**Basics Java & Spring Boot Microservices**

**Pre-requisites:**

* Participants must have good practical knowledge on Core Java especially like
  + Fundamentals - Control Statements, Loops, Arrays, Operators
  + OOPS concepts - Inheritance, Encapsulation, Abstraction, Polymorphism
  + Exception Handling
  + Collection Framework and Map
  + File Handling
  + Lambda Expression
* Participants can go through the below Udemy course

https://www.udemy.com/course/java-the-complete-java-developer-course/

**Training Methodology**

* Training will be on TDD & DDD approach
* The training will be based on a real-world use case and will be built from ground up in an iterative manner.
* All the sessions will be workshop based, where participants will be coding along with the instructor for the entire session.
* Concepts learnt will be added to the case study project in an iterative manner so that it must be an application at the end of the training program.

**Deliverables**:

* Codebase developed during workshop will be shared with the participants
* Reference materials will be shared with the participants

**Hardware & Software:**

* JDK 8 or later
* Eclipse IDE for enterprise / STS
* Database - PostgreSQL
* VS Code
* Postman
* Open internet connection for Maven and Node Module downloads

**Course Contents**

**Self Paced Learning (Mandatory)**

**Udemy Course link**

**https://www.udemy.com/course/java-the-complete-java-developer-course/**

**Topics to be learnt**

* Java Fundamentals
  + Operators
  + Conditional statements - if, if-else, switch
  + Loops - for, while, do-while
  + Arrays
  + break & return
* Encapsulation
  + private, public keywords
  + setters & getters method
* Inheritance
  + extends
  + super
* Polymorphism
  + overloading
  + overriding
* Abstraction
  + abstract class
  + interface
* Inbuilt classes
  + Object
  + String
  + LocalDate & LocalTime
* Functional Interface
  + Lambda expression
  + Method reference
* Exception Handling
  + try, catch, finally, throw & throws
  + Custom Exceptions
* Collection Framework & Map
  + List - ArrayList, LinkedList
  + Set - HashSet, TreeSet, LinkedHashSet
  + Map - HashMap, TreeMap, LinkedHashMap
* File Handling
  + Input & Output Streams
  + Reader & Writer

**Instructor Led Training - Basics**

**Day 1**

* **Pre-requisites** 
  + Collection Framework
  + Streams & Lambda expressions
* **PostgreSQL Database**
  + Queries
  + Datatypes
  + Functions & Operators
  + Type conversion
  + Parent and Child tables relationship
  + Indexes
  + Full Text Search
  + Performance Tips
  + Transaction Isolation

**Day 2**

**Spring & Spring Boot**

* **What is Spring**
  + Features of Spring
  + Dependency injection
  + Bean Configurations
  + Application Context
  + ClassPathXmlApplicationContext
  + Setter injection
  + Constructor injection
* **Spring Boot**
  + Advantages of Spring Boot
  + @SpringBootApplication
  + Auto-Configurations
  + Spring Boot starters
  + Component scanning
  + Spring boot configuration file
  + YAML
  + Properties

**Day 3**

* **Configuring Spring Boot Application**
  + Adding Jetty server
  + Adding Undertow server
  + Creating an executable Jar
  + Creating a WAR
  + Deploying to the external server
  + Adding Dev-tools
  + Adding Actuator
* **Spring Boot Web**
  + Front Controller
  + View Resolver
  + Controller
* **Multilayer architecture**
  + View Layer
  + Controller Layer
  + Service Layer
  + DAO Layer
* **Spring Boot with REST**
  + @RestController
  + @RequestMapping
  + JSON & XML data
  + Path parameters
  + Request body
  + HTTP status codes
  + HTTP methods mapping

