For a **gRPC service** in a **Spring Boot application**, a **graceful shutdown** requires handling **both** the Spring lifecycle (@PreDestroy) and the **gRPC server shutdown** explicitly. Unlike HTTP-based services, gRPC servers need to be manually stopped to prevent abrupt terminations.

**✅ Graceful Shutdown for gRPC in Spring Boot**

**1️⃣ Shutdown Hook for gRPC Server**

Since gRPC servers do not automatically stop when Spring Boot shuts down, you must **override the shutdown behavior** and stop the gRPC server gracefully.

**🔹 Example using @PreDestroy**

import io.grpc.Server;

import jakarta.annotation.PreDestroy;

import org.springframework.stereotype.Component;

@Component

public class GrpcShutdownHandler {

private final Server grpcServer;

public GrpcShutdownHandler(Server grpcServer) {

this.grpcServer = grpcServer;

}

@PreDestroy

public void onShutdown() {

System.out.println("Shutting down gRPC server gracefully...");

if (grpcServer != null) {

grpcServer.shutdown(); // Initiates a graceful shutdown

try {

if (!grpcServer.awaitTermination(10, java.util.concurrent.TimeUnit.SECONDS)) {

System.out.println("Forcing gRPC server shutdown...");

grpcServer.shutdownNow(); // Force shutdown if it takes too long

}

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

grpcServer.shutdownNow();

}

}

System.out.println("gRPC server shutdown complete.");

}

}

📌 **Explanation:**

* Calls grpcServer.shutdown() to stop the server gracefully.
* Uses awaitTermination(10, SECONDS) to wait for ongoing requests.
* If requests take too long, calls grpcServer.shutdownNow() to force a shutdown.

**2️⃣ Use DisposableBean for Cleanup (Alternative to @PreDestroy)**

If you want to use **Spring’s lifecycle interface** instead of @PreDestroy, you can implement DisposableBean:

import io.grpc.Server;

import org.springframework.beans.factory.DisposableBean;

import org.springframework.stereotype.Component;

@Component

public class GrpcServiceShutdown implements DisposableBean {

private final Server grpcServer;

public GrpcServiceShutdown(Server grpcServer) {

this.grpcServer = grpcServer;

}

@Override

public void destroy() {

System.out.println("Cleaning up resources before gRPC shutdown...");

grpcServer.shutdown();

}

}

**3️⃣ Ensure Kubernetes (AKS) Stops Sending Requests Before Shutdown**

Since you are running in **Azure Kubernetes Service (AKS)**, you need to **stop traffic before terminating the pod**.

**🔹 Add preStop Hook in Kubernetes Deployment**

Modify your Deployment YAML:

lifecycle:

preStop:

exec:

command: ["/bin/sh", "-c", "sleep 10"]

📌 **Why?** This allows Kubernetes to **remove the pod from the service registry** before shutting it down.

**4️⃣ Enable Spring Boot Graceful Shutdown**

Ensure your Spring Boot app allows **graceful shutdown**:

server.shutdown=graceful

**🔹 Testing the gRPC Shutdown**

1️⃣ **Start the gRPC server** and make some test requests.  
2️⃣ **Send a SIGTERM signal** to the process:

kill -SIGTERM <pid>

3️⃣ **Check logs:**

* Should show **ongoing requests completing** before shutdown.
* No new requests should be accepted.
* gRPC server should shut down cleanly.

**🔥 Final Best Practices for Lossless gRPC Shutdown in Spring Boot**

✅ Use @PreDestroy or DisposableBean to **shutdown the gRPC server gracefully**.  
✅ Call grpcServer.shutdown() and awaitTermination() before forcing shutdown.  
✅ Use a **Kubernetes preStop hook** to stop requests before shutdown.  
✅ Enable server.shutdown=graceful in **Spring Boot**.

Would you like an example with **gRPC clients handling retries** during shutdown? 🚀