many JSP apps**, iterator and conditional tags for flow control, XML document tags, internationalization tags, DB access tags
   17. **Java Server Faces** is a UI Framework for web apps i.e. GUI component framework, model for rendering components, render kit

**J2EE Application Design considerations**

1. **Session State on client** can be maintained as Hidden form fields, browser cookies or directly embedding session in URI. Data is stringified in hidden forms & browser cookies. This results in Performance issues when data is large. Session data can be retrieved from server using **unique session ID**, that results in improved performance and no size constraint
2. **Session data persists on server** till specified timeout or manual session invalidation or externally removing state from server. Web email servers use similar state managementMethods. Requires **a** **load balancing strategy** in case of multiple servers
3. **Client Access Control** enables hiding portions of JSP view using tag library attributes i.e. guarded part can be added under **section <corePatterns:guard>**
4. View Guard Strategies while design of JSP views
   1. **all or nothing guard** where logic is implemented in a centralized control, preventing access from unauthorized users
   2. **Guarding a view portion** such as based on user role, a portion of view can be hidden. eg Employee's internal rating on a view can be visible to manager but not to employee
   3. **Portions of view not displayed on internal system state** i.e. based on system environment, display layout modified etc. eg Single processor CPU may not be required to show Multi CPU view details
5. **Guarding by configuration** uses inbuilt security mechanism in web container to guard a page, security constraints defined in web.xml. **<Auth-constraint> attribute specified in web.xml** such that resource is accessible only in a specified role
6. **Handling for Duplicate form submission** through **use of synchronizer token with session** & submission to prevent acceptance of another submission of same form by client. Use of **browser Back/Front buttons** can make this happen accidentally
7. **Client side Validation is limited in scope**, such as checks like form field is empty or semantically compliant. Uses embedded Java script within client view
8. **Server Side Validation** is form centric that uses java component on server to semantically validate input form field
9. **Java Bean Helper Classes** are used to hold intermediate state when passed with client request. **JSP engine auto copies parameters** passed to Javabean helper properties

<jsp:setProperty name="Helper Javabean Name", property="\*"/> //copies input parameters to respective properties

1. Bad Design Practices for presentation tier
   1. **Include custom tag helper at top of JSP view** for access control leading to difficulty in maintainability with multiple JSP files. Use **"View Helper" or "Front controller"**
   2. **Use of Presentation tier structures in Business tier for parameter passing**. This would require non web clients would need to wrap data to HttpServletRequest structure, increase tier coupling. Same is case in exposing presentation tier structures to domain objects Copying input information to individual parameters may have small performance impact but better maintenance
2. **Stateless beans** for Scalability can be disadvantage, as intermediate state in conversational process may need to be fetched from DB or client on network, reducing performance
3. Primary Key representation for composite key requires a class to be specified as a primary key. It should **implement overloaded "equals()" and "hashcode()"** methods
4. **EJB container Service** for business tier provides infrastructure for scalability, security, performance, clustering etc.
5. Entity beans Services provide object view of persistent data. They are transactional & survive container crashes
6. **J2EE design considerations** in summary include use of Design Patterns (Template, Strategy, Value Based), Checked Exceptions, Logging framework, Scoping of variables, Final declaration, Use of interfaces for loose coupling
7. **J2EE Application design** should include below summarized criteria
   1. Controlling view access to a client based on role, section access & entire access
   2. Handling Duplicate form submission
   3. Centralized validation approach
   4. Sharing presentation tier data structures with business logic & domain objects
   5. Handle Bean Interactions & issues with fine grained entity & session beans
   6. Data access encapsulation using DAO, and use of connection pool
   7. Move data access logic to **integration tier**
   8. Use of **session beans** for **inter entity bean relationship**
   9. Handling **user exceptions** from **session beans**
   10. Use of controller in JSP v/s in Servlet
   11. Handling of display processes in JSP, Servlet etc.
8. **Bad Design Practices for business tier**
   1. Mapping object model classes to entity beans, leading to **fine grained beans and many inter entity bean relationships**, impacting performance. Use **composite entity pattern** to model parent child relationships as coarse objects
   2. Mapping each **row in DB table to entity bean**, leading to fine grained entity beans, impacting performance. Design EJB application using object oriented model instead of DB
   3. Mapping each use case to a session bean, leading to large no. of session beans & increasing complexity. Use of **session facade to group related bean interaction to a single session bean**
   4. Expose all bean attributes with getter/setter methods, leading to numerous remote invocations for a service. **Use value object to transfer data to/from client to minimize remote calls**
   5. Add **Service location burden on client**, leading to including lot of lookup code in application code, increasing complexity. **Encapsulate service lookup details using service locator**
   6. Use of **entity beans for read only objects**, leading to performance overhead as EJB container still invokes ejbStore() in flow. **Use DAO pattern** to encapsulate data access from DB
   7. **Store entire tree of entity bean dependent objects**, leading to performance degradation. Store only **changed objects to persistent store** since last update, through Store optimization strategy
   8. **Exposing EJB exceptions to non EJB clients**, as client can handle application exceptions but may not know about EJB exceptions such as java.rmi.RemoteException. Use **Business delegate to encapsulate service exceptions** and throw and application exception
   9. Using **entity bean finder methods to return large results set**, leading to excess remote calls. **Use transfer object** instead of remote reference
   10. **Client aggregating data from business components**, leading to network overhead. Decouple client from business tier components, **use "transfer object assembler"**
   11. **Using enterprise beans for long transaction,** blocking container resources, threads etc. Use **service activator** for long transactions
   12. **Reconstructing conversational state through stateless beans**, causing performance degrade. Use "session facade" or **"transfer object assembler"**
9. Application Development roles
   1. **Product Provider** include vendors providing J2EE platform APIs, OS, Database, App Server, web server
   2. **Tool Provider** creates development, assembly, packaging tools
   3. **Application Component Provider** is a company creating web components, EJBs etc. for use in J2ee application
   4. **Enterprise Bean Developer** writes bean source code, deployment descriptor, packages class files in EJB JAR
   5. **Web Component Developer** writes servlets, JSP, HTML, deployment descriptor and packages
   6. Application Client Developer
   7. **Application Assembler** receives app modules from component providers and assembles in EAR file, specifies deployment descriptor for application
   8. **Application Deployer and Administrator** configures and deploys J2EE application, oversees runtime environment details i.e. security settings, transaction controls, connection to databases, configures operational environment
10. **Infrastructure considerations** for J2EE Application
    1. Choice of application server: application portability across app servers influences design. Websphere & JBoss commonly used
    2. Support for clustering : on demand if performance requirement exceeds capability of single server in real time
    3. Database selection: it is least portable stage, database specific optimizations i.e. stored procedures can be incorporated
    4. Physical location of system: based on whether DB server/ App server are on Same or different machines or on network
    5. Use of EJB, XSLT, JSP, etc as implementation
    6. If Business objects are doing more work than web tier, there is a case for running more EJB containers

**EJB**

1. **Enterprise Java beans** are J2EE components **managed by Application server Container** and encapsulate business logic forming back end of Enterprise Java application. It **uses core services of EJB container** e.g. transaction management, persistence, security etc. Main categories of EJB are session beans representing business logic, Message driven bean for workflow logic & JPA entity (or entity bean formerly) for persistence logic. EJB can be accessed by client in Local access mode by default, or **remotely through remote client access mode** or through web services
2. **Benefit of using EJB over stored procedure** for business logic is avoiding need port SQL Stored procedure on change of database
3. **Two phase commit protocol** is used to coordinate a single transaction across multiple databases, for ensuring data integrity. In **prepare phase** database or resource indicates of its readiness to make changes. If all resources indicate of their readiness to commit, they are updated else they are rolled back to previous state
4. EJB offers easy transaction management & is generally useful for large enterprise systems, **EJB 3.X though has simplified creation** of enterprise java bean applications by allowing **annotations** instead of XML, and also has reduced dependency on EJB framework as in previous version. EJB 3.2 supports execution of **beans in embedded container**, which like normal containers provide common services like connection pooling, transaction management etc. From EJB 3.1 onwards, the framework transitioned towards conventional approach using annotations instead of configuration file as in case of spring, which most developers found easy to work with. Through **dependency injection induced in EJB3.0**, container automatically fetches dependencies on lines of spring framework. **EJB is described by metadata**, which tells container how to manage bean and is **provided through annotations** in Java code that **can be overridden in XML file** if needed such as changing deployment location
5. EJB instances can be managed either in **Container managed or bean managed mode,** which can be set through annotations such as **@RunAs or @RolesAllowed** for Security type beans or **Transactionmanagement(BEAN|CONTAINER)** for transaction type beans. Container doesn’t permit direct access to beans rather **access is controlled through a proxy**, on which container can apply constraints such as enhanced security. Proxy also allows container to scale as in it can create multiple EJBs with client being transparent to which EJB it is accessing.
6. Since EJB cannot be accessed directly, **client cannot invoke its constructor**, rather it has to tell container to do so such as in 2.0 clients used home interface create method to invoke bean. While in **EJB3.0 constructor is triggered** through actions such as JNDI lookup or DI, which itself uses JNDI lookup. EJB life cycle transitions from **"do not exist"** to **post construct stage** where secondary initialization is performed on bean, bringing it to **Ready state** where it can interact with other components, and then stays in memory till container removes it following an external event such as **service shutdown**. Prior to removal bean release any resources in pre removal stage. **DI can be used to inject** an EJB, a resource, PersistenceContext and EntityManager through respective annotations
7. **Bean Managed Persistence (BMP)** is handled by developer using ejbLoad(), ejbStore() methods exposed by entity beans. BMP is good for legacy data store, but not suitable for RDBMS as batch functionality support is missing. Query finder has problem with BMP in entity beans
8. EJB can be **accessed within a servlet** by declaring member on Bean class and annotating it with @EJB

*@EJB(lookup="java:global/SalutationEJBProj/SalutationBean")*

*private SalutationBeanRemote salutationBean;*

**Servlets**

1. A Servlet has a **Service method** that takes parameters as HTTPServletRequest & HTTPServletResponse. Inside this method, request type is checked, whether it is GET or POST and doGet() or doPost() Methods is invoked.
2. **Deployment info for servlet** is present in web.xml. **Load-on-Startup** parameter, if set to a positive values loads servlet during deployment to avoid time consuming one time initialization on first request to servlet
3. A **request is forwarded** to another resource or for including content from another resource through **RequestDispatcher**, which can be obtained from HTTPServletRequest

*RequestDispatcher rd=request.getRequestDispatcher("servlet2");*

*rd.forward(request, response);//method may be include or forward*

1. A **ServletConfig** is created by web container for each servlet & can be used to retrieve configuration information from web.xml file, representing a single servlet object. Servlet configuration can be used by an instance to access information such as shopping cart for a user
2. **ServletContext** is created by web container and is similar to ServletConfig, but is common across all servlets in system. It enables sharing of objects between servlets or JSP in entire application, for instance used in getting application session related information
3. **Context parameters** **are specified under <context-param> tag in web.xml** can be read by all servlets in application.

String myContextParam = request.getSession().getServletContext().getInitParameter("myParam");

1. Saving and retrieving **session info through Cookie**
   1. Servlet’s doPost() method gets called when Submit button is clicked
   2. Inputs posted by client are retrieved from Request
   3. At server side, both Request and Response objects passed as parameter to doGet() are processed
   4. Cookie object is created with a server generated key /value for received inputs, which is then added to response object
   5. Response object is further updtaed by setting its content type
   6. Browser receives response object, retrieves cookies from it and stores in cache along with server IP
   7. On next request to server by same browser, cookie matching server IP gets sent to server
   8. Server at its end gets cookie scans for the key it placed, if found retrieves data and sets in response object

**Java Server Pages**

1. **JSP is an extension of servlet**, comprising of HTML/JSP tags, and is easier to maintain than servlet. Apart from standard servlet features, it is possible to use **implicit objects, predefined tags, expressions, Custom tags** within JSP. No need to recompile and there is lesser code as most is substituted by tags
2. **JSP Life Cycle**:

JSP Page --> |Translator| -> Servlet file (.JAVA) -->|Compilation| -> Class file --> JRE --|Create| ->Servlet Object

1. Within JSP, **Scriptlet tag** (<%) enables processing java code

<body>

<% out.print("welcome to jsp"); %>

</body>

1. **JSP Implicit Objects** are created by web container and are available to all JSP pages. They are - out, request, response, config, application, session, pageContext, page, session. **Application object** corresponds to servletContext, **config** corresponds to servletConfig. **PageContext** is used to set, remove, get attributes from scope of page, session, application & request
2. **Exception handling in JSP** can be done by defining an error-page element in web.xml, and point it to an error.jsp file that gets loaded in case of exception. It is also possible through **scriptlet directive <%@ page errorPage="error.jsp" %>** on top of processing JSP page.
3. Jstl library is a collection of JSP tags for presenting content. It coprises of expression language, validators & tag libraries. Tag libs are of type Core, Formatting, SQL and XML. Expression language is used to access implicit and scoped objects. Tag library is declared using Tag lib directive

*<%@* ***taglib uri****="http://java.sun.com/jsp/jstl/core" prefix="c" %>*

*<c:set var="contextRoot" value="${****pageContext.request.contextPath****}" />*

*<a class="nav-link" href="****${contextRoot}/about****">About</a>*

1. Spring MVC also provides its own tag library to develop Spring JSP views, provided by **spring-webmvc** dependency in maven pom file. Tags provide useful common functionality when working with spring MVC

*<%@taglib prefix="form" uri="http://www.springframework.org/tags/form" %>*

*<%@taglib prefix="spring" uri="http://www.springframework.org/tag*

These **taglib directives** declare that JSP page uses custom tags related to Spring along with location of the library. **Uri attribute** value resolves to a location that the servlet container understands and **prefix attribute** informs which bits of markup are custom actions on JSP page.

1. JSP action tags are used to control flow between JSP pages and load JAVA beans

*jsp:forward, jsp:include are similar to corresponding servlet methods*

*jsp:useBean, set/get property working with bean objects*

*jsp:plugin embed other components in JSP i.e. applet*

*jsp:fallback provides alternate message if plugin not working*

1. A JSP file can be included in another JSP using include directive. Included file may not have a JSP header. Enables separation of a JSP file into shared modules among different JSPs

*<c:if test="${homeClicked == true}">*

***<%@include file="home.jsp"%>***

*</c:if>*

Conditional include can be setup based on a model property value which is received from MVC controller i.e. homeClicked property.

1. A quick way of **getting a landing page for a web application** is using a bootstrap template, copying its source in the JSP file returned by MVC. **CSS, JS and images** referred to by bootstrap template are added as resources inside a folder in **webapp folder**. This folder is then **mapped to resources** in spring dispatcher configuration file. All references to original css/js/images in bootstrap templat**e** are then replaced to point to **local resource directory**. Local resources variables are created using **spring tag library**.

**XML Parsers**

1. SAX and DOM handle xml parsing in different ways. **DOM parser makes an in memory** representation of XML doc. It is preferred for **smaller XML docs** or where parsing is required in **several different** ways. DOM constructs a tree of nodes. XML elements are **stored as objects** whose properties can be directly accessed. **Navigation is easier** as whole doc is in memory.
2. SAX parses XML **doc line by line** and is more memory efficient. It also **throws events** after parsing every line that can be captured by app.. SAX is preferred for **large docs and is faster at runtime** as it doesn’t require loading entire doc in memory. In SAX objects have to be created and **code writing is required for object creation**. Backward navigation is not possible.

