###### Self Introduction

Hi, I’m Rajendra Lokhande. I’ve been working as a backend developer at Advanced Infoscan Pvt. Ltd. since November 2023, where I work in the logistics domain with our client, Expeditor. Before this, I was with Tata AIA from April 2021 to October 2023, where I focused on backend systems and server-side work. Earlier in my career, I worked at Bajaj Allianz from January 2019 to November 2020, which is where I started developing my skills in backend development.

Technically I have used Java, Java 8 features, Spring Boot, microservices for developing the project. MySQL Database for Hibernate implementation.



**Project Details**



The Livestock Shipping Management System is a real-time application designed to **streamline the process of loading livestock on to ships, tracking their journey, and calculating transportation costs**. It serves the needs of **port administrators**, **livestock handlers**, and **shipping companies**, providing a **comprehensive solution**

for **managing the logistics of livestock transport**.

**Services that we work on:**

**1. Ship Management Service:**

This microservice allows port administrators to manage and monitor the available ships, including their capacity and schedules. It maintains an up-to-date list of ships in the system.

**2. Livestock Loading Service:**

Responsible for scheduling and managing the loading of livestock on to ships. It tracks livestock types, quantities, and loading schedules. Port administrators can use this service to schedule livestock loading on to ships.

**3. Tracking Service:**

Provides real-time tracking of ships and their cargo (livestock). It records GPS coordinates, routes, and other relevant data to monitor the location and progress of each ship.

**4. Cost Calculation Service:**

Calculates the transportation cost based on livestock type, route, and any additional charges. This service is essential for estimating the cost of each shipment.

**We calculate the cost by using:** base formula ->

Cost = (Base Rate + Route Fee + Additional Charges) \* Livestock Quantity.

**5. Database:**

Utilizes a relational database e.g. MySQL to store information about ships, livestock loading requests, tracking data, and cost-related information.

**What we measuredly focus on during the development: (For who)**

● **Port Administrators:** They manage and oversee the entire process. They can schedule livestock loading, view tracking information, and calculate transportation costs.

● **Livestock Handlers:** Involved in the physical loading and handling of livestock.

● **Shipping Companies:** Utilize the system to schedule livestock shipments and track them in real-time.

●

**So, there are several services in this project & I am involved in implementation of TrackingService**.

**tracking Service involves monitoring and tracing the movement of goods or vehicles from their point of origin to their destination. It provides real-time visibility into the location, status, and condition of shipments throughout the transportation process. This tracking can be done using** **various technologies such as GPS tracking devices, RFID tags, or barcode scanners.**

In this Tracking System, delays or disruptions in transportation can occur due to various factors such as traffic congestion, road closures, adverse weather conditions, or vehicle breakdowns. It's essential to promptly notify relevant stakeholders, such as customers or logistics managers, about these delays to manage expectations and minimize inconvenience. So, I implemented the functionality like **Utilize an email notification service to send timely updates about transport delays to stakeholders. for that we have implemented->**

· **JavaMailSender and TemplateEngine Dependency Injection**:

* The class constructor injects an instance of JavaMailSender, which is necessary for sending emails.

· **ExecutorService for Concurrent Email Sending**:

* An ExecutorService is used to manage a pool of threads for sending emails concurrently. This improves efficiency by utilizing multiple threads, especially when sending notifications to a large number of recipients.

· **sendLateShipmentEmail Method**:

* This method is responsible for sending a late shipment email notification to a recipient.
* It takes two parameters: the recipient's email address (recipient) and details about the late shipment (orderDetails).
* Inside the method, a new task is submitted to the ExecutorService using executorService.submit().
* When you call executorService.submit(), you're submitting a task (in this case, the task of sending an email) to the ExecutorService for execution. This allows the email to be sent asynchronously, meaning it runs in a separate thread or threads managed by the ExecutorService. This prevents the main thread of your application from being blocked while the email is being sent.
* Within the asynchronous task, a MimeMessage is created using JavaMailSender.createMimeMessage(), and a MimeMessageHelper is used to set email properties such as recipient, subject, and content.
* The email content is directly set as a string without using any template. This string includes details about the late shipment, such as the order details. It is set using helper.setText().
* Finally, the email is sent using JavaMailSender.send(mimeMessage).
* Any MessagingException that occurs during the email sending process is caught and logged.

· **shutdownExecutorService Method**:

* This method shuts down the ExecutorService once all email tasks have been completed. It ensures proper cleanup of resources and termination of threads.

**Role And Responsibility**



I play a crucial role in designing, implementing, and maintaining the backend of our Livestock Shipping Management System. My responsibilities involve creating the APIs, business logic, data management, and ensuring the security and performance of the server-side components

1. Requirement gathering and analysis
2. Developing and integrating the RESTful APIs and ensuring communication between diff. services and database using SB & spring data JPA
3. Implements core java concepts like oops, collection fw, multithreading, design patterns
4. Using advanced java concepts for developing application code
5. Used entity, controller, repositories to handle data from db
6. Writing unit tests and collaborating with QA engineers to ensure comprehensive test coverage and timely bug fixes
7. Collaborated with cross-functional teams to deliver high-quality software solutions on time.
8. And deliver code by using Git and documentation.