**Day 4**

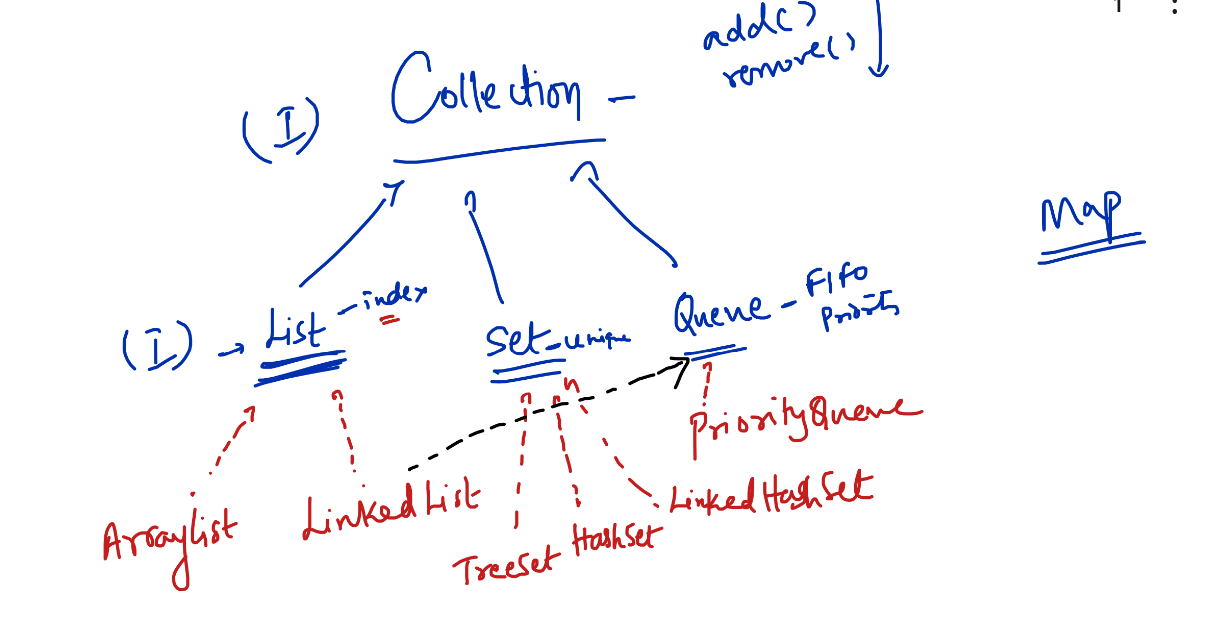
* **JDBC & ORM Framework**
  + JDBC vs ORM
  + Benefits of ORM Framework
  + JPA Specification & Implementations
  + Entities
  + Id & Generators strategy
  + SessionFactory
* **Spring Data JPA**
  + Spring Data JPA
  + Entity
  + Repository
  + CrudRepository
  + JpaRepository
  + Query & Modify
  + Parent & Child mappings

**Day 5**

* **Spring Cloud & Microservices with Spring Boot**
  + Microservices Introduction
  + Why and When to use
  + Monolithic vs Microservice Architecture
  + Benefits of Microservices
  + Challenges in using Microservices Architecture
  + Breaking down a Monolithic app to Microservice app
* **Microservices with Spring** 
  + Spring Cloud & Spring Boot Projects
  + Eureka Server
  + Eureka Client
  + Disabling the client features in Service Discovery
  + Eureka Dashboard
  + Locating the multiple instance of same Microservices

Collection Framework:

Maintain complex data in various formats – it provides many APIs that can manage the complex data like storing in insertion order, sorted order, keeping only unique data



List: It maintains elements in insertion order, it is index based

ArrayList: Maintains elements in contiguous memory address

LinkedList: Maintains elements in non-contiguous memory address

Set: It maintains only unique elements

TreeSet: It maintains elements in sorted order

HashSet: It maintains elements in random order, its retrieval is faster compare to other algorithms

LinkedHashSet: It maintains elements in insertion order

Queue: It is used when you want to remove the element and process it

Priority Queue: It removes elements in sorted order

List sorting: You can sort the elements in the List using the sort method, it takes Comparator as an argument

Comparator<T>: It is a functional interface which has one abstract method compare(T t1, T t2)

class SortById implements Comparator<Employee> {

// returns +ve, 0 or -ve  
 public int compare(Employee e1, Employee e2) {   
 100 – 100 : 0  
 return e1.getId() – e2.getId(); // it can return 0 or -ve or +ve  
 }  
}

list.sort( new SortById() ); // old approach of implementing Comparator.