**J2EE Project creation using Maven**

1. Maven Project creation through eclipse using Standard Maven Template creates a POM and Project Structure
2. Through UI, 3rd party dependencies are added to pom.xml. On Build, corresponding jars are downloaded from maven repositories to local disk (mvn local repository)
3. **Additional repositories can be provided under <repositories>** section in pom.xml. e.g for primefaces, maven needs to search through a configured repository as the JARs are not present under default repositories
4. It may be required to use a JAR not found in any repository, but still it is needed in maven local local repository e.g. ojdbc.jar for oracle database connection. **To install it into maven local repository** **use maven command on command line** in location where downloaded JAR is stored

mvn install:install-file -DgroupId=com.oracle -DartifactId=ojdbc14 -Dversion=10.2.0.1.0 -Dpackaging=jar -Dfile=ojdbc14.jar -DgeneratePom=true

1. A web project requires web.xml to be present under WEB-INF folder. Each element in xml belongs to a namespace, included in XML file through xmlns: tag

**Java Messaging**

1. JMS is a **Messaging service** that allows components to send and receive the messages. It supports **Point to point queuing** and **publish subscribe model**. JMS components include **JMS provider,** **JMS client**, **messages, Administered objects & native clients**. JMS provider handles data conversion, security of the messages and the client triggering, Administered object is a **pre-configured JMS object** for use of JMS clients and available in JNDI namespace. A JMS session is a **context for sending and receiving JMS messages.** It can be locally transacted, non-transacted or distributed transacted.
2. **Publishers** are sender apps that tag each message with a topic name and then sends it to pub sub server. **Subscribers** are receiviBootbootng apps that choose which topics to receive. Subs can subscribe to multiple topics. **Pub sub server** defines uniform message structure and directs messages
3. **JMS API** supports Message, Text Message, Bytes Message, Object Message and Map Message formats. **Java EE provides MDBs** that enable application clients to consume **messages asynchronously**
4. **An MOM** is placed between the client and server as an **intermediary between them** and provides store for messages until client requests to read message. A **Broker** is a universal translator of message formats between applications.
5. JMS API provides **vendor independent access to messaging systems** (JMS providers) i.e. Websphere MQ, Jboss messaging, bea weblogic etc. Supports both Point to point & publish subscribe topics
6. In Pub-Sub model, publishers **submit messages to classes, instead of users**. Users subscribing to a class receive published messages w/o knowledge of publisher. This offers less flexibility in modifying message structure, but good scalability. JMS Support **topic based & content based filtering**
7. **Multitasking** is useful when real time delivery of messages is preferred over reliability. It works with pub-sub model and only for **non-durable subscribers**
8. A **JMS Server** comprises of resources including queues, topics, connection factory and persistent storage. Resources can be looked up using JNDI
9. A **transaction in JMS** is a group of messages sent and received by application. On commit received message are removed from persistent store, while sent messages are delivered. JTA supports transaction over multiple resources i.e. jms, ejb. To use JTA user transaction in WL server, **XA enabled connection factory** must be configured
10. **Apache MQ is a message broker with full JMS support.** It has a web frontend for monitoring messages. Active MQ can run as standalone or embedded inside JUnit test. **KahaDB** is by default used as persistent storage. Protocols used by Apache MQ are **MQTT protocol**, Stomp & FuseMq. **MQTT** is very lightweight, designed for embedded devices, very small message payload. **Stomp** isa text oriented message protocol, making it easy to write a client to connect to stomp message broker. It is not as fast as binary and a natural protocol for web sockets, **FuseMq** has centralized distributed broker configuration through zookeeper
11. Message design considerations
    1. Avoid using user defined properties, instead **use standard JMS header fields**
    2. Application objects should implement **Externalizable**
    3. **Temporary destinations** exist only for duration of connection, and do not survive a JMS server restart
    4. For **durable subscriptions**, messages are persisted unless delivered or expired
    5. **QueueBrowser** object is used to browse messages in a queue
    6. Messages can be **filtered using message selectors** that are in form of SQL expression and can be applied to a subscription
    7. A message delivered as persistent will **always be stored by broker** to be delivered later to a consumer

**Spring Framework**

1. Spring framework is a platform for developing Java and web applications. It has **several IO projects** like Web services, integration with No-SQL, etc. Coreframework is based on **IOC and DI**. In IOC pattern framework calls application components as specified and **DI enables injecting dependency in another class** externally through XML configuration or annotations
2. Setting up a **spring project**
   1. Add spring dependencies to Maven POM file
   2. Create XML configuration file in /src/main/resources folder, with bean declarations
   3. Load beans through ApplicationContext
3. A **Spring Bean Factory** reads XML configuration file and returns a new **spring bean.** It allows optional initialization of bean members. An **Application Context** is similar to bean factory with more features..

*ApplicationContext context = new ClassPathXmlApplicationContext (“spring-config.xml”);*

*Customer cust = (Customer) context.getBean (“customer”);*

1. Setter injection is performed through setting bean properties in configuration file

*<beans>*

*<bean id="triangle" class="shapes.Triangle">*

*<property name= "type" value="Equilateral" />*

*<property name= "Border" value="15" />*

*</bean>*

*</beans>*

Sets member variables of Triangle bean.

1. With Constructor based DI spring tries to choose correct bean class constructor and attempts auto conversion of arguments. Hints **type and index** are used with overloaded constructors

*<bean id="triangle1" class="shapes.Triangle">*

*<constructor-arg value="Scalene" />*

*<constructor-arg value="10" />*

*</bean>*

*<bean id="triangle2" class="shapes.Triangle">*

*<constructor-arg value="Scalene-2" />*

*</bean>*

*<bean id="triangle3" class="shapes.Triangle">*

*<constructor-arg index="1" value="25" />*

*<constructor-arg index="0" value="Scalene - 3" />*

*</bean>*

*<bean id="triangle4" class="shapes.Triangle">*

*<constructor-arg type="int" value="20" />*

*<property name= "type" value="Equilateral" />*

*</bean>*

1. DI for dependent beans happens through **ref keyword** in bean properties or in constructor-args. Ref keyword can point to either bean ID or bean Name attribute

*<bean id="centerPoint" class="shapes.Point" name="test-point">*

*<constructor-arg index="1" value="20" />*

*<constructor-arg index="0" value="30" />*

*</bean>*

*<bean id="circle1" class="shapes.Circle" name="test-circle">*

*<property name="radius" value="10" />*

*<property name="center"* ***ref="centerPoint****" />*

*</bean>*

<bean id="endPoint" class="shapes.Point" name="test-point1">

<constructor-arg index="1" value="50" />

<constructor-arg index="0" value="80" />

</bean>

<bean id="line1" class="shapes.Line">

<constructor-arg index="2" value="15" />

<constructor-arg index="0" ref="centerPoint" />

<constructor-arg index="1" ref="test-point1" />

</bean>

1. A **collection of dependent beans** isinitialize through **a "list"** instead of value

*<bean id="curve1" class="shapes.Curve">*

*<property name="points">*

*<list>*

*<ref bean="centerPoint"></ref>*

*<ref bean="endPoint" />*

*</list>*

*</property>*

*</bean>*

1. **Auto wiring** performs automatic DI of dependent beans. Specified through attributes **"**byName" & "byType" in configuration file. Either bean Id or bean name in XML configuration file should match with name of field declared in bean class

*<bean id="line2" class="shapes.Line"* ***autowire="byName">***

*<property name="length" value="45" />*

*</bean>*

*<bean id="start-point" class="shapes.Point"* ***name="start"****>*

*<constructor-arg index="1" value="50" />*

*<constructor-arg index="0" value="40" />*

*</bean>*

*<****bean id="end****" class="shapes.Point">*

*<constructor-arg index="1" value="20" />*

*<constructor-arg index="0" value="30" />*

*</bean>*

*public class Line implements Shape*

*{*

*private Point start;*

*private Point end;*

*private int length;*

*}*

1. If both auto wiring and property initialization are used, property will take priority

<bean id="line2" class="shapes.Line" autowire="byName">

<property name="length" value="45" />

<property name= "start" ref="end" />

</bean>

1. Application Context is creates all beans when reading configuration file. Bean **scope is singleton** by default i.e. same bean instance is returned on calling getBean() at different places.

*Line l2 = (Line) appContext.getBean ("line2");*

*l2.draw ();*

*Line l3 = (Line) appContext.getBean ("line2");*

*L3.setLength (80);*

*l2.draw ();*

*l3.draw ();*

Bean l2 and l3 will point to same bean instance with singleton scope.

1. If scope is made **prototype,** new instance of bean is returned every time on getBean()

*<bean id="line2" class="shapes.Line" autowire="byName"* ***scope= "prototype"****></bean>*

1. **ApplicationContextAware interface** can optionally be implemented for a bean. It provides bean access to application context. **BeanNameAware interface** gives access to name of bean. Bean name is determined by bean **attributes “id” or “name”.**
2. **Bean definition inheritance**, allows inheriting common configuration among beans definitions through parent attribute.

*<bean id="start-line" class="shapes.Line" >*

*<property name="length" value="45" />*

*<property name= "start" ref="centerPoint" />*

*</bean>*

*<bean id="start1" class="shapes.Point" name="centerPoint">*

*<constructor-arg index="1" value="20" />*

*<constructor-arg index="0" value="30" />*

*</bean>*

*<bean id="line3" class="shapes.Line"* ***parent="start-line"*** *autowire="byName">*

*</bean>*

Configurations **“length” and “start”** get inherited through BDI with “end” gets added through auto wire

1. **Collections** can also be inherited through BDI using **merge attribute**.

*<bean id="curve-start" class="shapes.Curve">*

*<property name="points">*

*<list>*

*<ref bean="centerPoint"></ref>*

*</list>*

*</property>*

*</bean>*

*<bean id="curve2" class="shapes.Curve"* ***parent="curve-start">***

*<property name="points">*

*<list* ***merge="true****">*

*<ref bean="endPoint"></ref>*

*</list>*

*</property>*

*</bean>*

Curve2 will have both centerPoint and endPoint

1. **Lifecycle callback** methods for spring beans a supported through XML configuration at global level or bean level. **Init-method** gets executed at time of bean creation, differs for singleton & prototype beans**. Destroy-method** is only called for singleton beans after shutdown hook has been registered with application context

*AbstractApplicationContext context = new ClassPathXmlApplicationContext("spring-config.xml");*

*appContext.registerShutdownHook();*

<**bean** id=triangle **init-method="myInit" destroy-method="cleanup"**/>

At global level configuration can be made for all beans by specifying same attributes **in "beans" tag**

<**beans** default-init-method="myInit" default-destroy-method="cleanup"/>

1. **Bean Post Processor** class implements **BeanPostProcessor interface.** It contains code that runs on initialization of all beans.

*//***Register BeanPostProcessor in XML**

*<bean class="DisplayBeanPostProcessor" />*

**//Define BeanPostProcessor class**

*Public class DisplayBeanPostProcessor extends BeanPostProcessor{*

***//Bean and bean name get passed as arguments to interface methods***

*@Override*

*Public Object postProcessAfterInitialization(Object bean, String beanName)*

*{*

*///This gets called after bean initialization.*

*}*

*@Override*

*Public Object postProcessorBeforeInitialization(Object bean, String beanName)*

*{*

*///This gets called before bean initialization.*

*}*

*}*

1. Bean post processor is used to perform custom configuration before and after bean creation by container i.e. writing **custom annotations** and letting framework know on how to interpret them on beans
2. A BeanFactoryPostProcessor is invoked immediately on initialization of Spring application context.
3. **@Required** Annotationis a part ofsprings inbuilt bean post processor. It lets framework validate whether member object of a bean has been initialized.

*Public class Circle*

*{*

*Private Point center;*

***@Required***

*Public void setCenter(Point center)*

*{*

*this.center = center;*

*}*

*}*

**//Register in XML**

<bean class="org.springframework.beans.factory.Annotation.ReqiredAnnotationBeanPostProcessor" />

If **“center”** in not initialized for any bean of type Circle, exception will get thrown at initialization instead of run time.

1. **@Autowired** by defaultuses “by-type” for DI and then attempts DI with **"by-name”.** If then looks for a bean having **qualifier attribute and matching value.** Auto wiring is enabled through **AutowiredAnnotationBeanPostProcessor** of spring

*public class Sphere implements Shape*

*{*

*Point3D center;*

*int radius;*

***@Autowired***

***@Qualifier("also-sphere-center")***

*public void setCenter(Point3D center) {*

*this.center = center;*

*}*

*}*

*<bean id="point3d1" class="shapes.Point3D" name="sphere-center">*

*<property name="xCoord" value="50" />*

*<property name="yCoord" value="60" />*

*<property name="zCoord" value="90" />*

*</bean>*

*<bean id="point3d2" class="shapes.Point3D" name="cone-center">*

***<qualifier value="also-sphere-center" />***

*<property name="xCoord" value="30" />*

*<property name="yCoord" value="60" />*

*<property name="zCoord" value="40" />*

*</bean>*

Bean Sphere member “center” is auto wired to bean with name “cone-center” through qualifier attribute

All annotation bean post processors can be defined using a single directive. This requires adding context schema reference to XML configuration file

<context:annotation-config />

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

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

***xmlns:context="http://www.springframework.org/schema/context"***

*xsi:schemaLocation="*

*http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans.xsd*

***http://www.springframework.org/schema/context***

***http://www.springframework.org/schema/context/spring-context.xsd">***

1. JSR 250 **@Resource** annotation is alternative to @Autowired. @Resource first searches dependencies by name and then by type while @Autowired does opposite.

**@Resource(name="cone-center")**

public void setCenter(Point3D center) {

this.center = center;

}

<bean id="point3d2" class="shapes.Point3D" name="cone-center">

<qualifier value="also-sphere-center" />

<property name="xCoord" value="30" />

<property name="yCoord" value="60" />

<property name="zCoord" value="40" />

</bean>

@Resource will inject bean with name “cone-center” through name parameter while @Autowired will require a qualifier attribute for same injection

1. JSR 250 @PostConstruct and @PreDestroy work in same way as life cycle methods InitializeBean and DsiposingBean

***@PostConstruct***

*public void InitSphere(){*

*System.out.println("Sphere:: Initializing");*

*}*

***@PreDestroy***

*public void DestroySphere(){*

*System.out.println("Sphere:: Disposing");*

*}*

1. **@Component** defines a **java class as a spring bean**. Has same effect as defining bean in spring.xml. Bean gets same name as name of Bean class in lower case. To enable @Component, **"component-scan" directive** should be added to spring.xml. Other similar annotations are @Service, @Repository & @Controller. Bean class name should be unique across base package hierarchy.

*<context:component-scan* ***base-package="shapes****" />*

***@Component***

*public class Sphere implements Shape*

*{*

*Point3D center;*

*int radius;*

*@Autowired*

*@Qualifier("also-sphere-center")*

*public void setCenter(Point3D center) {*

*this.center = center;*

*}*

*}*

*Sphere s2 = (Sphere)****appContext.getBean("sphere")****;*

1. Steps for setting up **AOP with Spring**
   1. Add dependencies to Maven POM file i.e. spring-aop, aspect-rt, aspect-jweaver & cglib.
   2. Add AOP namespace to configuration file,
   3. Enable AOP through <aop:aspectj-autoproxy />
   4. Define an aspect class, annotate with @Aspect
   5. Annotate methods of this class i.e. advise methods
2. **AOP programming** allows addressing cross cutting concerns i.e. logging and transaction management.
3. Advise methods are executed around target method as per advise method annotation and point cut expression

*@Before("execution(public String getName())")*

*public void LoggingAdvise(){*

*System.out.println("In Transaction aspect advise");*

*}*

LoggingAdvise executes before any method matching specified signature is called

1. Join Point contains information about method that triggered advise

@Before("execution(public \* get\*())")

public void LoggingAdviseGeneral(JoinPoint jp)

{

System.out.println("In logging aspect general advise "+ **jp.getTarget());**

}

1. **@Advise** is applied on a method of aspect class, and inserted at a join point. Different advise types include @around, @before, @after. **Join point has information** about actual method called that triggered advise. An advise method can have a JP as an argument. JP refers to all places in code where advise can be applied.