**Difficulties**



**1.** In our application performance issue happens and when we don’t get ship schedules and their availability in some point while the testing after the

so many check we find out like in our application the shipping companies are third party API with the help this API we have get information of ships availability so we noticed that **Respect the rate limits and usage policies imposed by the API provider. Monitor the rate limits to avoid being blocked due to excessive requests.**

With the help of

**Apache Commons Lang:**

RateLimiter rateLimiter = new RateLimiter(10); // 10 requests per second if (rateLimiter.tryAcquire()) {

// Make the API request

} else {

// Handle rate limit exceeded

}

**Explanation**:

### 1. Rate Limiting with Apache Commons Lang's

* This code implements rate limiting to control the rate at which API requests are made to a third-party service (e.g., a livestock tracking API).
* The RateLimiter allows only 10 requests per second. If there are available permits (i.e., the rate limit is not exceeded), the API request is made.
* If the rate limit is exceeded, the code handles the situation where the request cannot be made immediately.

**Benefits**:

* **Respecting Rate Limits**: By using RateLimiter, you ensure that your application adheres to the rate limits specified by the API provider, preventing excessive requests that could lead to being blocked.**By limiting the** r**ate of requests, it prevents overwhelming the third-party API, ensuring it remains responsive and available.**
* **Prevents API Abuse**: Ensures that the application complies with the usage policies of the API provider, reducing the risk of being blocked or restricted.

After that

2. When making API requests at that time we facing a issue like blocking a application.

Then we ware find out this happen because of synchronous calls and this is also from third party API

**To overcome this problemwe use:**

We use **CompletableFuture** to interact with a third-party API ->

**Create a Method** for Asynchronous API Calls:

import java.util.concurrent.CompletableFuture;

public class ThirdPartyApiClient {

public CompletableFuture<String> callThirdPartyApi() {

return CompletableFuture.supplyAsync(() -> {

// Perform the asynchronous API call here (e.g., using HttpClient)

});

}

// Replace this with your actual API call logic

String apiResponse = makeApiCall();

return apiResponse;

// Simulate an API call (replace with actual API call logic)

private String makeApiCall() {

// Simulate a delay try {

Thread.sleep(2000); // Simulating a 2-second delay

} catch (InterruptedException e) { Thread.currentThread().interrupt();

}

return "API Response Data";

**Explanation**:

* For this we defines a method callThirdPartyApi() that performs an asynchronous API call using CompletableFuture.
* The actual API call is simulated in the makeApiCall() method, where a delay of 2 seconds is introduced to mimic the API response time.

· In this example, the makeApiCall() method introduces a delay of 2 seconds using Thread.sleep(2000).

· This delay simulates the time it takes for the external API to process the request and respond.

· During this simulated delay, the current thread (representing your application) is paused, mimicking the waiting period for the API response.

· After the delay, the method returns a simulated API response data ("API Response Data").

**Why Use It?**:

* **Non-Blocking Execution**: By making the API call asynchronously, the application can continue executing other tasks while waiting for the response, improving overall responsiveness and throughput.
* **Improved Efficiency**: Asynchronous calls allow for better resource utilization, especially in scenarios where multiple API calls are made concurrently.
* **Concurrency**: CompletableFuture allows you to leverage multiple threads to perform API calls concurrently, potentially speeding up the processing of multiple requests simultaneously.
* **Error Handling**: CompletableFuture provides robust error handling and composition features, allowing you to handle exceptions and compose complex asynchronous workflows with ease.

A CompletableFuture is a class in Java used for asynchronous programming, introduced in Java 8 as part of the CompletableFuture API. It represents a future result of an asynchronous computation.

Here's a simplified explanation:

**Asynchronous Programming**: In regular programming, tasks are performed sequentially, one after the other. In asynchronous programming, tasks can be executed concurrently, allowing the program to continue executing other tasks while waiting for certain operations to complete.

**CompletableFuture**: It's a special kind of Future that can be manually completed (or cancelled) by another thread. It provides a way to perform tasks asynchronously and then obtain the result once the task is completed.



**Microservice & Monolithic**



**1.) What is Monolithic?**

Monolithic architecture is a software design approach where an entire application is built as a single, self-contained unit. In this architecture, all the components and functionalities of the application are tightly coupled and run within a single process or a single codebase.

**Disadvantages** of Monolithic Architecture:

● It difficult to manage

● For single change redeployment of whole application

● Difficult to adapt new technology for single functionality

● Single bug may affect a whole application

**2.) What are MICROSERVICES?**

Microservices is an ARCHITECTURAL STYLE and approach to software development that structures an APPLICATION as a collection of small, loose

coupled, and independently deployable SERVICES. Each SERVICE is responsible for a specific business capability and communicates with other services through well- defined APIs.