You can use anonymous implementation to avoid creating multiple classes

list.sort(new Compator<Employee>() {   
 public int compare(Employee e1, Employee e) { …. return int …}  
});

In Java 8 lambda expression is introduced to minimize writing anonymous implementation: it can be applied only for functional interfaces (having only one abstract method)

list.sort( (e1, e2) -> e1.getId() – e2.getId() );  
// sorting in descending order  
list.sort( (e1, e2) -> e2.getId() – e1.getId() );   
// sorting based on the salary  
list.sort( (e1, e2) -> Double.compare(e1.getSalary(), e2.getSalary() ) );   
// sorting based on the date of joining – LocalDate has a compare method  
list.sort( (e1, e2) -> e1.getDoj().compareTo(e2.getDoj()));   
// sorting based on the name – String has a compareTo method  
list.sort( (e1, e2) -> e1.getName().compareTo(e2.getName()));

Java Streams: These are data in the collections which you can process in an efficient way(without changing the existing datastructure) with less code like filtering, sorting, transforming, iterating

In Java Streams we have two types of operations:

intermediate operation: filter, sort, map – they give a new stream so that you can chain the methods

ex: stream().filter().filter().sorted().map()

terminal operation: forEach, collect – they are the last operation in the stream, you can’t chain the stream methods

ex: stream().forEach()

ex: stream().collect()

filter(): it is used to apply the conditions on the stream to get only those data that matches to the condition

sorted(): it is used to apply the sorting logic using Comparator

map(): It is used to transform a stream into another form, like transform employee to string or numbers or to person

collect(): It is used to convert from one collection to another

forEach(): It is used to iterate the stream

Comparable vs Comparator

Comparable is used when you want to apply default sorting technique / natural ordering

ex: String, Integer, Double, LocalDate they all have default sorting technique i.e., ascending order

Comparable must be implemented in the same class that you want to compare, it has a method called compareTo(T t)

Comparator is used to override the default sorting technique, it has a method called compare(T t1, T t2), it is implemented in a separate class

|  |  |
| --- | --- |
| Comparable<T> | Comparator<T> |
| Class Employee implements Comparable<Employee> { …. } | Class Employee { }  class SeparateClass { lambdas for Comparator } |

Comparable implementation within the same class:   
class Employee implements Comparable<Employee> {  
 public int compareTo(Employee emp) { return this.id – emp.getId(); }  
}  
Comparator implementation outside the class:  
Comparator<Employee> compareSalary = (x, y) -> Double.compare(x.salary,y.salary);

Note: If you don’t have Comparable or Comparator and try to sort you get an exception at runtime

list.stream().sorted(): Gives exception if Employee doesn’t have Comparable, it has Comparable then sorts   
list.stream.sorted( (x, y) -> x.id – y.id ): Uses Comparator, if Employee has Comparable then it will be overridden by Comparator

Postgres

Stores all the data in a pgsql\_data, you must create this folder to initialize the database

Steps to perform

* + - 1. creating a folder(done only once): pgsql\_data
      2. Initializing the database(done only once): initdb.exe -D path-of-pgsql\_data -U postgres -W -E UTF8 -A scram-sha-256

>>> it asks you to enter password and also to confirm password : Welcome@123

* + - 1. Starting the database(done whenever you want to start the database):   
         *pg\_ctl -D path-of\_pgsql\_data -l logfile start*
      2. Login to the database with username & password:  
         psql.exe -U postgres

Note: All the above steps are performed in the terminal

Datatypes in postgres

1. int
2. bigint
3. varchar
4. date
5. timestamp
6. json

Creating & connecting to the database

create database npci\_db;

\c npci\_db;

Creating serials: generate numbers automatically starting from 1

create table employee(id serial, name varchar(20));  
insert into employee(name) values(‘Alex’);

Creating sequence: When you want to start from a particular number

create sequence emp\_seq start 500;

Generating the sequence

insert into employee values(nextval(‘emp\_seq’), ‘Alex’);

Parent & Child table relationships:

primary key & foreign key relationships

customer table linked with account table

employee table linked with department table

student table linked with marks table

Full text search

It is mainly used to perform complex searches where it can search the derived words from the base words by ignoring the stop words like to, from, but, between, and

inspire -> inspires, inspiration, inspired, inspirational

friend -> friendly, friends, friendship

satisfy -> satisfies, satisfied, satisfaction

postgres uses tsvector to identify all the base & derived words from the dictionary

In Full text search you will use 2 main functions

1. to\_tsvector(column): It takes the column and identifies all the base & derived words in the dictionary
2. to\_tsquery(text): it takes the text that needs to be searched against the tsvector.

Index: It is used to increase the performance when you are searching

Transaction isolation level

By default transaction is read committed for all the users, but we can create a transaction so that when multiple users are in multiple transactions they don’t see other users modification even if they commit, this is called as repeatable read