*@Aspect*

*Public class LoggingAspect*

*{*

*@Before(“allCircleMethods()”)*

*Public void LoggingAdvise(JoinPoint jp)*

*{*

***//Gets handle of object whose method is called***

*System.out.println(jp.getTarget());*

*System.out.println(jp.toString());* ***//prints out method details***

*}*

*}*

1. AOP Point cut expressions with advise annotations

@Before ("**execution(public String getName())"**)

public void LoggingAdvise(){

System.out.println ("In logging aspect advise");

}

LoggingAdvise() will executed before any bean method having **signature getName()**

@Pointcut("**within(shapes.Point)**")

public void PointMethods(){

}

@Before("args(len) and !PointMethods()")

public void LoggingAdviseWithInt(int len)

{

System.out.println("LoggingAdviseWithInt:: " + len);

}

LoggingAdviseWithInt() will execute for all methods with “int” argument but will exclude methods of Point class

1. An advise can be applied to selective methods through custom advise annotations

*public @interface TestAnnotation {*

*}*

*@Around("@annotation(aspect.TestAnnotation)")*

*public void LoggingAroundAdviseLine(ProceedingJoinPoint jp)*

*{*

*}*

LoggingAroundAdviseLine() gets applied to all methods annotated with @TestAnnotation

public class Line

*{*

***@TestAnnotation***

*public void shiftStart(int xOffset, int yOffset){*

*startPoint.setxCoord (startPoint.getxCoord() + xOffset);*

*startPoint.setyCoord (startPoint.getyCoord() + yOffset);*

*System.out.println ("New Point after ShiftStart: " + startPoint);*

*}*

*}*

1. Messages can be configured to be read from a properties file through spring’s inbuilt class **ResourceBundleMessageSource.** It is defined as a bean inside spring.xml with attribute “id” or “name” as **“messageSource”.**

*<bean id="messagesource" class="org.springframework.context.support.ResourceBundleMessageSource">*

*<property name="basenames">*

*<list>*

*<value>messages</value>*

*</list>*

*</property>*

*</bean>*

1. **Localization of** messages is done through country specific properties file in same directory as base messages.properties file. Naming convention is country specific & locale should be specified

For US: messages\_en\_US.properties

For Germany: messages\_de.properties

Locale parameter is specified in getMessage()

appContext.getMessage("greeting", null, "This is default one", **Locale.GERMANY**)

1. Spring provides support for **event publishing, listening & writing**, through interfaces ApplicationListener, ApplicationEvent, ApplicationContext, ApplicationEventPublisherAware etc..
2. **JDBC data source** is configured as a spring bean with Driver, connection url and other properties. Data source can be replaced in XML without making code changes

<bean name="datasource1"

class="org.springframework.jdbc.datasource.DriverManagerDataSource">

<property name="driverClassName" value="org.apache.derby.jdbc.ClientDriver" />

<property name="url" value="jdbc:derby://localhost:1527/db;create=true" />

</bean>

<bean name="datasource2"

class="org.apache.commons.dbcp.BasicDataSource">

<property name="driverClassName" value="org.apache.derby.jdbc.ClientDriver" />

<property name="url" value="jdbc:derby://localhost:1527/db;create=true" />

<property name="initialSize" value="2"/>

<property name="maxActive" value="5"/>

</bean>

Connection URL is obtained by injecting data source

public class JdbcImplSpring

{

@Resource (name="datasource2")

DataSource datasource;

Connection conn = datasource.getConnection ();

}

1. Query preparation and execution is handled through JdbcTemplate, in which data source is injected. JdbcTemplate provides overloaded methods for execution of queries for argument types and return types

*@Resource (name="datasource2")*

*public void setDatasource(DataSource datasource)*

*{*

*this.datasource = datasource;*

*jdbcTemplate.setDataSource(datasource);*

*}*

Datasource is auto wired to JDBC template at initialization

*public int getCircleCount()*

*{*

*String sql = "select count(\*) from circle";*

*return jdbcTemplate.queryForInt(sql);*

*}*

*public String getCircleName(int id)*

*{*

*String sql = "select name from circle where id = ?";*

*return jdbcTemplate.queryForObject(sql, String.class, id);*

*}*

1. User defined Event extends ApplicationEvent class

*public class SphereDrawEvent extends ApplicationEvent*

*{*

*public SphereDrawEvent(Object source) {*

*super(source);*

*}*

*@Override*

*public String toString() {*

*return "Sphere draw event occurred";*

*}*

*}*

Event is published from an object method

public class Sphere implements Shape, ApplicationContextAware

{

Point3D center;

int radius;

ApplicationContext appContext;

public void draw() {

appContext.**publishEvent (new SphereDrawEvent (this));**

}

}

1. To use **hibernate with spring**, hibernate JARs are required in POM file i.e. hibernate-core & javassist. Session factory requires a data source, path to packages where hibernate entities are located & hibernate specific properties. It is then injected into a spring bean from where hibernate queries can be run

*<bean name="datasource1"*

*class="org.springframework.jdbc.datasource.DriverManagerDataSource">*

*<property name="driverClassName" value="org.apache.derby.jdbc.ClientDriver" />*

*<property name="url" value="jdbc:derby://localhost:1527/db;create=true" />*

*</bean>*

*<bean id="hibernateSessionFactory"*

*class="org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean">*

*<property name="dataSource" ref="datasource1" />*

*<property name="packagesToScan" value="dao.shapes" />*

*<property name="hibernateProperties">*

*<props>*

*<prop key="dialect">org.hibernate.dialect.DerbyDialect</prop>*

*</props>*

*</property>*

*</bean>*

*@Repository*

*public class HibernateDaoImplSpring*

*{*

***@Resource(name="hibernateSessionFactory")***

*SessionFactory sessionFactory;*

*public int getCircleCount(){*

*String hql = "Select count(\*) from Circle";*

*Query query = sessionFactory.openSession().createQuery(hql);*

*return ((Long)query.uniqueResult()).intValue();*

*}*

*}*

1. With Hibernate4 and Spring4 LocalSessionFactoryBean is used for session factory creation

**Hibernate**

1. Hibernate is a **persistence framework** performing ORM and querying databases through HQL and SQL.
2. Hibernate framework creates an **entity instance using Reflection API.** Hibernate Model classes are annotated with **@Entity.**  Class field annotated with **@Id is primary key** in DB. Table names and field names by default assume names of entity class and fields. In database these names **can be overridden** through annotations.
3. Hibernate converts **Java data types to SQL data types** while saving to DB and does reverse while retrieving data.
4. Core interfaces are **SessionFactory, Session, Configuration, Transaction, Query and Criteria**.

*UserDetails user = new UserDetails ();*

*SessionFactory sf = new Configuration ().configure ().buildSessionFactory ();*

*Session session = sf.openSession ();*

*session.beginTransaction ();*

*session.Save (user);*

*session.getTransaction ().commit ();*

*session.close();*

Object is retrieved from database through session.get() method that takes in entity class and primary key

*Session session = sf.OpenSession();*

*session.BeginTransaction ();*

***UserDetails user1 = (UserDetails)session.get(UserDetails.class, 1);***

1. Hibernate 5 uses **ServiceRegistryBuilder** to create session factory

*public class HibernateUtil*

*{*

*private static StandardServiceRegistry stdReg;*

*private static SessionFactory sf;*

*static*

*{*

*try*

*{*

*if(sf==null){*

*stdReg = new StandardServiceRegistryBuilder().configure().build(); MetadataSources mds = new MetadataSources(stdReg);*

*Metadata mdata = mds.getMetadataBuilder().build();*

*sf = mdata.buildSessionFactory();*

*}*

*}*

*catch (Exception e)*

*{*

*e.printStackTrace();*

*StandardServiceRegistryBuilder.destroy(stdReg);*

*}*

*}*

*}*

1. A Hibernate Configuration file i.e. **hibernate.cfg.xml** contains settings related to database It is placed in root src folder of project.

*<hibernate-configuration>*

*<session-factory>*

*<!—Database settings 🡪*

*<!—JDBC connection pool setting 🡪*

*<!—SQL Dialect🡪*

*<!—Second level cache if present🡪*

*<!—Echo SQL to stdout🡪*

*<!—Schema creation options at app startup🡪*

*<!—Names of model classes (annotated @entity) 🡪*

*</session-factory>*

*</hibernate-configuration>*

1. **hbm2ddl.auto** property in hibernatecfg.xml, when **set to "create"** drops and creates a new schema each time on application start up. To preserve current schema and existing data it should be **set to "update"**. It also adds **new columns** to existing tables if a new field is added to entity class. There are **no deletions to existing tables and columns** with update option. Changes to properties of exiting columns will only be applicable to new entries. Additional options in hibernate5 are **“validate”** and **“create-drop”.** Create-drop preserved database state at the end of hibernate application run.
2. Session methods Save () or Persist () are used to insert record into DB. Save() returns ID (primary key) of record inserted. Other methods are update() and saveorupdate();

Employee emp = CreateEmployee ("Vijay Diwevdi","v.diwedi@gmail.com",new Date (), 100010.5);

session.beginTransaction ();

**session.persist (emp);**

session.getTransaction ().commit ();

emp = CreateEmployee ("Rajesh Diwevdi","r.diwedi@gmail.com", new Date(), 100010.5);

**Integer id = (Integer)session.save(emp);**

System.out.println ("Employee Id = " + id);

session.getTransaction ().commit ();

**Employee emp1 = session.get(Employee.class, 1);**

emp1.setEmail ("diwedi@gmail.com");

emp1.setSalary (120500.0);

session.beginTransaction ();

**session.update(emp1);**

session.getTransaction ().commit();

1. Session.merge() merges properties of detached object in a new session. Update() does same but doesn’t work across sessions

*Employee2 emp1;*

*try (Session session = HibernateUtil.getSessionFactory().openSession())*

*{*

*emp1 = session.get(Employee2.class, 1);*

*System.out.println (emp1);*

*}*

*emp1.setName("Shankar");*

*try (Session session = HibernateUtil.getSessionFactory().openSession())*

*{*

*session.beginTransaction ();*

***session.merge(emp1);***

*session.getTransaction().commit();*

*}*

Detached object emp1 is updated in DB in a new session through merge ()

1. In a real application, a client interacts with entities through **service layer & DAO laye**r. DAO layer contains code for CRUD operations on entity. Service layer calls methods on DAO layer with additional business logic. Client program uses service layer for its interactions. Both service & dao layers are referenced through interfaces

*public interface EmployeeService*

*{*

*public void CreateEmployee(Employee emp);*

*public void UpdateEmployeeById(Integer Id, Double salary);*

*public Employee findEmployeeById(Integer Id);*

*public void DeleteEmployeeById(Integer Id);*

*public default void CleanUp(){*

*EmployeeDao.close ();*

*}*

*}*

*public interface EmployeeDao {*

*public void addEmployee(Employee emp);*

*public void updateEmployeeById(Integer Id, Double salary);*

*public Employee findEmployeeById(Integer Id);*

*public void deleteEmployeeById(Integer Id);*

*static void close() {*

*HibernateUtil.Close ();*

*}*

*}*

*public class EmployeeServiceImpl implements EmployeeService {*

*@Override*

*public void CreateEmployee (Employee emp) {*

*// add business logic here*

*new EmployeeDaoImpl().addEmployee(emp);*

*}*

*}*

*public class EmployeeDaoImpl implements EmployeeDao {*

*@Override*

*public void addEmployee(Employee emp) {*

*try (Session session = HibernateUtil.getSessionFactory().openSession()) {*

*session.beginTransaction();*

*session.save(emp);*

*session.getTransaction().commit();*

*} catch (HibernateException e) {*

*e.printStackTrace();*

*}*

*}*

*}*

***Client program***

public class HibernateCRUDTest {

**static EmployeeService empService = new EmployeeServiceImpl();**

public static void main(String[] args) {

Employee emp = getEmployee("Vijay Diwevdi","v.diwedi@gmail.com",

new Date(), 100010.5);

empService.CreateEmployee(emp);

*}*

There can be **single service & DAO interfaces** having different implementation for each entity

1. Primary key generation can be customized using a custom sequence instead of hibernate sequence that is used by default. A sequence generator creates ID before commit

*public class Employee*

*{*

*@Id*

***@GeneratedValue(strategy=GenerationType.AUTO, generator="custom\_gen")***

***@SequenceGenerator(name="custom\_gen", initialValue=10, allocationSize=5, sequenceName="Employee\_seq")***

*@Column(name="employee\_id")*

*private Integer id;*

*}*

1. **@DynamicUpdate** on an entity optimises update query i.e. it is generated only for updated columns. If updated value is same as original value, update query is not generated. In normal update, update query is generated for all columns. **@DynamicInsert** has similar behaviour when providing default column values
2. **@uniqueConstraint** defines combination of columns names not allowed duplicate entries

*@Entity*

*@Table(name="book",* ***uniqueConstraints=@UniqueConstraint(name="uk\_book\_author\_title",***

***columnNames={"book\_title", "author\_Id"}))***

*public class Book*

*{*

*}*

1. An entity field is not saved into DB if it is of **type static or annotated as @Transient**
2. **@Lob** on a field is used to support a type with length more than 255 chars i.e. description field. By default string type is converted to **varchar[255]**
3. A primary key for an entity could be natural key having business significance or a **surrogate key** which is auto generated. Hibernate supports strategies for generating surrogate primary key

*@Id @GeneratedValue(strategy = GenerationType.TABLE)*

*private Integer userId;*

1. Column names of embedded object can be overridden through annotation **@AttributeOverride**

*@AttributeOverride(name = "street", column = @Column(name="HOME\_STREET"))*

*private Address address;*

1. **Composite keys** are handled by hibernate through **@EmbeddedId on @Embeddable class** having composite fields. Embeddable class should be Serializable

*@Embeddable*

*public class EmployeeKey implements Serializable*

*{*

*private int empId;*

*private int deptId;*

*}*

*@Entity*

*public class Employee*

*{*

*@EmbeddedId*

*EmployeeKey empKey;*

*}*

1. **@ElementCollection** is used on list of **value objects** in an entity. Hibernate creates a new sub table for entries in list. Combined with **@JoinTable and @JoinColumn** name of sub table and linked column name can be configured. Auto primary key is generated for collection through hibernate annotations.