**Key Feature**: AUTONOMY, LOOSELY COUPLED, SINGLE RESPONSIBILITY, CONTINUOUS DEPLOYMENT AND DEVOPS.

**3.) What is Eureka Server?**

Eureka Server is a component of Netflix's Eureka, which is a service registry and discovery tool used in microservices architecture. In short, Eureka Server helps microservices applications locate and communicate with each other in a dynamic, cloud-based environment.

**Key Feature:** Service Registration -> registers itself using hostname, port Heartbeat Monitoring -> sending periodic heartbeat requests to ensure the availability.

Load Balance -> to distribute the incoming client requests across multiple instances of the same service.

**How you use Eureka Server in your project or explain the steps:**

**1.** Add dependency: **spring-cloud-starter-netflix-eureka-server**

**2.** Add annotation on main class: **@EnableEurekaServer**

3. Configure using application.yml or properties -> Server: port: **8761**

eureka:instance: hostname: localhost client: registerWithEureka: false

fetchRegistry: false serviceUrl: defaultZone:

http://${eureka.instance.hostname}:${server.port}/eureka/

**4.) What is Eureka Discovery Client?**

It's a library or module that helps microservices communicate with the Eureka Server, allowing them to register their availability and discover other services in the same ecosystem.

**How you use Eureka Discovery Client in your project or explain the steps:**

**1.** Add dependency: **spring-cloud-starter-netflix-eureka-client**

**2.** Add annotation on main class: **@EnableEurekaClient**

3. Configure using application.yml or properties ->

**eureka:**

**client:**

**enabled: true**

**serviceUrl:** [**http://localhost:8761/eureka/**](http://localhost:8761/eureka/)

**5.) What is API Gateway?**

It provides a single, unified interface for clients to access the functionalities and resources offered by multiple microservices.

**Key Feature:**

1. **Load balancing:** API Gateway can distribute traffic across multiple backend services.

2. **Fault tolerance**: API Gateway can handle failures of backend services and route traffic to healthy services.

3. **Monitoring**: API Gateway can provide monitoring and metrics for API traffic.

4. **Caching and Response Management:** The API gateway can cache

responses from microservices to improve performance and reduce the load on the underlying services.

5. **Authentication & Authorization:** security

**How you use API Gateway in your project or explain the steps:**

1. Add dependency: spring-cloud-starter-api-gateway

2. Add annotation on main class: @EnableZuulProxy

3. Configure your API Gateway routes:

**@Configuration**

public class GatewayConfig {

**@Bean**

public RouteLocator customRouteLocator(RouteLocatorBuilder builder) {

return builder.routes()

// Route for GET requests to /products

.route("product-service", r -> r.path("/products")

.uri("http://product-service-host:product-service-port"))

// Route for POST requests to /products

.route("product-service-post", r -> r.path("/products")

.uri("http://product-service-host:product-service-port"))

// Add more routes as needed

.build();

}

}

**6.) How API gateway integrated?**

Its done with the help of SPRING CLOUD GATEWAY.

**7.) What is Hystrix Dashboard?**

**The Hystrix** Dashboard is a web-based monitoring tool provided by Netflix's HYSTRIX LIBRARY. Hystrix is a **latency and fault tolerance library** designed to help developers build resilient and **fault-tolerant distributed systems**. The Hystrix Dashboard provides a **visual representation of the metrics** and real-time status of the **Hystrix circuit breakers** in a microservices architecture. It allows you to **monitor the health and performance of individual services** that use Hystrix for fault tolerance.(DEVELOPED BY NETFLIX)

**Hystrix circuit breakers** work by monitoring the calls made to remote services and detecting when those services become unresponsive, experience high latency, or produce errors. When a certain threshold is reached, the circuit breaker opens, preventing further calls to the remote service.

**CIRCUIT BREAKERS HAS FOLLOWING STATES:**

1. **CLOSED** (service running properly)

2. **OPEN** (if failure rate>THRESHOLD (set in the config) then the circuit is open)

3. **HALF-OPEN** (after a few seconds (setup in config), it is tried to get

response from this dead service by sending some requests, if succesful, its closed)

**Steps:**

**1.) Add Hystrix Dependency ->** hystrix-core

**2.) Create a Hystrix Command:->**

import com.netflix.hystrix.HystrixCommand;

import com.netflix.hystrix.HystrixCommandGroupKey;

public class MyHystrixCommand extends HystrixCommand<String> {

private final String fallbackResponse;

public MyHystrixCommand (String fallbackResponse){ super(HystrixCommandGroupKey.Factory.asKey("ExampleGroup")); this.fallbackResponse = fallbackResponse;

}

@Override

protected String run () throws Exception {

// Implement your potentially failing code here return "Success";

}

@Override

protected String getFallback() {

return fallbackResponse;

}

}

3.) **Execute the Hystrix Command**:

MyHystrixCommand myCommand = new MyHystrixCommand("Fallback response");

String result = myCommand.execute();

4.) **Hystrix Circuit Breaker Configuration**: HystrixCommand.Setter.withGroupKey (HystrixCommandGroupKey.Factory.

a sKey("ExampleGroup"))

.andCommandPropertiesDefaults(HystrixCommandProperties.Setter()

.withCircuitBreakerRequestVolumeThreshold(20)

.withCircuitBreakerErrorThresholdPercentage(50)

.withCircuitBreakerSleepWindowInMilliseconds(5000));

5.) **Hystrix Dashboard (Optional) dependency**: hystrix-dashboard

**8.) What is Kafka?**

Kafka allows data to be published, consumed, and processed in real-time, facilitating the integration of diverse data sources and enabling real-time data- driven decision-making.

**How you use Kafka in your project or explain the steps:**

**1.** Add dependency: **spring-kafka**

2. Define Kafka Topics:

**a] 'ship event' ->** for ship- related events such as depratures, arrival and location updates.

**b] 'ship-messages' ->** for communication messages between different parts of our application.

3. We use **AIS (Automatic Identification System) data**, which is commonly used for tracking ships, which has two sources like

**AIS Receivers:** Deploy AIS receivers or sensors in areas where you want to track ships.

**AIS Data Providers:** APIs provided by AIS data providers.

4. **AISLib (Java Library):** AISLib is a popular open-source Java library for parsing AIS messages. It supports parsing AIS messages in various formats, including NMEA and AIVDM/AIVDO.

AISLib provides classes and methods to decode AIS messages and extract information such as vessel positions, speed, course, and identification details.

5. we create kafka producer class:

@Service

public class AISDataProducer {

@Autowired

private KafkaTemplate<String, AISData> kafkaTemplate;

public void sendAISData(AISData aisData) {

kafkaTemplate.send("ais-data-topic", aisData);

}

}

6. Publish AIS Data: aisDataProducer.sendAISData(newAISData);

7. Create kafka consumer class:

@Service

public class AISDataConsumer {

@KafkaListener (topics = "ais-data-topic", groupId = "my-group")

public void listen (ConsumerRecord<String, AISData> record) {

// Process the received AIS data

AISData aisData = record.value();

// Add your custom processing logic here

}

}



**OOPS**



**1.) Abstraction**: Abstraction is nothing but the hiding the implementation details

and showing functionality to user. In my project we achieve abstraction using

***[a] Service abstraction***: We create service interface like ship tracking, kafka Service, Livestock Allocation, etc

***[b] Repository abstraction***: We create repository interface to abstract database interaction.

***[c] Using DTO (Data Transfer Object)***: DTOs abstract the structure of data transferred between different parts of an application.

***[d] Dependency Injection***: Use dependency injection to abstract and inject dependencies into components. This allows you to easily replace implementations when needed, promoting flexibility and testability.

**2.) Encapsulation**: Binding a data, methods or members in a single unit it’s called encapsulation. In our project we achieve a encapsulation using private data members like **private ship name, private ship type etc**

**3.) Polymorphism (method overloading and method overriding):**

***[a] Overloading***: allows you to define multiple methods in the same class with the same name but different parameter lists. In our project we use a **Loading**

**Livestock class** in that class we achieve overloading to add different parameters

**like add livestock (int quantity, String type);**

**add livestock (int quantity);**

***[b] Overriding*:** allows you to provide a specific implementation for a method in a subclass that is already defined in its superclass. In our project we use interfaces like ship tracking, loading classes, allocation class, etc so we have to override the

method in service class.

**4) Inheritance:** It’s a parent child relationship, when you inherit from existing class, you can reuse method and fields of the parent class as well as you can add own fields or method also. One object acquires all the properties and behavior

of parent object. In our project we create ship class and extend it using base class like public class Ship {

private String name;

private int capacity;

// Constructors, getters, setters, and other methods

}

public class LivestockShip extends Ship {

private double liveStockCapacity;

// Constructors, getters, setters, and other methods

}

public class PassengerShip extends Ship {

private int passengerCapacity;

// Constructors, getters, setters, and other methods

}



**Design Patterns**



**1.) Factory Method Pattern:** The Factory Method pattern is a creational design pattern that provides an interface for creating objects but allows subclasses to alter the type of objects that will be created.