*public class Employee*

*{*

*@Id*

*EmployeeKey empKey;*

*String firstName;*

*String lastName;*

*@ElementCollection*

*@JoinTable(name="EMP\_ADDRESS",*

*joinColumns = @JoinColumn(name="EMP\_ID"))*

*@GenericGenerator(name = "hilo-gen", strategy="hilo" )*

***@CollectionId(columns={@Column(name = "ADDRESS\_ID")},generator="hilo-gen",***

***type= @Type(type="long"))***

*Collection<Address> addressList = new ArrayList ();*

*}*

The sub table will have a column for employee primary key along with fields in Address object. If an employee entity is deleted, hibernate removes linked address fields in EMP\_ADDRESS table

1. Collections are by default fetched only when getter for collection is called i.e. **Lazy fetching** happens.

*@ElementCollection (fetch = FetchType.LAZY)*

*Collection<Address> addressList = new ArrayList ();*

Collection addressList gets populated from DB only when getAddressList () is called on object. Other mode is EAGER Fetch mode

1. One to one mapping between entities is performed through **@OneToOne.**

*@Entity*

*public class Person {*

*@Id*

*private Integer Id;*

*private String name;*

*@OneToOne*

*@JoinColumn(name = "VEHICLEID")*

*private Vehicle vehicle;*

*}*

Entity person is mapped to Entity Vehicle, a column VEHICLEID is inserted in table Person

1. One to many mapping is performed through **@OneToMany**

*@Entity*

*public class Person {*

*@Id*

*private Integer Id;*

*private String name;*

*@OneToMany*

*@JoinTable(name="VEHICLE\_PERSON", joinColumns = @JoinColumn(name = "PERSON\_ID"),*

*inverseJoinColumns = @JoinColumn(name = "VEHICLE\_ID"))*

*private Collection<Vehicle> vehicles = new ArrayList<Vehicle>();*

*}*

Hibernate creates a join table having columns names referring primary keys of entities Person and Vehicle. @ManyToOne is optionally used in Vehicle entity to refer to owner

1. One to many mapping is also performed through **“mappedBy” and @ManyToOne.** With this a new join table in not created, instead a foreign key is added to MANY side of relationship

*@Entity*

*public class Person {*

*@Id*

*private Integer Id;*

*private String name;*

***@OneToMany (mappedBy = “owner”)***

*private Collection<Vehicle> vehicles = new ArrayList<Vehicle> ();*

*}*

*@Entity*

*public class Vehicle*

*{*

*@Id*

*private Integer Id;*

*private String name;*

***@ManyToOne***

***@JoinColumn(name = "PERSON\_ID")***

*private Person1 owner;*

*}*

A new column PERSON\_ID is created in Vehicle entity pointing Person entity primary key

1. While using **“mappedBy”** in @OneToMany and @ManyToOne, setters on both sides of relationship should be called

*Person1 p1 = new Person1 ();*

*p1.setId (10);*

*p1.setName ("Shaukat");*

*Vehicle v1 = new Vehicle ();*

*v1.setId (101);*

*v1.setName ("Car");*

*Vehicle v2 = new Vehicle ();*

*v2.setId (102);*

*v2.setName ("Jeep");*

***p1.getVehicles ().add(v1);***

***p1.getVehicles ().add(v2);***

***v2.setOwner (p1);***

***v1.setOwner (p1);***

1. Many to Many mapping is performed through **“mappedBy” and @ManyToMany.** If “mapped by” is not used hibernate will create 2 mapping tables.

@Entity

public class Teacher{

@Id

private Integer Id;

private String name;

@ManyToMany

Collection <Student> students = new ArrayList ();

}

@Entity

public class Student

{

@Id

private Integer Id;

private String name;

@ManyToMany(mappedBy = "students")

Collection <Teacher> teachers = new ArrayList ();

}

1. With **CASCADE option** mapped entities are not required to be saved explicitly, it automatically happens when left side is saved

@Entity

public class Person {

@Id

private Integer Id;

private String name;

@OneToMany (mappedBy = “owner”, **cascade = CascadeType.PERSIST**)

private Collection<Vehicle> vehicles = new ArrayList<Vehicle> ();

}

When entity Person is persisted, all vehicle entities in collection get auto persisted

1. Hibernate by default supports **single table inheritance strategy** with a discriminator type column giving info on entity type. Discriminator type column name and contents can be optionally configured through annotations

*@Entity*

*@Inheritance (strategy = InheritanceType.SINGLE\_TABLE)*

***@DiscriminatorColumn(***

***name = "VEHICLE\_TYPE",***

***discriminatorType = DiscriminatorType.STRING)***

*public class Vehicle*

*{*

*@Id*

*private Integer Id;*

*private String name;*

*}*

*@Entity*

***@DiscriminatorValue ("BIKE")***

*public class TwoWheeler extends Vehicle*

*{*

*private String steeringHandle;*

*}*

1. **Table per class strategy** creates new table for each entity subclass. Subclass Table inherits all base class fields. Blank columns as in case of single table strategy are avoided

*@Entity*

*@Inheritance (strategy = InheritanceType.TABLE\_PER\_CLASS)*

*public class Vehicle {*

*@Id*

*private Integer Id;*

*private String name;*

*}*

1. **Joined strategy** avoids empty columns of single table strategy & duplication of table per class. New tables are created for inherited types but only ID is repeated

*@Entity*

*@Inheritance (strategy = InheritanceType.TABLE\_PER\_CLASS)*

*public class Vehicle {*

*@Id*

*private Integer Id;*

*private String name;*

*}*

Complete data is retrieved through SQL JOIN on parent and child tables

*Select \* from Vehicle join FourWheeler on Vehicle. ID = FourWheeler.Id*

1. With **optimistic locking** hibernate first checks version of a record it is updating in a transaction. An entity field is annotated @version that hibernate queries and updates in every transaction. Update throws an exception if field value at start of transaction is different from value when updating in database.

*public class Person2*

*{*

*@Id*

*@Column(name = "id")*

*private Long id;*

*@Column(name = "person\_name", length = 60)*

*private String personName;*

***@Version***

***@Column(name = "version")***

***private long version;***

*}*

1. Version less optimistic locking is performed using ALL and DIRTY options. With ALL option all fields of record are locked i.e. works **similar to Version approach**. DIRTY option includes a select query and locks only the field which is being updated i.e. concurrent transactions can update different fields of same record

*@Entity*

*@DynamicUpdate*

***@OptimisticLocking(type = OptimisticLockType.DIRTY)***

***@SelectBeforeUpdate***

*public class Person4*

*{*

*}*

*@Entity*

*@DynamicUpdate*

***@OptimisticLocking(type = OptimisticLockType.ALL)***

*public class Person3*

*{*

*}*

1. Hibernate supports **sorted and ordered collections**. In Sorted collections data sorting happens through Java collections framework after data is fetched from DB. In **ordered collections** ORDER\_BY clause is inserted in SQL query returning collection data in sorted order from DB itself. **A comparator interface** needs to be implemented to use a sorted collection such as SortedList, SortedMap etc. Hibernate **supports Set, List, Array, Map & Bag collection types**. A Bag collection is similar to Set but it can contain duplicate values and gets used where collection values are repeatable such as answer keys of a question paper

1. **Caching in Hibernate:** Hibernate **has first level of cache** provided by session object. At session level hibernate allows caching through primary key as well as natural ID. An update to a entity field also updates in session cache.
2. Second level cache is set up across sessions. Requires dependent libraries for L2 cache implementation and hibernate cache integration

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-ehcache</artifactId>

<version>3.6.10.Final</version>

</dependency>

<dependency>

<groupId>net.sf.ehcache</groupId>

<artifactId>ehcache</artifactId>

<version>2.10.4</version>

</dependency>

1. L2 cache is **configured** in **hibernate.cfg.xml**

<property name="hibernate.cache.use\_second\_level\_cache">true</property>

<property name="hibernate.cache.region.factory\_class"> org.hibernate.cache.EhCacheRegionFactory

</property>

1. L2 Caching is enabled per entity basis and a caching strategy is applied

**@Entity**

**@Cacheable**

**@Cache (usage=CacheConcurrencyStrategy.READ\_ONLY)**

public class Person2

{

@Id @GeneratedValue(strategy = GenerationType.AUTO)

private Integer userId;

String lastName;

}

L2 caching will work for Person object only on read operations i.e. SELECT queries

1. A many to one mapped entity is not cached in l2 cache.
2. **Query caching** is also supported through L2 Cache. A cache entry includes a complete query **including parameter values**. Caching may not work if parameter values in query are reordered in WHERE clause. Hibernate first looks into L2 cache for results and then hits database.

*<!-- Enable l2 for query cache -->*

***<property name="hibernate.cache.use\_query\_cache">true</property>***

*Session sessionR1 = sf.openSession();*

*sessionR1.beginTransaction ();*

*Query query1 = sessionR1.createQuery ("From Person2 where age >: ageVal");*

*query1.setParameter ("ageVal", 35);*

***query1.setCacheable (true);***

*List<Person2> list1 = query1.list ();*

*sessionR1.getTransaction ().commit ();*

*sessionR1.close ();*

*sessionR1 = sf.openSession ();*

*sessionR1.beginTransaction ();*

*Query query2 = sessionR1.createQuery ("From Person2 where age >: ageVal");*

*query2.setParameter ("ageVal", 35);*

***query2.setCacheable (true);***

*List<Person2> list2 = query2.list ();*

1. Hibernate maintains a **modify timestamp** for a table. Beyond this time if a cached query is accessed hibernate will hit database.
2. By default hibernate uses AUTO flush mode which flushes changes to DB on commit. ALWAYS flush mode flushes after every SQL query

Employee emp = CreateEmployee("Vijay Diwevdi","v1.diwedi@gmail.com", new Date(), 100010.5);

**session.setHibernateFlushMode(FlushMode.ALWAYS);**

session.beginTransaction();

**Integer id = (Integer)session.save(emp);**

BigInteger Count = (BigInteger)session.createNativeQuery("Select count(\*) from emp\_table").uniqueResult();

System.out.println("Employee Count =" + Count);

session.getTransaction().commit();

Employee count will be returned as 1 with Flush mode as ALWAYS. In AUTO flush mode it will be 0 since database will be updated on commit

1. With MANUAL flush mode, flush has to be explicitly called to flush changes to database

**session.setHibernateFlushMode(FlushMode.MANUAL)**;

session.beginTransaction();

Integer id = (Integer)session.save(emp);

***Session.flush()***

1. A Driver manager services one request at a time. For web app Connection pool is to be configured. It is created on an application server i.e. tomcat. For tomcat it is configured as a JNDI resource, which is then referenced in hibernate.cfg.xml

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

*<Context>*

*<Resource name="jdbc/MyDB"*

*auth="Container"*

*type="javax.sql.DataSource"*

*driverClassName="com.mysql.jdbc.Driver"*

*url="jdbc:mysql://localhost:3306/testdbforhibernate5"*

*username="root"*

*password="root"*

*maxActive="100"*

*maxIdle="20"*

*minIdle="5"*

*maxWait="10000">*

*</Resource>*

*</Context>*

*<hibernate-configuration>*

*<session-factory>*

***<property name="hibernate.connection.datasource">java:comp/env/jdbc/MyDB***

***</property>***

*<property name="dialect">org.hibernate.dialect.MySQL5Dialect</property>*

*</session-factory>*

*</hibernate-configuration>*

1. JNDI resource can be added to server itself in which case its globally available or to project only. Based on JNDI parameters server creates minimum connections at start. Every request is executed in a separate thread and has a separate DB connection.
2. C3p0 is a third party connection pool. Additional dependencies in POM file. Application server is not required for configuring this connection pool

*<dependency>*

*<groupId>com.mchange</groupId>*

*<artifactId>c3p0</artifactId>*

*<version>0.9.5.2</version>*

*</dependency>*

*<dependency>*

*<groupId>org.hibernate</groupId>*

*<artifactId>hibernate-c3p0</artifactId>*

*<version>5.2.11.Final</version>*

*</dependency>*

If hibernate-c3p0 is not provided, hibernate will use driver manager for connection instead of C3 connection pool

1. Hibernate integration with **Hikari connection pool** is similar but uses extra dependency for logging purpose.
2. Hibernate supports **stateless session** that is useful with batch operation. It doesn’t support caching so there is no chance of out of memory. Also useful if read only operations are performed on DB
3. N Plus one query problem happens in hibernate with one to many relation, for a parent having N children a Select query is fired for each child with a normal HQL.

*List<Department> departments = session.createQuery("From Department", Department.class)*

*.getResultList();*

*for (Department department : departments)*

*{*

*List<Employee4> employees = department.getEmployees();*

*}*

*A select query would be fired for each department employee when accessed*

This is resolved by using a JOIN query instead of simple HQL

*departments = session.createQuery("From Department d JOIN fetch d.employees", Department.class)*

*.getResultList();*

Making Fetch type EAGER doesn’t resolve this since N queries for children still get executed in advance

1. **Log4J integration** with hibernate
2. Hibernate support interceptors that can be used to inspect or modify values written or read from DB.

*public class LoggerInterceptor extends EmptyInterceptor {*

*private static Logger logger = LogManager.getLogger(LoggerInterceptor.class);*

*@Override*

*public boolean onLoad(Object entity, Serializable id, Object[] state, String[] propertyNames, Type[] types) {*

*logger.info("onLoad method is called...");*

*if (entity instanceof Person4) {*

*Person4 person = (Person4) entity;*

*logger.info(person);*

*Arrays.asList(state).forEach(logger::info);*

*}*

*return super.onLoad(entity, id, state, propertyNames, types);*

*}*

*@Override*

*public boolean onSave(Object entity, Serializable id, Object[] state, String[] propertyNames, Type[] types) {*

*logger.info("onSave method is called...");*

*if (entity instanceof Person4) {*

*Person4 person = (Person4) entity;*

*logger.info(person);*

*}*

*return super.onSave(entity, id, state, propertyNames, types);*

*}*

*}*

An empty interceptor class implements interceptor interface providing default overrides for all intercept interface methods. Entity values can be fetched in onSave() method from entity getters. In onLoad() entity values are fetched from state array.

1. An Interceptor can be registered at Session Factory level as well as Session level

*Session session = HibernateUtilNPlus1.getSessionFactory ().*

*withOptions().interceptor(new LoggerInterceptor()).openSession()*

1. Available **caching strategy options** are NONSTRICT\_READ\_WRITE, READ\_ONLY, READ\_WRITE & TRANSACTIONAL.
2. **READ\_ONLY strategy** assumes data will only be **fetched from DB but not written**
3. With **READ\_WRITE** hibernate checks what updates have happened to DB and updates cache accordingly

1. With NONSTRICT\_READ\_WRITE, update checks are not enforced**, so cache data can be out of sync with DB.** TRANSACTIONAL is most strict with little to gain from L2 caching
2. Processor brings in **data into cache via cache line**, which are blocks of fixed size with Cache entry getting created when cache line is copied
3. **Isolation Level with multiple users**: A "Session.Lock" mode in hibernate **controls entity from being modified and read from multiple sources** simultaneously based on **isolation level**. Isolation gives user the feeling that he owns the database when several users are concurrently querying. **Query result depend on one of configured isolation leve**l
   1. READ UNCOMMITTED: read data is **not consistent** with other parts of table, however this level ensures **quickest performance** as data is read from table blocks directly
   2. READ COMMITTED: rows returned by a query are those that were **committed when the query was started**.
   3. REPEATABLE READ: the rows returned by a query are those that were **committed when the transaction was started**, Data is fetched from undo or rollback segment for each row that is found to be too new bringing some performance penalty
   4. SERIALIZABLE: all transactions occur in a completely isolated fashion, or serially, similar to REPEATABLE READ but with varied implementation
4. **Load vs Get** in Hibernate: **Get hits DB if object is not found in cache** and returns a fully initialized object **while load returns a proxy and initializes object only if a method is invoked**, thus improving on performance. Load throws an exception if object is not found. Can be used in cases where an object can be found in DB with certainty. E.g. Symbol of a stock whose quote is to be updated frequently.
5. **Persisting detached object:** An Object is said to be detached when session closes and to persist detached object we need to create a new session & call session.Update
6. **HQL** **works with persistent objects** and their properties, RATHER than tables or columns. It returns a list of objects matching a criteria

*Query query = session.CreateQuery("from UserDetails");*

*Query query = session.CreateQuery("****select username from UserDetails****");*

*List<String> usernames = (List<String> )query.list();*

1. For multiple fields, hql could be written as below

*Query query = session.CreateQuery("select new map (userId, username) from UserDetails");* or