**Steps:** (in spring boot project) [a] create interface

[b] create a class which implement concrete factory

[c] create service class using a logic

[d] create controller class which called a service class to create ship

Example: **[interface]**

public interface ShipFactory { Ship createShip();

}

**[implement concrete factory]**

// LiveStockFactory.java

@Component

public class LiveStockFactory implements ShipFactory {

@Override

public Ship createShip() {

return new LiveStockFactory ();

}

}

// PassengerShipFactory.java

@Component

public class PassengerShipFactory implements ShipFactory {

@Override

public Ship createShip() {

return new PassengerShip();

}

}

**[Service class]**

// ShipService.java

@Service

public class ShipService {

private final ShipFactory shipFactory;

@Autowired

public ShipService (ShipFactory shipFactory)

{this.shipFactory = shipFactory;

}

public Ship createShip (String type) {

switch (type.toLowerCase()) {

case "livestock":

return shipFactory.createShip(); case "passenger":

return

shipFactory.createShip(); default:

throw new IllegalArgumentException ("Invalid ship type: " + type);

}

}

}

**[Controller class]**

// ShipController.java

@RestController

@RequestMapping("/ships")

public class ShipController {

private final ShipService shipService;

@Autowired

public ShipController (ShipService shipService) {

this.shipService = shipService;

}

@PostMapping("/create")

public ResponseEntity<Ship> createShip (@RequestParam String type) {

try {

Ship ship = shipService.createShip(type); return ResponseEntity.ok(ship);

} catch (IllegalArgumentException e) {

return ResponseEntity.badRequest().body(null);

}

}

}

2.) **Strategy Pattern:** The Strategy pattern is a behavioral design

pattern that defines a family of algorithms, encapsulates each one, and makes them interchangeable. It allows you to select an algorithm or behavior at runtime.

**Steps:** (in spring boot project)

[a] create interface

[b] create a class which implement concrete strategies

[c] create service class using a logic

**[d]** create controller class which called a service class to create ship Example: **[interface]**

// CostCalculationStrategy.java

public interface CostCalculationStrategy { double calculateCost(double weight);

}

**[implement concrete factory]**

// StandardShippingStrategy.java

@Component

public class StandardShippingStrategy implements CostCalculationStrategy {

@Override

public double calculateCost(double weight) {

// Implement standard shipping cost calculation logic return weight \* 2.5;

}

}

// ExpeditedShippingStrategy.java

@Component

public class ExpeditedShippingStrategy implements CostCalculationStrategy {

@Override

public double calculateCost(double weight) {

// Implement expedited shipping cost calculation logic

return weight \* 5.0;

}

}

**[Service class]**

// ShippingService.java

@Service

public class ShippingService {

private final ShippingCostCalculatorFactory calculatorFactory;

@Autowired

public ShippingService(ShippingCostCalculatorFactory calculatorFactory) {

this.calculatorFactory = calculatorFactory;

}

public double calculateShippingCost (String type, double weight) { CostCalculationStrategy calculator = calculatorFactory.getCalculator(type); if (calculator == null) {

throw new IllegalArgumentException ("Invalid shipping type: " + type);

}

return calculator.calculateCost(weight);

}

}

**[Controller class]**

// ShippingController.java

@RestController

@RequestMapping("/shipping")

public class ShippingController {

private final ShippingService shippingService;

@Autowired

public ShippingController(ShippingService shippingService) {

this.shippingService = shippingService;

}

@GetMapping("/calculate")

public ResponseEntity<String> calculateShippingCost(

@RequestParam String type,

@RequestParam double weight) {

try {

double cost = shippingService.calculateShippingCost(type, weight);

return ResponseEntity.ok("Shipping cost: $" + cost);

} catch (IllegalArgumentException e) {

return ResponseEntity.badRequest().body(e.getMessage());

}

}

}



**Junit & Mockito**



**1.) JUnit:** JUnit is a widely used testing framework for Java applications. It

provides annotations and assertions for writing unit tests to verify individual components of our code.

**Junit annotations:**

[**@Test**: This annotation is used to mark a method as a test method.](mailto:@Test:)



[**@BeforeAll and @AfterAll**: These annotations are used to deﬁne methods that](mailto:@AfterAll:) run once before and after all test methods in a test class, respectively.



[**@BeforeEach and @AfterEach**: These annotations are used to deﬁne](mailto:@AfterEach:) methods that run before and after each test method in a test class, providing a clean state for each test.



[**@DisplayName**: This annotation allows you to specify a custom display](mailto:@DisplayName:) name for a test class or method.



[**@Disabled**: This annotation is used to temporarily disable a test class or](mailto:@Disabled:) test method.



[**@RepeatedTest**: This annotation allows you to specify that a test method](mailto:@RepeatedTest:) should be repeated a certain number of times.



[**@ParameterizedTest**: This annotation is used for parameterized testing, where](mailto:@ParameterizedTest:) you can run the same test method with different sets of parameters.



**Postman:** While not a testing framework in the traditional sense, Postman is a popular tool for testing APIs. You can create automated API tests and perform integration testing using Postman

**2.) Mockito:** Mockito is a mocking framework that can be used in conjunction with JUnit. It helps you create mock objects to isolate the components you are testing from their dependencies.

We use Mockito mostly in our project

Example:

public class ShipServiceTest {

@Mock

private ShipFactory shipFactory;

@InjectMocks

private ShipService shipService;

@BeforeEach

public void setUp() {

MockitoAnnotations.initMocks(this);

}

@Test

public void testCreateLiveStockShip() {

// Arrange

When (shipFactory.createLiveStockShip ()).thenReturn(new LiveStockShip ());

// Act

Ship liveStockShip = shipService. createLiveStockShip ("livestock");

// Assert

assertNotNull(liveStockShip);

assertTrue(liveStockShip instanceof LiveStockShip);

}

@Test

public void testCreatePassengerShip() {

// Arrange

When (shipFactory.createShip()).thenReturn(new PassengerShip());

// Act

Ship passengerShip = shipService.createShip("passenger");

// Assert assertNotNull(passengerShip); assertTrue(passengerShip instanceof PassengerShip);

}

@Test

public void testInvalidShipType() {

// Arrange when(shipFactory.createShip()).thenThrow(new

IllegalArgumentException ("Invalid ship type"));

// Act & Assert assertThrows(IllegalArgumentException.class, () ->

shipService.createShip("invalidType"));

}

}

**3.) What is diff between Junit and Mockito ?**

While JUnit focuses on testing individual units of code, Mockito specializes in managing dependencies and mocking external interactions.

**4.) Which server is used for local testing ?**

depending on the type of application we are developing,



Apache Tomcat is one of the most popular choices for running Java web applications locally.



Spring & Spring Boot





**1.) What is Spring ?**

Spring framework is also called the framework of frameworks, It provides support to various other frameworks such as Struts, Hibernate, Tapestry, EJB, JSF etc.

**Advantages:**

Spring is non-invasive

Spring is light weight

Loose coupling (we cannot change in java project change in xml file) Spring has its own container

Spring integrate another framework/tool easily.

It supports all layer of application (web base, desktop base)

2.) **List out the modules of spring?**

There is total 6 modules in spring- • Spring core module

• Spring application context (J2EE) module

• Spring web MVC module

• Spring AOP (aspect oriented programing) module

• Spring ORM (object relational mapping) module

• Spring JDBC (DAO: data access object) module

3.) **What is the SpringBoot?**

Spring boot is not a framework it provide add-on feature of spring application. Spring Boot make it easy to create standalone application we just run spring boot application. Spring boot application are light weight in term of code and configuration.

4.) **What are the features of springboot?**

□Create Standalone application (***both web and desktop application***).

□It provides ***embed tomcat server.***

□ ***Starter Dependency***: - with the help of this feature, springboot aggregates common dependencies together and

eventually improves productivity.

□ ***Spring Initializer:*** - which can create an internal project structure for you.

□Logging and security.

□SpringBoot ***automatically configuration.***

**5.) Which annotation you used in your project? For Entity Classes:**

1. [**@Entity**: This annotation is used to mark a class as a JPA entity,](mailto:@Entity:) representing a table in the database.

2. [**@Id**: Marks a field as the primary key of the entity.](mailto:@Id:)

3. [**@GeneratedValue**: Specifies the strategy for generating primary key values.](mailto:@GeneratedValue:)

4. **@ManyToOne** and [**@JoinColumn**: Define a many-to-one relationship](mailto:@JoinColumn:) between entities.

**For Repository Interfaces:**

1. [**@Repository**: Marks an interface as a Spring Data repository, enabling](mailto:@Repository:) data access and manipulation.

2. **JpaRepository<T, ID>**: A generic interface provided by Spring Data JPA that extends **CrudRepository** and provides additional methods for data manipulation.

**For Controllers:**

1. [**@RestController**: Combines](mailto:@RestController:) **@Controller** and **@ResponseBody**. Marks a class as a controller that handles HTTP requests and returns JSON responses.

2. [**@RequestMapping**: Specifies the base URL path for mapping controller](mailto:@RequestMapping:) methods.

3. [**@PostMapping**: Maps a method to handle HTTP POST requests.](mailto:@PostMapping:)

4. [**@GetMapping**: Maps a method to handle HTTP GET requests.](mailto:@GetMapping:)

5. [**@PathVariable**: Extracts values from the URL path and maps them to](mailto:@PathVariable:) method parameters.

6. [**@RequestBody**: Binds the request body to a method parameter, useful for](mailto:@RequestBody:) handling JSON or XML payloads.

**For Dependency Injection:**

1. [**@Autowired**: Autowires beans by type. Injects instances of required beans](mailto:@Autowired:) into other beans.

**For Data Persistence:**

1. [**@Transactional**: Specifies that a method should be executed within a](mailto:@Transactional:) transaction. Used for managing database transactions.

These annotations help configure various aspects of the Spring Boot application, including defining JPA entities, creating repository interfaces for data access, mapping controller endpoints, and managing dependencies. Keep in mind that this is a simplified example, and in a real-world scenario, you might need to use additional annotations and configurations based on your requirements.