*Query query = session.CreateQuery("select new Object[](field1, field2, ..., fieldN) from UserDetails");*

1. Using positional arguments with HQL

*Query query = session.CreateQuery ("from UserDetails where userId > ? And username = ?);*

query.setInteger (0, Integer.parseInt (minUserId));

query.setString (1, userName);

1. HQL supports **named Parameters** for passing SQL arguments

*Query query = session.CreateQuery ("From Circle where ID >* ***:userId****");*

*query.setParameter ("userId", 10);*

*return query.list()*

1. In Hibernate 3 Stored procedure having output parameter is indirectly called through a wrapper procedure.

Query query = session.createSQLQuery ("**CALL GetNameByAddressWrapper( ?)")**

query.setParameter(0, 4);

List result = query.list();

for (Object o: result){

System.out.println(o);

}

*CREATE PROCEDURE `getNameByAddressWrapper`(IN addrId Integer)*

*BEGIN*

*call getNameByAddress(addrId, @outName);*

*select @outName;*

*END*

*CREATE PROCEDURE `getNameByAddress`(IN addressID integer, OUT custName varchar(20))*

*BEGIN*

*select b.firstName into custName from cust\_address a, customer b*

*where a.address\_id = addressID and a.custId = b.custId;*

*END*

1. Hibernate 5 supports StoredProcedureQuery for stored procedures

*StoredProcedureQuery procedureQuery = session.createStoredProcedureQuery("phone\_count");*

*procedureQuery.registerStoredProcedureParameter("personId", Int.class, ParameterMode.IN);*

*procedureQuery.registerStoredProcedureParameter("phoneCount", Int.class, ParameterMode.OUT);*

*procedureQuery.setParameter("personId", 1);*

*procedureQuery.execute();*

*Long phoneCount = (Long) procedureQuery.getOutputParameterValue("phoneCount");*

*System.out.println("Phone Count:"+phoneCount);*

1. Native **SQL functions** are called in similar manner, no wrapper is required

*query = session.createSQLQuery("****Select GetNameByAddressId ( ?)"****).SetParameter (0, 4);*

*System.out.println (query.uniqueResult ());*

*CREATE FUNCTION `getNameByAddressID`(addressID Integer) RETURNS varchar(20)*

*BEGIN*

*declare custName varchar(20);*

*select b.firstName into custName from cust\_address a, customer b*

*where a.address\_id = addressID and a.custId = b.custId;*

*RETURN custName;*

*END*

1. Native SQL Functions are invoked in Hibernate 5 using Callable statements

*int personId = 2;*

*final AtomicReference<Integer> callCount = new AtomicReference<>();*

*Work work = new Work(){*

*@Override*

*public void execute(Connection connection) throws SQLException*

*{*

***try (CallableStatement callableStatement = connection.prepareCall(***

***"{ ? = call call\_count(?) }" ))*** *{*

*callableStatement.registerOutParameter( 1, Types.INTEGER );*

*callableStatement.setInt( 2, phoneId);*

*callableStatement.execute();*

*callCount.set(callableStatement.getInt(1) );*

*}*

*}*

*};*

***session.doWork(work);***

*if(callCount != null){*

*System.out.println("Call Count:"+callCount.get());*

*}*

1. Named Queries defined HQL query on entity
2. Fetching records using HQL in an employee address one to one mapping. FETCH keyword is required in HQL when getting all records

*private static void getEmployeeAndAddressWithHQL(int Id)*

*{*

*try (Session session = HibernateUtil.getSessionFactory().openSession())*

*{*

*String hql = "FROM Employee2 emp left join fetch* ***emp.address*** *where emp.id = :empId";*

*Query <Employee2> query1 = session.createQuery(hql);*

*query1.setParameter("empId", Id);*

*Employee2 emp = query1.uniqueResult();*

*System.out.println(emp);*

*}*

*}*

*@Entity*

*public class Employee2 {*

*@Id*

*@GeneratedValue*

*private Integer Id;*

*private String name;*

*@OneToOne (cascade=CascadeType.ALL)*

*@JoinColumn(name = "address\_id")*

*private Address2 address;*

*}*

*@Entity*

*public class Address2*

*{*

*@Id @GeneratedValue*

*private Integer addressId;*

*private String state;*

*private String city;*

*}*

1. Fetching fields using **HQL in an employee address one to one mapping** or one to many mapping. HQL returns a list of object arrays in this case.

*try (Session session = HibernateUtil.getSessionFactory().openSession())*

*{*

*String hql = "Select emp.name, emp.salary, addr.city from Employee3 emp left join emp.addresses addr where emp.id > :empId";*

*Query <Object[]> query1 = session.createQuery(hql);*

*query1.setParameter("empId", id);*

***List<Object[]> empfields = query1.list();***

*for(Object objects[] : empfields ){*

*String name = objects[0].toString();*

*String city = objects[2].toString();*

*Integer sal = (Integer)objects[1];*

*System.out.println("Name: "+ name + " City: " + city + " Salary: " + sal);*

*}*

*}*

*@Entity*

*public class Employee3 {*

*@Id*

*@GeneratedValue*

*private Integer Id;*

*private String name;*

*@OneToMany(cascade=CascadeType.ALL, mappedBy=”employee”)*

*private Address2 address;*

*}*

*@Entity*

*public class Address3*

*{*

*@Id @GeneratedValue*

*private Integer addressId;*

*private String state;*

*private String city;*

*@ManyToOne*

*@JoinColumn(name="employeeId")*

*Employee3 employee;*

*}*

1. **@NamedNativeQuery annotation** defines a **native SQL query** to be executed by hibernate, which requires a class along with query expression for hibernate to populate it with query result. Native SQL query is preferred for **database specific features** i.e. CONNECT BY in oracle

*@NamedNativeQuery(name="UserDetail.byName", query="select \* from user\_details where username = ?", resultClass=UserDetails.class)*

1. **Native queries** are created using CreateNativeQuery and return a list of object arrays

*try (Session session = HibernateUtil.getSessionFactory().openSession())*

*{*

*String hql = "Select \* from Person";*

*Query <Object[]> query1 =* ***session.createNativeQuery(hql);***

*List<Object[]>fields = query1.list();*

*for (Object objects[] : fields ){*

*BigInteger id = (BigInteger) Object[0];*

*String name = objects[0].toString();*

*Integer sal = (Integer)objects[2]];*

*System.out.println("Name: "+ name + " City: " + city + " Salary: " + sal);*

*}*

*}*

1. Types for entity columns can be specified through addScalar() method to avoid additional conversions at time on return

*try (Session session = HibernateUtil.getSessionFactory().openSession())*

*{*

*String hql = "Select ID, Name from Person";*

***Query <Object[]> query1 = session.createNativeQuery(hql)***

***.addScalar(“Id”, LongType.INSTANCE)***

***.addScalar(“Name”, StringType.INSTANCE);***

*List<Object[]>fields = query1.list();*

*for (Object objects[] : fields ){*

*Long id = (Long)object[0];*

*String name = (String)objects[1];*

*}*

*}*

1. **Criteria Object** is an **alternative to where clause** for handling complex queries added as a restriction on Criteria as below, in which object user with name "User1" is returned.

*Criteria criteria = session.createCriteria(UserDetails.class)*

*criteria.add(Restrictions.eq("username", "User1"));*

*List<UserDetails> users = (List<UserDetails>)criteria.list();*

1. **Adding Restrictions** **through criteria API** supports chaining which is equivalent to AND in SQL. OR restriction can be added through "Restrictions.Or" keywords as below

*criteria.add(Restrictions.or(Restrictions.between("userId",0,3), Restrictions.between("userId",7,10)));*

1. **Projections through Criteria:** Projections enable **returning specific fields rather than whole object** and is used as below, with userId fields retrieved for all users

*Criteria criteria = session.createCriteria(UserDetails.class).setProjection(Projections.property("userId"));*

1. Projections also **support** **aggregate functions** like count, max etc.

Criteria criteria = session.createCriteria(UserDetails.class).setProjection(Projections.count("userId"));

1. **Query by Example** is again used with complex queries to tell hibernate to return a **particular type of object as represented by Example**, also allowing exclusion of certain fields

*UserDetails exampleUser = new UserDetails();*

exampleUser.setUserName(User 5)

*Example example = Example.create(exampleUser);*

*Criteria criteria = session.createCriteria(UserDetails.class).add(example); //all objects matching*

1. Fields can be **excluded from an Example through exclude property**

*Example example = Example.create(exampleUser).excludeProperty("userName"); //this eliminates username as selection criteria element*

1. **Like attribute**

*exampleUser.setUserName(User 1%)*

*Example example = Example.create(exampleUser).enableLike() //all users matching specified wildcard get selected*

**Integration Strategies and Patterns**

1. **Integration Strategies** aredefined by Communication patterns between processes & modules. Domain Model and Transaction processing model are couple of points on middle layer. TPM is like a command pattern. Domain Model represents business or functional model i.e. representing attributes and methods in classes of Domain being modelled
2. **Pipeline arch** has stages that it goes through e.g. web servers receive http request that is processed and response given. A Component architecture is a service oriented architecture having many components
3. A Logical model represents functional components. Physical models define how to get something assembled an where to assemble
4. A **message channel** is a logical channel to connect the applications. One application writes messages to the channel and the other one (or others) reads that message. **Queue and topic** are examples of message channels
5. **Message translator** transforms messages in one format to another. One application sends a message in XML format, but the other accepts only JSON messages so translator has to transform XML data to JSON
6. **Correlation Identifier** gives the possibility to match request and reply message when asynchronous messaging system is used. Producer generates a unique correlation identifier and sends message. Consumer processes messages and send reply with attached correlation identifier given in request message. Producer correlates request and reply message based on correlation identifier.
7. **Content-Based Router** examines message contents and route messages based on data contained in the message
8. **Content Enricher** enriches message with missing information. Usually external data source like database or web service is used.
9. **Event-Driver Consumer** enables providing an action that is called automatically by the messaging channel or transport layer. It is asynchronous type of pattern because receiver does not have a running thread until a call back thread delivers a message.
10. **Polling Consumer** is used when receiver is required to poll for a message, process it and next poll for another. This pattern is synchronous as it blocks thread until a message is received. This is in contrast with an event-driven consumer. An example of using this pattern is **polling for a file**
11. **Wire Tap** copies a message and **routes it to a separate channel**, while the original message is forwarded to the destination channel. It is generally used to inspect message or for analysis purposes.

**Design Patterns GOF**

1. Design Patters are categorized as Creational, Structural & Behavioural. **Creational patterns** abstract creation of an object framework. **Structural Patterns** decide **how objects should be composed** to provide different functions. **Behaviour Patterns** decide **communication between objects** at runtime
2. Creational Patterns prevent hardcoding object type when different types of object get created in an object hierarchy, making decisions based on inheritance. **Factories isolate object creation from their usage,** and are a choice when a class does not know which subclass instance is to be created. An abstract factory provides highest level of encapsulation, responsible for creating a group of individual factories.
3. **Abstract Factory** provides common interface for creation of **different families of dependent objects**. Abstract graphic editor i/f for creation of Schematic Editor and Part editor frameworks. Also graphic themes for different OS families
4. **Builder** enables **construction of a complex object**, with parts built independently. I.e. Image Loader can have abstract framework for different formats with ReadFileHeader(), ReadDataHeader(), Read Data() methods implemented for different file formats. Provides a **finer control over object construction** with access to its parts, compared to **factory which constructs an object once by invoking its constructor**. End products differ considerably compared to factory produced products.
5. **Factory** defines interface for creating an object, but **subclasses decide how exactly to create**. E.g. UIObjectFactory interface for instantiating UIObjects, based on object ID or type, specific subclass gets instantiated
6. **Prototype** class framework depends on **one or more prototype objects that are cloned** to create concrete object instances at runtime. Reduces number of classes in system where **end objects have minor variations**. e.g. used to model various **types of schematic page connectors** ie Offpage, bidirectional, unidirectional etc. Clone() & Initialize() are main object methods
7. **Singleton** ensures unique instance of class and its subclasses.
8. **Flyweight,** a structural pattern useful in **working with large no. of fine grained objects** having intrinsic and extrinsic state. **Intrinsic state** shared by all objects while **extrinsic state** is passed on as context **at run time to support rendering** at different coordinates or with different fonts as for characters. Lesser extrinsic state, better cost effectiveness in storage terms
9. **Bridge** maintains an **abstraction hierarchy & implementation hierarchy**. Client code refers to abstraction hierarchy only, underlying implementation invoked by server depending on platform specific aspects.eg for **remote control** an abstract hierarchy would comprise of remote designs for different devices and **implementation hierarchy includes** **concrete implementations** on design components for similar remote from each vendor. Solves design problem where if implementation is merged with design class, it would become difficult to manage with vendor specific custom implementation details. Better to isolate this in separate class.
10. **Decorator** adds **behaviour to an object at run time** by composing object in a decorator class while also **extending decorator from object class** to provide a common interface to decorator and object class. A decorator works well on light weight object suitable for effective composition like java IO objects or **adding a windows frame around a third party grid view component** through a decorator wrapper class e.g. an MFC based class frame around objective grid library. On cons side, decorator pattern can make **code maintenance more difficult** as it provides a lot of similar kind of small objects. In a retail shopping store using decorator can **compute cost of combination of items purchased** by a customer. Customer purchases a Britannia bread, that gets modelled as BritaniaBreadDecorator derived from bread. Its method getCost() returns 25. If customer buys biscuit along with Britannia bread, he gets 20 per cent discount on Britannia bread. This gets modelled as **BritaniaBreadBiscuitDecorator** inherited from BritaniaBreadDecorator. Its Method *getCost() returns 20 + (20%)\* BritaniaBreadDecorator->getCost()*
11. **Composite** enables class hierarchies **comprising of primitives and other composites sharing common base class**. Primitive operations need support for Adding, removing elements and traversal through iterator. e.g. **Component objects implements graphic object** and **composes of primitive objects** i.e. Line, rectangle, arcs
12. **Facade** provides a **simple interface to a complex system**, generally used as default. While interfaces of subsystems still remain exposed for special interaction of system. client send request to facade which forwards to subsystem COM Framework, invoke method names
13. **Adopter** enables usage of existing **third party class having incompatible interface** i.e. Adaptee. Class adapters inherit privately from adaptee and publicly from target i.e. base domain class sharing common interface with other domain objects. Object adapter composes adaptee and redirects method calls to adaptee wherever needed. **Class adapter** easier to use since **protected members of adaptee are accessible**, while **object adapter give better run time flexibility**. Used wrapper adapter class to add **CIS components to schematic from CIS symbol library**. Need to use existing classes whose interface is different
14. **Proxy** is a **placeholder** for accessing large resource consuming objects i.e. embedded bitmaps in document.
15. **Chain of Responsibility:** Request travels through objects in a chain of objects till handler is found. Useful when object issuing a request has no idea as to who will handle it. Dialogs, buttons need retrieve help. This pattern is useful in context help systems where clicking **Help on dialog box button, looks for help handler in button class**, if not found handler looked in dialog class and then in windows class.
16. **Command pattern** encapsulates one or more **commands as objects sub classed from abstract command class**. It provides a common framework for execution of all commands, supports Undo/Redo and using commands in a Macro
17. **Memento** stores **snapshot of internal object state** which can be retrieved later after command execution. Offers a **narrow interface for client** but wide interface for **originator object** that creates memento and is responsible to retrieve info at time of Undo. Useful in cases where movement of **connected components on view** needs recalculation of links between them
18. **Iterator** provides a **cursor feature** to access elements of a collection object sequentially. Iterator works **transparently on various structures** i.e. list, vectors, maps
19. **Visitor** enables performing **one or more operations on an existing object structure** without changing it. Operation could be monitoring performance of objects, calculating total cost, type checking on syntax trees. A **new operation** defined by adding **a new visitor as a subclass to visitor hierarchy**. There is a visitor hierarchy, with **each visitor node representing an operation on object tree**. Each object in object hierarchy has an **accept member**, a double dispatch operation that takes visitor base class as an argument and invokes object specific handler method of visitor. Visitor relies on object public interface to retrieve its state information and a suitable object interface is desirable for use with the visitor. Difficult to use in case object hierarchy is regularly changing.
20. **Template** provides a **skeleton framework for application/algorithm flow**, with variant behaviour left to be implemented by subclasses ie Document View Architecture. Factors out common behaviour reducing code size and making better flexibility.
21. **Strategy** defines & manages a **family of algorithms**, each encapsulated in a different class. Algorithm selection varies independently as per client type. **Syntax checker** makes a traversing choice depending on verilog/VHDL editors. Clients need not know which syntax checker is selected internally. Register allocation scheme for compilers. Poses some **problems when algorithms having different complexity levels** exposed via common strategy interface, as **extra information** may get passed on to algorithms that may not use it. As implementation choice, strategy objects should be **used only for complex algorithms**; else a default behaviour implemented within a context can be sufficed.
22. **Mediator** enables communication **between a set of objects** by managing their interdependencies. Promotes loose coupling between these objects or colleagues. E.g. coordination of controls in a dialog box can be done through Mediator. Mediator class gets notified when action on control is performed & it takes further action on other controls
23. **State** transparently enables behaviour change of object at run time when its state changes. State hierarchy represents an **abstract state class**, with specific state modelled as concrete classes. State transition logic is decentralized and can be further extended by adding new concrete states. A containing wrapper class delegates request to state class. States in TCPConnection are implemented using state pattern
24. **Interpreter** is used with a regular expression in interpreting a language; **depending on type of expression a concrete class is invoked**. Implemented with syntax checker in VHDL/Verilog
25. **Observer** communicates a **change in state of a subject, to dependent objects**, the observers, in order for them to take respective actions, i.e. In document view, if data in document changes associated view get notification to update themselves, somewhat similar happening in excel spread sheets & charts
26. Comparison of Patterns
    1. **Bridge vs. Adapter:** Bridge makes components work before they are designed, adapter afterwards
    2. **Decorator vs. Strategy**: If base object is heavy weight, strategy is better option.
    3. **Decorator vs. Compositor**: Decorator does not need modifying component code, just inheriting its type and composing it while Compositor recursively building a complex object from other simple and compound objects
    4. **Factory vs. Abstract Factory:** While factory is used to make a choice from hierarchy of classes, an abstract factory is responsible for instantiating factories catering to a family of classes
27. **Problem factory solves:** All concrete classes need not be in direct contact with client and client can use them through common interface. **Prevents cluttering of new in client** for each concrete class
28. A **singleton design pattern** is useful in situations requiring **cross cutting concerns** such as an **application logge**r and may be used internally in AbstractFactory, Builder & Prototype patterns along with Facade pattern. A disadvantage of singleton is that it **makes unit testing harder** as it **introduces a global state** requiring access in multi-threaded context to be serialized. To create a thread safe optimal singleton class it is desirable to **synchronize only critical section of creation code** through **double checked locking pattern** rather than synchronize whole create (getInstance()) method, making this of type lazy loading. In double checked locking singleton instance is checked twice, once at beginning of getInstance() method, and next in synchronized block inside first check, an expensive operation happening only on instance creation first time. As an alternative way, a **singleton class can contain static final instance** which gets initialized during class loading and is thread safe, this method being of type early loading.
29. Though Singleton has a **private constructor**, it can be invoked through reflection, and hence constructor should throw an exception if singleton instance exists. To prevent singleton duplication during serialization where readobject() returns a new instance every time, **readResolve() method is used** to replace new singleton object with original instance
30. A Builder pattern is used to build complex objects from simple ones **through separation of complex object construction** from their representation, abstracting steps such that different implementations of these steps can construct different object representations. **Object creation sequence is managed by director class** that takes in a concrete builder as parameter, performing necessary operations for object creation