**Exception Handling**



**1.) How you handle exception in your project?**

**Steps:**

**[a]Define Custom Exception Classes:** (Different type of errors can occurred)

public class ShipNotFoundException extends RuntimeException {

public ShipNotFoundException (String message) {

super(message);

}

}

public class InvalidInputException extends RuntimeException {

public InvalidInputException (String message) {

super(message);

}

}

**[b] Use Exception Handling in Controllers:**

@RestController

@RequestMapping("/ships")

public class ShipController {

@Autowired

private ShipService shipService;

@GetMapping("/{id}")

public ResponseEntity<Ship> getShipById(@PathVariable Long id) {

try {

Ship ship = shipService.getShipById(id); return ResponseEntity.ok(ship);

} catch (ShipNotFoundException ex) {

return

ResponseEntity.status(HttpStatus.NOT\_FOUND).body(ex.getMessage());

}

}

@PostMapping("/create")

public ResponseEntity<String> createShip(@RequestBody Ship ship) {

try {

shipService.createShip(ship);

return ResponseEntity.status(HttpStatus.CREATED).body("Ship created successfully.");

} catch (InvalidInputException ex) {

return ResponseEntity.badRequest().body(ex.getMessage());

}

}

}

**[c] Create a Global Exception Handler:**

@ControllerAdvice

public class GlobalExceptionHandler {

@ExceptionHandler(ShipNotFoundException.class)

public ResponseEntity<String>

handleShipNotFoundException(ShipNotFoundException ex) {

return

ResponseEntity.status(HttpStatus.NOT\_FOUND).body(ex.getMessage());

}

@ExceptionHandler(InvalidInputException.class)

public ResponseEntity<String>

handleInvalidInputException(InvalidInputException ex) {

return ResponseEntity.badRequest().body(ex.getMessage());

}

@ExceptionHandler(Exception.class)

public ResponseEntity<String> handleGeneralException(Exception ex) {

// Handle other unhandled exceptions return

ResponseEntity.status(HttpStatus.INTERNAL\_SERVER\_ERROR).body("An error occurred.");

}

}



**Multithreading**



**1.) Have you use multithreading in you project explain?**

Yes, we use multithreading in our project using ‘ExecutorService’

**2.) What are the different types of ExecutorService in Java 8?**

● **ThreadPoolExecutor** -> it’ s a highly configurable and customizable executor service. You can specify the core pool size, maximum pool size, keep-alive time, and other parameters.











● **Executors.newFixedThreadPool(int n) ->** Creates a fixed-size thread pool with a specified number of threads.



● **Executors.newCachedThreadPool()** -> Creates a thread pool that can grow as needed. Threads are added or removed dynamically based on the workload.



● **Executors.newSingleThreadExecutor() ->** Creates a single-threaded executor. Useful when you want to ensure that tasks are executed sequentially.



● **Executors.newScheduledThreadPool(int n**) -> Creates a thread pool for scheduling tasks.Suitable for executing tasks at fixed intervals or with delays.



● **ForkJoinPool ->** Specialized executor for parallel computing tasks, often used with the Fork-Join framework. Well-suited for parallelizing CPU- intensive tasks.



***[a] In our project is the scenario that we need to process multiple livestock loading requests concurrently.***

We use the **ExecutorService** to parallelize the processing.

Example:

@Service

public class LiveStockLoadingService {

private final ExecutorService executorService;

@Autowired

public LiveStockLoadingService(ExecutorService executorService)

{ this.executorService = executorService;

}

public void processLiveStockLoadingRequests(List<LiveStockRequestLoading>

requests) {

// Submit loading requests for parallel processing for (LiveStockRequestLoading request: requests)

{

executorService.submit(() -> {

// Process the loading request asynchronously

// This task will be executed by one of the threads in the thread pool processLoadingRequest(request);

});

}

}

private void processLoadingRequest(LiveStockRequestLoading request) {

// Implement the logic to process a single loading request

}

}

***[b] Mainly we need to track and monitor ships' locations or other real-time***

***data asynchronously, you can use the ExecutorService to periodically update ship information or perform monitoring tasks.***

Example:

@Service

public class ShipMonitoringService {

private final ExecutorService executorService;

@Autowired

public ShipMonitoringService(ExecutorService executorService) {

this.executorService = executorService;

}

public void startMonitoringShips() {

// Schedule a task for periodic ship monitoring executorService.scheduleAtFixedRate(() -> {

// Implement ship monitoring logic here updateShipLocations();

}, 0, 30, TimeUnit.SECONDS); // Monitor every 30 seconds

}

private void updateShipLocations() {

// Implement logic to update ship locations and send notifications

}

}



**Collections**



1.) Which collection you have used in your project?

[a] List -> ArrayList for getting lists of object from Database. [b] Set -> HashSet for mapping POJOs using hibernate. (One-to-Many & Many-to-One)

[c] Map -> for Global Exception



**Daily Routines**



It is a methodology which tells you what we can do to continues development and continue production mean never ending process.