**J2EE Patterns**

1. J2EE design patterns are divided into Presentation tier, Business tier & Integration tier
2. Presentation tier patterns are as below
   1. **Intercepting Filter** enables pre-processing and post-processing of a request
   2. **Front Controller** is a **centralized controller** for managing request handling flow
   3. **View Helper:** Encapsulates **logic in a view not related to presentation** such as formatting related code into Helper components **implemented as Java beans or custom tags**. A helper may represent a command object, delegate, or XSL transformer. A helper can be implemented as a Java bean or a custom tag. While making invocations to business tier, helper should simply invoke a business service
   4. **Composite View:** Creates an aggregate View from subcomponents. Multiple smaller views, static and dynamic, are pieced together to create a single template. These is done through either **java bean** or through **JSP include strategy**, a java bean implements the custom logic to control view layout and composition. Page layout decisions may be based on user roles or security policies, making it more powerful than standard JSP include functionality.
   5. **Service to Worker:** Combines a **Dispatcher component with the Front Controller and View Helper Patterns** to handle client request. Controllers delegate content retrieval to helpers, which in turn populate of the intermediate model for view. **Dispatcher is responsible for view management,** such as translate request to appropriate view or in more complex scenario invoke a business service for appropriate view to display
   6. **Dispatcher View:** Combines a Dispatcher component with the Front Controller and View Helper Patterns, **deferring many activities to View processing**. Controllers do not delegate content retrieval to helpers, as these activities are deferred to the time of view processing. The controller and the dispatcher have limited responsibilities, compared to Service to Worker pattern
3. Business Tier patterns are as below
   1. **Business Delegate:** reduces coupling between tiers and providing entry point for accessing services provided by another tier. Delegate may also cache results and references to remote business services improving performance. **Business Delegate uses a Lookup Service** to locate service, which is implemented as a Service Locator pattern
   2. **The Message Façade** pattern discusses how to partition logic for asynchronous use cases
   3. **EJB Command Pattern** advocates placing business logic in lightweight, plain Java Command beans. It decouples client from EJB itself
   4. **Transfer Object Factory:** prescribes centralizing data transfer object creation and consumption logic into a single layer (implemented as session beans or plain java factories).
   5. **Generic Attribute Access** discusses how to provide **a domain-generic interface** to attributes of an entity bean for maintainability and performance.
   6. **Value Object:** Facilitates data exchange between tiers by reducing network requests
   7. **Session Facade:** Hides business object complexity; centralizes workflow handling. Session Facade shows how to properly partition the business logic to help minimize dependencies between client and server, forcing use cases to execute in one network call and in one transaction. **A use case involves multiple fine-grained invo**cations of session/entity beans adding the overhead of multiple network calls/transactions
   8. **Composite Entity:** Represents a best practice for designing coarse-grained entity beans by grouping parent-dependent objects into a single entity bean.
   9. **Value Object Assembler:** Assembles a composite value object from multiple data sources.
   10. **Value List Handler:** Manages query execution, results caching, and results processing.
   11. **Service Locator:** Encapsulates complexity of business service lookup and creation; locates business service factories. Service Locator design pattern is used to locate services using JNDI lookup, **making use of caching**
4. Integration tier Patterns
   1. **Data Access Object:** Abstracts data sources; provides transparent access to data.
   2. **Service Activator:** Facilitates asynchronous processing for EJB components.
   3. **Domain Store:** implements transparent persistence for your object model along with DAO
   4. **Web Service Broker:** exposes one or more services in application to external clients as web service using XML and standard web protocols. It uses one or more Service Activators to perform asynchronous processing of a request

**Identity Federation**

1. **Federated identity management (FIM)** is arrangement made among enterprises, such that users use **same identification data** to obtain access to networks of all enterprises in the group. The use of such a system is called **identity federation**. E.g. User logging on network of BCONE was automatically given access to Success Factors or HRIS. Identity federation enables **SSO across different systems**
2. Identity federation is **supported through open standard protocols.** Most widely used are (**LIFF)** Liberty Identity Federation Framework & (**SAML)** Security Assertion Markup Language.New protocols i.e. Higgins, Internet2, Shibboleth, OpenID are also used for federating Identities.
3. To build a **circle of trust** a network has to be built between **service providers (SPs) and identity provider (IDP).** A valid federation request from an SP to IDP gets generated if the IDP recognizes SP. A **metadata exchange between IDP and SP** happens in the circle of trust and describes trusted partner. Metadata can contains public key certificates, key descriptors for message signing, a URL for the SSO service, a URL for the SLO (single logout) service, etc.
4. After **federation initiator SP** is within the circle of trust, the user can initiate an SSO request. A **successful completion of SSO** marks a link between federated user ID and the local user ID for that application. **SSO is achieved through** **session tokens or cookies** and is implementation specific. After a user is federated from a specific application he can access any other federated applications within circle of trust using federated user id. Some financial organizations on top of SSO can **opt for another security layer** i.e. a userid/password or even biometric authentication.
5. Assuming a user is successfully federated through an application A1 and attempts to log on to another application A2, User provides federated user id to log on to application A2. In this case identity federation products **automatically provision a new account**. When a user requests a resource in an application, application needs to know whether a request for an SSO should be initiated. It first checks **if the browser is IDP-enabled** by detecting a cookie in a specified sub-domain. If cookie is found browser is qualifies as IDP-enabled. **This cookie is identified as federation cookie** and is set by the SP. If the cookie is not found, a request for reading the federation cookie is initiated by SP to IDP and application initiates SSO, if cookie is present.
6. **SAML is used to communicate identities** between organizations. User tries to access an Application hosted on an SP. SP directs user request to concerned IDP that prompts user to log on. Once a user is authenticated, **IDP creates an xml based SAML assertion** (contains info such as user name, organization, department, designation etc.). IDP passes SAML assertion back to SP, based on which authorizes user access. In case another SP is to be accessed by user, IDP passes same assertion to other SP eliminating need for user to enter credentials again

**Load Balancers**

1. A **Hardware load balancer** is a dedicated load balancing hardware having a public IP sitting in front of web servers. All clients’ requests come into a single public IP (for load balancer) which then selects a webserver and forwards request, based on configured Load Balancing algorithm. Individual web servers **have agents running which report to load balancer system** aspects like CPU/Process utilization, other m/c statistics, based on which current request is routed to it or some other server. In case of **server failover** request gets redirected to another server
2. **Setting up a Hardware LB is complex** requiring good TCP/IP knowledge. Also it is a single point of failure, due to which an additional LB needs to be configured that can take over in case main LB goes down. Hardware LB provides an **additional layer of security** since webservers are not exposed to public internet
3. LB Algorithms
   1. Round Robin
   2. **Round Robin & Least Connection:** LB also monitors how many open connections each server has and redirects to one having least open connections
   3. **Connection Ratio:** configured by admin based on capabilities of server behind LB, i.e. for 2 requests going to server#1, 1 request will go to server#2
   4. **Fastest:** traffic sent to server giving fastest response
4. Software LB comprises of cluster of servers with one server as master cluster server
5. **Affinity:** once a user gets associated with a load balanced server, he will remain with server rather than hitting another web server on next request. Affinity Types are NONE, SINGLE, NETWORK. In **SINGLE affinity a request is identified by IP address** and mapped to server. Problem occurs when multiple clients behind NAT access service. Since NAT translates them to same IP, all requests go to a single server overloading it. With **NETWORK affinity, a requests from a network goes to same server**

**SOAP Web services**

1. SOAP is an XML based protocol for building platform web services. SOAP protocol uses standards from HTTP and XML. SOAP web service exposes business logic through WSDL. All Service providers **register their services on internet** through Universal Description, Discovery, and Integration **(UDDI) registry**. Client searches for a service in UDDI that returns all providers offering that service and Client then makes a choice and gets service's WSDL. SOAP Web service is useful in **enterprise application integration scenarios** with high QoS requirements
2. WSDL is a **contract definition** and contains a list of all ops in service, along with parameter types required in messages. WSDL is a kind of m/c readable and clients use tools to understand WSDL.
3. JAX-WS is a Java XML programming API and part of Java EE platform. It is used for SOAP web services. JAX-WS uses annotations, from Java SE 5, to simplify the development and deployment of web service clients and endpoints. In earlier versions of java there was JAX-RPC (java API for xml based remote procedure calls). A service class can be written using plain java, **and using JAX-WS that class can be converted to properly functioning web service** without needing to deal with xml. JAX-WS will auto generate WSDL. With client apps, JAX-WS enables writing java calls without needing to deal with WSDL. To make a class a web service it should be annotated with **@WebService annotation**. Web service is exposed through **Endpoint** which takes in a URL and a service class
4. **Wsimport tool** generates JAVA classes on client side representing a web service and compiles them into binary classes. This tool is a part of JDK and is run from inside project directory from console or can be **run from within eclipse** through "external tools configurations"

*wsimport -keep -verbose -s ./src/ -d ./bin/ <WSDL URL>*

1. All soap services implement **http bindings by default**; others i.e. JMS & SMPT could also be used. Soap level security mechanism was needed as SOAP supports multiple bindings. WS security adds overhead to soap processing due to increased size of message
2. WS Security is extension of SOAP to apply security to SOAP web services. Uses **XML signature & encryption** to provide end to end security. Security features in header of SOAP message. Non repudiation associates actions/changes to a unique individual. It achieves **Non repudiation** through Digital signatures. Alternatively non repudiation is achieved through writing transactions to an audit trail that is subject to security safeguards
3. WS-Security Standard specifies how to encrypt & sign SOAP messages and attach security tokens. Signature formats supported are **X-509, Username/password, SAML assertions, Kerberos and Custom tokens**. WS-Security works in app layer. It is used in E2E security especially when intermediary i.e. App level proxy is there which will terminate TCP connection. SOAP may support alternate bindings i.e. JMS or SMTP, where E2E security may be required. It might be required for service to know about end user, if service is relayed by a **(HTTP-) reverse proxy**. In such case A WSS header could be used to convey the end user's token.
4. A **reverse proxy is proxy server** that retrieves resources on behalf of client from other servers. These resources are returned to client as if they originated from original Web server. Reverse proxies hide existence of origin server, enabling firewall to protect against common web attacks, like DoS or DDoS. It can also act as load balancer
5. **WS-Addressing** is a standardized way of including message routing data within SOAP headers. A message using WS-Addressing may contain its own dispatch metadata in SOAP header.
6. Axis2 is an **implementation of JAX-WS**. JAX-WS implementation in JDK really is basic soap only. Complex WS-\* things like WS-Security, WS-RM, WS-Policy, etc..., are available in alternatives like CXF or Metro or Axis2. CXF has a good Spring support as well as OSGi support. Axis2's proprietary ADB data binding is faster than CXF. Axis2 supports multiple languages—including C/C++ version and Java version. It has more options for data bindings. In Axis2 **rampat/rahas modules** provides WS Security

**Spring Boot**

1. Spring Boot is an IO project build on top of spring framework for **faster web application development and bootstrapping through embedded application server**. It allows **components to be exposed as REST web services as per micro service** model thus avoiding need for redeployment after component maintenance activities.
2. When spring boot app starts it looks for “**application.Properties” file** in application root path.
3. Spring boot enables **cache manager implementation to be configured in pom.xm**l file. While **ConcurrentCacheManager** is default, others like GuavaCacheManager will have dependencies that need to be added to pom.xml file
4. Unit testing for web services is performed through **MockMvc** class from spring mock packages that **simulate mock http interactions.** It is initialized with application context in **setup() method**. Application context is inserted into bean class through auto wiring. Use of **Mockito library** for unit testing web services, isolates a class for testing by providing **mock objects**, for instance mocks a database connection or server file required for running a method
5. **Integration with spring security**: spring boot provides a security configuration adjustable through application.properties. All security events are **published through ApplicationEventPublisher** for which listeners can be created.
6. **Spring boot actuator suite** offers capabilities such as health check, metrics, traces which can be enabled through application configuration. **Actuator endpoint properties** can be configured in application.properties file. Security events for actuators are published through AuditService

**Securing spring Micro service**

1. Protocols used for communication with Micro service include HTTP (SOAP, REST), Apache Thrift RPC framework, Google RPC and AMQP. **Micro service security is said to cover all above protocols** with which Micro service communicates including commonly used HTTP
2. Every request to a Micro Service should include **a security token**. Token Specifications define token format, how it is to be obtained, libraries used for authentication & authorization and information present in token. **Standard token formats** include Kerberos on Windows (AD is based on Kerberos), SAML based on XML and JWT using JSON. While SAML Token requires a SAML server, Kerberos requires a Kerberos server, **JSON Web Token** (**JWT) token can be obtained from anyone**. JWT can be effectively used with SSO
3. **JWT token is a json object** on its own, signed and formatted as **JSON Web Signature** document (JWS) or encrypted as **JSON Web Encryption document** (JWE). It can be put in URL/HTTP header. JWT token format comprises of header containing a JSON web algorithm, a payload & signature. **JSON web algorithm specification** registers cryptographic algorithms used with JSON web signature. JWT token can be obtained from **OAuth/Openid connect server** using Openid connect protocol which is built on top of oauth2, JWT & TLS.
4. **Security programming model** involves below steps
   1. Micro service gets a request which includes JWT token & it checks if token is valid
   2. Token has information on what user is authorized to do
   3. Micro service uses info in token to make **access decisions**

**Persistence Implementation in Web Application**

1. Using Database connection pool: **Apache commons connection pool library** provides Data Source that supports connection pooling; it can be incorporated by including **commons-dbcp2 module** under apache.commons to path or to Maven POM file.
2. **Spring ORM module** provides different **entity manager factories** making configuration of connection to JPA implementation easier. Entities need to be annotated with **JPA annotations** to enable persistence provider to persist them. An entity class should be annotated with **@Entity** with primary key annotated with **@Id** to enable persistence
3. A Database Access object (DAO) class is required to **access entity object** & should be annotated with **@Repository**. This makes it an **injectable spring bean** and also **translates exceptions** **from different JPA frameworks** into common set of exceptions callable within service layer.
4. Dependency Injection of DAO class is **testable with JUnit framework**. Test class is annotated with **@RunWith** and **@ContextConfiguration**. @RunWith provides a **custom test runner**, allowing plugging in a spring test runner, that allows injection of spring bean within JUnit test & configuring bean using configuration file. @ContextConfiguration allows specifying **a spring bean configuration** XML file to configuring beans for test case. **Component-scan** declaration is used in XML file to enable dependency Injection

*@RunWith(SpringJUnit4ClassRunner.class)*

*@ContextConfiguration(“classpath: spring/business-config.xml”)*

*public class AccountRepoTest{*

*}*

1. Data source is used to connect to database and is declared in XML as a bean. Entity manager is then setup for Dependency injection of Data source

**///Data source bean definition**

<bean id="basicDataSource" class="org.apache.commons.dbcp.BasicDataSource">

<property name="driverClassName" value="com.mysql.jdbc.Driver" />

<property name="url" value="jdbc:mysql://localhost/my\_database" />

<property name="username" value="my\_username" />

<property name="password" value="my\_password" />

**//connection pool configuration**

<property name="initialSize" value="3" />

<property name="maxActive" value="10" />

</bean>

**///Entity manager bean definition**

<bean id="myEmf" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">

**//Data source bean is dependency injected**

<property name="dataSource" ref="dataSource" />

**//Specify hibernate as JPA vendor implementation**

<property name="jpaVendorAdapter">

<bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter" />

</property>

**//Set important hibernate properties**

<property name="jpaProperties">

<props>

<prop key="hibernate.hbm2ddl.auto">**create-drop**</prop>

<prop key="hibernate.dialect">org.hibernate.dialect.MySQL5Dialect</prop>

</props>

</property>

**//Location of hibernate entity files**

<property name="packagesToScan" value="org.baeldung.persistence.model" />

</bean>

1. **Entity Manager** and its Persistence Context should be **transaction scoped**, i.e. entities attached during transaction demarcation become detached after transaction is completed. This is done through **Transaction annotation driven command**, requiring **TX namespace**. By default transaction manager is auto wired based on name. **@Transactional and @Rollback annotations** should be used to make test methods transaction scoped

**//allowing spring to scan for transaction demarcations**

<tx:anntation-driven>

**//Entity manager is dependency injected**

<bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager">

<property name="entityManagerFactory" ref="myEmf" />

</bean>

@Before

@Transactional

@Rollback (false) **//in case of failure changes are rolled back**

public void Setup

{

}

1. To Set up access to database using Spring and hibernate, need these steps
   1. A **Hibernate configuration XML** file should be provided as **context configuration** argument in web.xml so that spring processes it. Location can be specified in **/resources/spring folder**
   2. Data source & **EntityManager Factory beans** are defined in Hibernate configuration XML file. **Data source bean** refers to properties which mainly include database driver, JDBC connection URL and other props i.e. username, password, max connections etc.
   3. **Entity Manager Factory bean** definition takes in data source bean as argument, which gets auto injected. Other props include a **JPAVendorAdapter**, in this case HibernateJPAVendorAdapter, **JPAProperties** collection & Packages to scan location for JPA entities. JPAProperties have **key value pairs** with basic props as hibernate.hbm2ddl.auto, hibernate.show\_sql. **Packages to scan location** specifies folders for objects annotated as entity and creates tables in DB

**Creating Web application with Spring MVC STEPS**

1. Create a **maven project in eclipse**, a simple prototype with web app project name in both GroupId & ArtifactId fields
2. Use **bill of materials dependency** to specify dependency management section from spring to avoid version conflicts. This will auto **manage transient dependencies** with no need to provide version for spring dependencies in POM file. Dependency Management section is added to pom file
3. **Minimum dependencies** required for spring mvc web app are **spring-context** which includes many spring modules as transient dependencies & **spring-webmvc** modules for developing MVC style apps
4. Inclusion of **build plugins**: maven includes a JAVA compiler plugin, but it complies with source & target of old Java version (1.5). To include higher versions of compiler use **maven compiler plugin**. This configure maven compiler to version 1.6 or above. Next is **tomcat embedded server** plugin
5. Create **webapp/WEB-INF directories** within src/main, a standard folder for web artefacts. Container tries to find descriptor file in this location **(web.xml).** A properly formed XML file with namespaces
6. Create a **front controller**; declare a servlet that maps to **springs dispatcher servlet**. Spring uses this front controller to map request to **application controller classes**. Dispatcher servlet is mapped to then any type of **URL (/) path is provided in mapping.** This configures web app context for spring mvc
7. Next create a **configuration file for web mvc project**, this is created in web app directory. Inside it has standard namespace provided by spring. Specify **component-scan package** and annotation driven directives
8. **InternalResourceViewResolver** bean is needed to allow spring to **map views** it returns to files. create a directory where **resolver will look for view files** in WEBINF folder
9. **Controller class** is required to map URL request that generates this view. It will be annotated **@controller**. Map classes method relative to context path with **@request-mapping**
10. To execute app in eclipse, use **maven run configuration** with target **tomcat:run**

**Testing Web Application using Mock MVC**

1. To test MVC controllers, spring has to be set up such that when json is sent to it via web, it is converted to java object, & when response sent back to client, its body contains json. **Jackson** does this conversion behind scene. Spring sets up a **message converter** once Jackson modules are included in class path. Modules jackson-core, jackson-annotation & jackson-databind are to be added to maven repository.
2. Include **spring test module** in path which will setup **mock mvc object** and allow sending requests to controller. Mock mvc creates mock for components in servlet API to enable testing, hence **servlet api module** should also be included in class path with **scope test**
3. Mock MVC is used to simulate request to a controller. To test controller **mock MVC instance** is required and **Mockito** is used to create it. **@InjectMocks** creates instance of controller class and injects **mocks** or proxy objects.
4. **Init() method** of **MockitoAnnotation** class is used to **inject controller in Mock MVC object**. Next mock MVC object is created which is built using **mockMVCBuilder class**, having a static function that allows building 2 kinds of mock MVCs. It also allows specifying web app configuration file for mock mvc environment. It creates some default message converters allowing some annotations recognized

MockitoAnnotations.*initMocks*(**this**);

mockMvc = MockMvcBuilders.*standaloneSetup*(controller).build();

1. **Mock MVC request builder** class allows building request through method chaining. Result builders are used to **validate result** using message chaining
2. **Mock MVC perform() method** to make a **web request** built using method chaining ie setting content type, media type etc. It is directed to controller method annotated **@RequestMapping**. Result handler is used to check for certain kinds of output. It can also be used to print request & response using print() method for debugging purpose. When a request is sent, mock MVC creates a **mock http servlet request** & mock http server response.

mockMvc.perform(*post*("/test12")

.content("{\"title\":\"Test Blog - Post\"}")

.contentType(MediaType.***APPLICATION\_JSON***))

.andExpect(*jsonPath*("$.title", *is*("Test Blog - Post")))

.andDo(*print*());

1. Message converters are used implicitly to convert b/w json & java object. Through **annotations,** hint is provided to mock MVC that object is to be converted using **http message converters.** Another way is to use **responseEntity** with a class type as parameter. **getters/setters are required** on class so that Jackson can read class variables and convert b/w json representations. Json conversion is also setup using **@RequestBody and** **@responseBody annotation.** Through @RequestBody it is specified how to receive JSON object when user posts to controller

@RequestMapping(value = "/test12", method = RequestMethod.***POST***)

**public** **@ResponseBody** BlogEntry Test12**(@RequestBody BlogEntry blog**){

**return** blog;

}

@RequestMapping("/test11")

**public** **ResponseEntity**<Object> Test11(){

BlogEntry blog = **new** BlogEntry();

blog.setTitle("Test Blog Entry -11");

**return** **new** **ResponseEntity**<Object>(blog, HttpStatus.***OK***);

}

1. **json-path module** is used to check various parts of JSON object for correctness i.e. property value validation

*mockMvc.perform(post("/test12")*

*.content("{\"title\":\"Test Blog - Post\"}")*

*.contentType(MediaType.APPLICATION\_JSON))*

*.****andExpect(jsonPath("$.title", is("Test Blog - Post")****))*

**Spring Restful Web service**

1. **REST web services** allow systems to access web resources using a uniform set of stateless operations. Other web services expose custom sets of operations ie. WSDL, SOAP
2. **Web Service Discovery** enables finding suitable web services for a given task. Publishing a Web service involves making software artifacts accessible to potential consumers through end points. WSDL provides machine readable description of web service. **WSDL 2.0** offers better support for RESTful web services, as it accepts binding to all HTTP request methods apart form GET/POST
3. HATEOAS enables clients to interact with a network application dynamically through **hypermedia links**. A REST client needs no prior knowledge about how to interact with an application. All actions the client can take are discovered through **web resource representations** returned from server. Client transitions through application states by selecting from the links in a resource representation.
4. **Hypermedia**, an extension of the term hypertext, includes graphics, audio, video, plain text and hyperlinks. Adobe Flash, Macromedia Authorware etc. are used to create stand-alone hypermedia applications.
5. Spring HATEOAS provides APIs making it easy to create REST representations based on HATEOAS principle. It also provides a way to convert to a variety of hateoas formats. To create a framework for adding links in response from web service, a **resource** is created extending from **ResourceSupport class** of Hateoas library. This web service resource is a wrapper over entity additionally containing hypermedia links. An entity is then accessed through this resource class rather than being directly accessed from within controller.
6. **ResourceAssemblerSupport Class** of Hateoas enable conversion between resources and entity objectsClass

**Design of Blogging Web Application**

1. Web application architecture **comprises of entities** or POJO classes with properties. Entities are
   1. **Account** (containing Name, ID, Password and other details),
   2. **BlogEntry** (containing Blog ID, Blog Title, other metadata, Reference to Blog)
   3. **Blog** (containing ID, Owner, Blog content).
2. At **service layer,** service classes provide interface methods to access entities. A Controller client **uses service layer to access entities,** creates a web resource and returns to client
3. Service layer also contains **exceptions and Utility classes**. Exceptions are mainly **Runtime Exceptions** as they are easier to capture in an appropriate layer without re-throwing.
4. **Controller** receives a web service request and invokes a method using service layer to operate on entity.
5. Rest layer contains **controllers and web service resources.** Resources extending from **ResourceSupport** class of Hateoas library. **Dependent hyperlinks** are returned to web client.
6. In testing **web request is simulated using Mockito** which creates service layer as mock object and injects into controller. Mockito supports **argument captor** feature
7. Rest layer also has **exception classes** annotated with **@ResponseStatus** having an HTTP status value.
8. To verify test cases with exceptions, it is required to **throw service layer exception as rest layer exceptions** that will change http response sent to client. Since rest layer exceptions are annotated, **spring generates a response** using exception **http status code** specified in annotation

**Logging using Slf4j**

1. Slf4j provides abstract layer so that any logging framework can be used. By default eclipse is using commons logging framework. To add Slf4j commons logging framework should first be removed. This gets downloaded as part of dependencies of spring-core & commoms-dbcp2 artifacts. Using exclusions in maven under spring core and commons dbcp2 dependencies wile remove commons logging JAR from class path

<dependency>

<groupId>org.apache.commons</groupId>

<artifactId>commons-dbcp2</artifactId>

<version>2.1.1</version>

**<exclusions>**

**<exclusion>**

**<groupId>commons-logging</groupId>**

**<artifactId>commons-logging</artifactId>**

**</exclusion>**

**</exclusions>**

</dependency>

1. Include logback classic framework which implements slf4j directly along with slf4j dependencies in pom file

<!-- logback-classic for Slf4j -->

<dependency>

<groupId>ch.qos.logback</groupId>

<artifactId>logback-classic</artifactId>

<version>1.2.3</version>

</dependency>

<!-- Slf4j /jcl-over-slf4j -->

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>jcl-over-slf4j</artifactId>