We follow **agile** methodology or process and use work in **sprint** is of two weeks

and **two sprint** is called **Release.** And any task is come then ticket is allocated to us in **Jira**.

Ticket are four types .CR [Change Request], SR [Service Request], PSR [Paid service

Request], PCR [Paid change Request].

These tickets are divided into priority basics that is p1, p2, p3, p4. P1 is called critical, p2 is called high priority, p3 is called medium and p4 is called low priority. Once our team lead assign the ticket to us then we acknowledge that ticket.

We are start doing **analysis**, and mark as a **analysis in progress** and after that we do **technical analysis** and **business analysis** for more clarification. we can ask to business analyst and if it is not satisfied then we can ask to client with team lead and manager meeting with client can’t happen after once in a release (1 month).

After that we start development and we do coding in development environment. Once is coding is done. we do unit testing and Integration testing. Then it deploy on testing environment. i.e QA Environment.

If any problem occur then we fix it and deploy again on QA Environment.

When it’s ok then go to UAT [User Acceptance Testing] Environment. If code bust here then we need analysis and fix it. If issue resolve then it will go to an production environment. In production environment if any issue occur then we have to fix it.

All This communication is done in the JIRA.

Daily work report communication against tickets all the sort of record will stored in Jira





Company Details





Website: www.bgbsystem.com

Address: #62/1, New No:07, 1st Cross, 2nd Main, Ganganagar, Bangalore-560032.

Manager: Anand Kumar

Senior HR Name: Ganesh Hariharan

HR Name: Ganesh Hariharan

Supervisor (or) Reporting Manager (Or) Project Manager

Name : Niranjan S

Mail ID : niranjan@ www.bgbsystem.com

DOJ : 14-Octomber-2019

CTC(Annual) : 7,70,400 [Seven Lakh Seventy Thousand Four Hundred]

(In Hand) : 64,200 [Sixty Four Thousand Two Hundred] PAN : AELPW1264M



**Questions**



**1.**Are you looking for job change? **YES**

**2.** Current working company? **GLOWBRITE SYSTEMS pvt lmt**

**3.** Permanent or contract? **Permanent**

**4.** In How many projects you have involved? **I am involved in 3-projects**

**5.** Relevant Experience and Total Experience? **Total 4 years of experience and relevant is 4 years**

**6.** Current CTC and Expected CTC? **Current CTC is 7.5 Lpa and I’ll expect 13 Lpa**

**7.** Are you ready for relocate? **Yes**

**8.** Any problem with rotational shifts? **No**

**9.** Why are you looking for change? Or why do you want to leave your present job or company? **For better opportunity and carreer growth**

**10.** Notice Period? **I'm serving my notice period.**

**11.** Current stage of the project? **on going**

**13.** How soon you can join after Selection? - **Immediate after the last working day**

**14.** Highest qualification? **Bachelor of computer application.**

**15.** Any offer in hand? **Yes**

**16.** Do have any bond with your parent company? **No**

**17.** Do you have FORM NO.16? - **Yes I will provide necessary documents after completion of my L1 and L2.**

**18.**Do you have Bank statement? - **Yes**

19.You have stayed in your current job for quite a long time, why?

**I joined the company as a fresher. I was in my learning phase and I have gained knowledge.**

**a.** What do you know about us? **from google job searching site b.** What do you do to improve your knowledge **practice**

**c.** Can you perform under pressure **Yes.**

**d.** Are you comfortable working in a team **Yes.**

**e.** What irritates you about co-workers **Late replies, late acknowledgment**

**19.**Is there any particular kind of person you cannot work with **No**

**20.**What qualities would you look for in your senior **confidence and Decision- making ability, Motivating and getting the best out of the individual by promising and commiting to giving him/her the best too.**

**21.** What motivates you at work **work satisfaction**

**22.** Can you work independently **yes**

**22.**Will you be happy to work in night shifts or over the weekends **no**

**23.** Have you ever committed a mistake at work **no**

**24.** What position would you prefer while working on a project **software engineer**

25. What are the most important things for you as a manager----

**Communication, leadership and management**

26. Have you ever had a problem with your peer? Can you give us an example--

**yes not problems but arguments regarding documentation during peer review.**

**27.**Can you tell us something about your previous boss/Manager---**Anand**

**Kumar. He was always support to me and my colleagues**

**29.** Is there anything that you do not like about your last or current job---**I have to upgrade my knowledge to micro services but Im not getting opportunity.**

30.Have there been instances, when your decision was challenged by your colleague or manager?

**31.** How long can you commit to work with us—**As long as I feel as though I**

**am making continuous contributions to its success**

**32.**Would you like to ask us anything---**no**

29.Can you perform under pressure? **yes**

**Showing the direction and the destination**

**Motivating and getting the best out of the individual by promising and commiting to giving him/her the best too.**