<version>1.7.24</version>

</dependency>

1. Add a log back configuration file in resources folder
2. Exception Handling in web application
   1. Exception Handling class which acts as a controller is annotated with @ControllerAdvice. This advice allows using same exception handling techniques across entire application, not just to an individual controller
   2. @ExceptionHandler on a class method decides which “view” should be returned back if exception is raised
   3. @ExceptionHandler(NoHandlerFoundException.class)
   4. public ModelAndView HandleNoHandlerFoundException()
   5. Annotated method is invoked if no web resource is found for a URL provided, redirecting to returned JSP view
   6. @ExceptionHandler(ProductNotFoundException.class)
   7. public ModelAndView HandleNoProductFoundFoundException()
   8. For a custom exception, User defined exception calss in passed as argument to @ExceptionHandler, so annotated method is called whenever custom exception occurs.
   9. For a custom exception, exception is explicitly thrown by a method
   10. For a general exception such as mal formed URL, Exception class is passed as argument to @ExceptionHandler
   11. @ExceptionHandler(Exception.class)
   12. public ModelAndView HandleGeneral(Exception e)
   13. Cause of exception is available through Exception class argument to methoid
   14. Stack trace is also printable on web page using printStackTrace method of Exception argument
   15. StringWriter sw = new StringWriter();
   16. PrintWriter pw = new PrintWriter(sw);
   17. ex.printStackTrace(pw);
   18. Stack dump gets out of web page and it is required to scroll horizontally to read. To make it wrap around css property should be applied in rendering page
   19. <blockquote style="**word-wrap:break-word**">
   20. ${errorDescription}
   21. </blockquote>
3. JSTL structure <c:choose> -- <c:when> -- <c:otherwise> is akin to JAVA switch statement, dynamically returning content to be displayed on JSP page
4. <c:choose>
5. <c:when test="${product.quantity < 1}">
6. <a href="javascript:void(0)" class="btn btn-success disabled"><strike>
7. <span class="glyphicon glyphicon-shopping-cart"></span> Add to Cart</strike></a>
8. </c:when>
9. <c:otherwise>
10. <a href="${contextRoot}/cart/add/${product.id}/product" class="btn btn-success">
11. <span class="glyphicon glyphicon-shopping-cart"></span> Add to Cart</a>
12. </c:otherwise>
13. </c:choose>
14. javascript:void(0) in href makes link unavailable. Bootstrp calss “disabled” makes button “add to cart” disabled. <strike> tag is used for a strike across button
15. mRender function is used with jQuery data table to display alternate values in a column
16. {
17. data : 'quantity',
18. mRender : function(data, type, row) {
19. if (data < 1) {
20. return '<span style="color:red">Out of Stock!</span>';
21. }
22. return data;
23. }
24. }
25. If col data is LT 1 “Out of Stock” is retruned. Row parameter in mRender function represents entire row and used to access other other column values in data table ,
26. To use bootstrap classes in web page, bootstarp css and script files should be included either form CDN or from resource folder in web application after downloading bootstrap.css and bootstrap.js files
27. <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css">
28. <script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js"></script>
29. A web page is divided into 12 columns, through classes col-md-\* or col-xs-\* etc. columns occupied by an html element can be specified. Col-md-offset-\* specifies starting offset in columns of element
30. <div class="row">
31. <div class="col-md-offset-2 col-md-8">
32. <h4>Product Management</h4>
33. </div>
34. </row>
35. Element “Product Management” will start after 2nd column in webpage and will extend up to 8 columns in length, leaving 2 columns to right of page.
36. Spring form avoids need for conversion between values coming from controls to their respective types. It is done by spring framework. Spring form is part of Spring MVC and gates added to JSP page with taglib directive.
37. <%@taglib prefix="sf" uri="http://www.springframework.org/tags/form"%>
38. @ModelAttribute auto links spring form to model properties. Model is returned through MVC controller. Each field of model property maps to a spring form element
39. <sf:form class="form-horizontal" modelAttribute="product">
40. <div class="form-group">
41. <label class="control-label col-md-4">Enter Product Name: </label>
42. <div class="col-md-8">
43. <sf:input type="text" path="name" class="form-control" placeholder="Product Name" />
44. </div>
45. </sf:form>
46. Product is returned as a property of ModelAndView object from controller method
47. **@ModelAttribute** at method level is use to populate certain model property values in form such as in select control.

@ModelAttribute("categories")

public List<Category> getGategories()

{

return categoryDao.getCategoryList();

}

1. The list is accessible on spring form in JSP through “categories”
2. Attributes action & method are used on spring form to handle submission.
3. <sf:form class="form-horizontal" modelAttribute="product" action="${contextRoot}/manage/product" method="POST" >
4. Action attribute tells URL to which request is to be submitted using specified HTTP method.
5. File upload to a web application is supported through Mulitpart which is in spring-web module. Spring form should hgave and enctype attribuite as multipart/form-data.

<sf:form class="form-horizontal" modelAttribute="product" method="POST"

action="${contextRoot}/manage/product**" enctype="multipart/form-data">**

1. A multipart-resolver is configured in dispatcher-servlet.xml

*<bean id="multipartResolver"*

*class="****org.springframework.web.multipart.support.StandardServletMultipartResolver****" /*>

1. Configure file parameters in web.xml for file upload

*<servlet>*

*<servlet-name>dispatcher</servlet-name>*

*<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>*

***<multipart-config>***

***<max-file-size>20848820</max-file-size>***

***<max-request-size>418018841</max-request-size>***

***<file-size-threshold>1048576</file-size-threshold>***

***</multipart-config>***

*</servlet>*

1. Create a file upload utility class which will contain ABS file path (on local m/c) and real path (on tomcat server). Use **transferTo() method** of multi part file and create a new file in both locations

**Web Development using HTML, JSP, JQUERY, Boot strap & CSS**

1. Online Shopping Application developed using Spring MVC, JSP, Hibernate, HTML, JQuery, Boot strap CSS
2. ID Selector in JQuery to select an HTML element by ID and perform dynamic operations
3. HTML Page division into Row & Columns
4. Use of **JSTL Core** expressions to assign and use variables in JSP file
5. Use of **JSTL for each loop** to dynamically assign names to items displayed in Side menu
6. Use of JSTL expressions to replace **static HTML items** with dynamic ones
7. Dependencies for using **hibernate with spring** in POM file
8. DI for **hibernate repository** implemented into MVC controller. **Spring context** dependency required for using @Repository annotation
9. Include a **project dependency** in another project’s POM file. Dependency of shopping backend project was included in Online shopping frontend
10. Retrieve data in form of JSON from MVC endpoints by including Jackson dependencies in POM file
11. Annotation **@ResponseBody** on controllermethodtriggers JSON conversion of a retrieved JAVA object
12. Customization of public **boot strap templates**, copying locally & changing references to CSS/JS file links through JSTL expression variables
13. Conditional rendering of JSP file based on Boolean expression. Selection of a hyperlink would render a different JSP page
14. Use of bootstrap **Bread crumb class** for menu design

**Database Migration from SQL to NoSQL**

1. Why want to migrate from SQL to NoSQL? NOSQL can handle large volumes of data
2. NoSQL data structures
   1. Key-Value Store (Redis, DynamoDb)
   2. Key-Document (Mongo, CouchBase), provides ability to operate with nested & hierarchical structures
   3. Column-Family store allows each record to consist of independent no. of columns i.e. Cassandra can have thousands of columns per record
   4. Graph DBs represents entities & relationships b/w them (Social graphs, road maps etc) i.e. NeoDB
3. Migration Objectives can be Better performance, Scale current solution, Process a variety of data etc.
4. Migration considerations: Learn about Application Traffic, How Queries look like i.e. Simple lookups, inner requests, traversals, BI/Analysis etc.
5. Learn about dataset:
   1. Is it simple KV, not important what is stored in V
   2. Is data structured or semi structured
   3. Is there lot of nested/hierarchical data
   4. Is data graph oriented i.e. addresses what is liked by friend of friends
6. What can be sacrificed?
   1. Which data doesn’t require Strong consistency i.e. at any point of time all users of DB see data in same state
   2. Transactional guarantees i.e. multiple operations as a single unit
   3. Joins
7. Types of storage requirements
   1. Fast storage for lookups, non-critical data, short transactions
   2. RDBMS suitable for data fitting into single server
   3. Document oriented store for hierarchical data & aggregate oriented read writes
   4. Graph oriented storage for traversal queries, social relations etc.
   5. Scalable storage for big data processing
8. Issues
   1. Data duplication or demoralization i.e. Data required for a join should be on same server
   2. Aggregates involve embedding a data structure into another
   3. Updates on duplicated data could be a problem. Also querying across aggregates is heavy & complex
9. Incremental background synchronization through Tungsten synchronizer that allows tracking changes in RDBMS via transaction log and apply them to NoSQL storage e.g. User profiles in MySQL are synced with AeroSpike via property configured Tungsten replicator
10. Tungsten Replicator is an **open source replication engine** supporting different types of extractor. Data can be extracted from MySQL, Oracle and applied to transactional stores or NoSQL stores i.e. Mongo DB. During replication, Tungsten Replication assigns data a unique global transaction ID. This enables data to be exchanged between different databases and different database versions. During replication, information can be filtered and modified, and deployment can be between on-premise or cloud-based databases. Tungsten Replicator includes support for parallel replication for performance. It also uses advanced topologies i.e. fan-in, star and multi-master for cross-site deployments.

**Mongo DB**

1. Mongo DB is a cross-platform document-oriented NoSQL database. It uses JSON-like documents with schemas. Mongo DB supports **field, range queries, regular expression searches**. Queries can return specific fields of documents and also include user-defined JavaScript function. Can also return a random sample of results. Fields in a Mongo DB document can be indexed with primary and secondary indices.
2. Mongo DB provides **high availability** with replicas. Each replica set member may act in primary or secondary replica role at any time. All read/writes are done on the **primary replica** by default. When a primary replica fails, the replica set conducts **election process** to determine which secondary should become the primary. Secondary can optionally serve read operations, but that data is only **eventually consistent**.
3. Mongo DB scales horizontally using sharding. Data is split into ranges (based on the shard key) and distributed across multiple shards. Shard key is hashed to map to a shard. Mongo DB can run over multiple servers.
4. Mongo DB supports **grid file system**. It exposes functions for file/content manipulation to developers. GridFS divides a file into chunks, and stores each as a separate document.
5. **Aggregation framework** enables users to get results as obtained through SQL GROUP BY clause. Aggregation operators can be pipelined. $lookup operator joins docs from multiple documents
6. **JavaScript** can be used in queries, aggregation functions and sent directly to the database for execution.
7. Mongo DB supports **capped collections** which maintain insertion order and behave like circular queue when size is reached
8. With Mongo DB 3.0 concurrency control is implemented at the **collection level** for MMAPv1 storage engine. MMAPv1 is MongoDB’s original storage engine based on **memory mapped files**. It supports workloads with high volume inserts, reads, and in-place updates. Since Mongo DB 3.2, **WiredTiger storage engine** is the default storage engine. WiredTiger approach supports **document level locks** enabling atomic operations on document level instead of collection level
9. Mongo DB by default records all modifications to an **on-disk journal**. With default configuration it writes to the data files on disk every 60 seconds while to the journal files every 100 milliseconds. From 3.0 onwards Mongo DB uses **Power of 2 Sized Allocations** i.e. every document is stored in a record which contains the document itself and padding which allows document to grow with updates minimizing likelihood of reallocations.
10. Different allocation strategies support different workloads: Power of 2 allocations is more efficient **for insert/update/delete workloads**; while exact fit allocations is ideal for collections without update/delete workloads.
11. For **Read intensive configuration** in Mongo Driver Read Preference Modes, Set write concern to majority & read preference to secondary. Writes are slow but read throughput increases
12. Mongo DB profiler can be configured to only write profile information about queries that took longer than a given threshold. Helps finding out needs of DB
13. Collections in Mongo DB are similar to tables in RDB

*db.creteCollection('customers'); //creates collection named customers from mongo shell*

1. Documents can be inserted into collection. An **\_id field** gets auto added. Multiple documents can be added as an array. A new field can be added on fly for a new document

*db.customers.insert({first\_name: 'John"}) //inserts doc with* ***first name as “John”***

*db.customers.insert([{first\_name: 'John", last\_name: "Doe"}, {first\_name: 'John", last\_name: "Doe", gender: "Male"}]) // inserts doc with* ***first name as “John” and other fields***

*db.custoemrs.find().pretty(); ///fields displayed with* ***formatting*** *on shell*

1. To update a field of a document uniquely, document id should be specified, else all matching documents on filed get updated

db.customers.update({first\_name: "John"}, {first\_name: "John", last\_name: "Doe", gender: "male" }) // Finds all docs with first name as John and update them with adding gender field

1. Using **operator $set**, keeps existing fields as it is and adds new fields

db.customers.update({first\_name: "John"}, {$set:{gender: "male" }}) //retains all fields, adding gender field

1. Using **$inc operator** numeric fields can be incremented

db.customers.update({first\_name: "John"}, {$inc:{age: 2}}) //increments age filed by 2

1. A field can be removed using **$unset operator**

db.customers.update({first\_name: "John"}, {$unset:{age: 1}}) //removes age filed

1. **upsert** is used to insert if document is not found

db.customers.update({first\_name: "Mary"}, {first\_name: "Mary", last\_name: "Samson"} {**upsert: true**}) //insert document if not found

1. A field for a particular doc can be renamed using **$rename operator**

db.customers.update({first\_name: "John"}, {$rename:{"gender": "sex"}}) //gender field is renamed to sex

1. Document is deleted through remove command

*db.customers.remove({first\_name: 'John"}) //remove all docs with name john*

1. A parameter **justOne** will restrict removal to only first item found

db.customers.remove({first\_name: 'John"}, {justOne: true}) //remove first doc found

1. **$Or operator** is used to retrieve multiple documents with one or more matching criteria

*db.customers.find({$or:[{first\_name: "sharon"}, {first\_name: "john"}]).pretty();*

*db.customers.find({gender: "male"}).pretty();*

1. **Operators $gt, $lt, $gte, $lte** return matching docs with less or greater constraints

*db.customers.find({age: {$lt:40}}).pretty(); //return all younger than 40*

1. Search on a **subfield**

*db.customers.find({"address.city" : "Boston"}) //retruns docs with city Boston*

1. Mongo DB clients may require a key in quotes
2. Search **on array element**

db.customers.find({memberships : "mem1"}) returns docs having mem1 as one of memberships

1. Sorting is done on retrieved docs on specified field

db.customers.find().sort({first\_name: 1}) //sorts docs in ascending order on first name

db.customers.find().sort({first\_name: -1}) //sorts docs in descending order on first name

1. **Aggregate methods** applied to retrieved docs

db.custoemrs.find().count(); //returbs count of records

db.custoemrs.find({gender: "male"}).count(); //returns total docs having male gender

db.custoemrs.find().limit(5); // displays 1st 5 docs

db.custoemrs.find().limit(5).sort({last\_name: 1}); //sorts first 5 docs on last name

1. **foreach command** can be used to iterate over documents and apply operations through java script

db.custoemrs.find().forEach(function(doc){print("Customer Name: " + doc.first\_name)});

1. With CAP theorem Mongo DB sits on side of partition tolerance & consistency. It favours consistency over availability. Any JSON object can be put as a doc inside mongo DB.
2. Indices can be created on any field or combination of fields
3. Database Documents & collections are key terms used. DB contains collections that contain docs. Data cannot be moved across collections in different DBs
4. **Mongoimport tool**, can push comma separated data to mongo database. **mongomtimport** is similar to mongoimport but also allows writing plugins & perform transformations i.e. encrypt fields, hash, lookup.
5. **Firehose**, a framework allowing java modules to plug into it and allow push data & do transformations

**SQL & HQL Queries**

1. Select \* from employee2 **left join** address2 on employee2.address\_Id = address2.addressId

*Joins tables’ employee2 & address2 on columns address\_id in employee2 & addressId in address2. Prints records in employee2* ***even with no matching address***

*Equivalent HQL query*

*FROM Employee2 emp left join fetch emp.address”*

1. select \* from employee2 left join address2 on employee2.address\_Id = address2.addressId where employee2.Id = 1;

*Performs a left join but prints only for employee Id 1;*

Equivalent HQL query

FROM Employee2 emp left join fetch emp.address where emp.id = :empId";

1. Select emp.name, addr.city from Employee2 emp left join Address2 addr on emp.address\_Id = addr.addressId where emp.id > 1

*Joins tables’ employee2 & address2 on columns address\_id in employee2 & addressId in address2 and retrieves fields name from Employee and City from address*

Equivalent HQL

1. Select emp.name, addr.city from Employee2 emp left join emp.address addr where emp.id > :empId