

Taller de Práctica: Arquitectura Híbrida y Estrategias de Resiliencia Avanzada

Carrera:	Software	Nivel:	Quinto	Periodo Lectivo:	2025-2026(2)
Docente:	John Cevallos	Paralelos:	A-B	Número de Taller:	2p-1
Asignatura:	Aplicación para el Servidor Web			Fecha/Horas:	8 de Diciembre 2025 (2 horas académicas)

1. Descripción General

Basándose en la arquitectura del proyecto educativo de referencia "Books & Loans", los estudiantes evolucionarán la arquitectura de sus Proyectos Autónomos para separar una parte crítica de su lógica de negocio en un sistema de microservicios distribuidos.

El sistema operará bajo una **arquitectura híbrida** (HTTP + RabbitMQ). El objetivo central es elevar la robustez del sistema a un nivel empresarial, investigando e implementando una Estrategia de Resiliencia Avanzada que no haya sido cubierta en el repositorio de las prácticas de ejemplo.

2. Objetivos de Aprendizaje

Arquitectura Híbrida: Implementar un sistema que combine endpoints REST para la comunicación con el cliente y mensajería asíncrona mediante RabbitMQ para procesos internos críticos.

Integridad Eventual: Comprender y gestionar los retos de la consistencia de datos en sistemas distribuidos.

Patrones Avanzados: Investigar e implementar tecnologías y patrones modernos como Change Data Capture (CDC), Idempotencia, CQRS, Bulkheads u Orquestación.

3. Requisitos de Arquitectura

Cada grupo seleccionará **dos (2) entidades** de su proyecto autónomo actual que tengan una relación de dependencia directa (Maestro-Movimiento / Recurso-Consumidor).

Ejemplos válidos: Producto-Orden, Cita-Médico, Inscripción-Curso, Cuenta-Transferencia.

3.1 Componentes del Sistema

API Gateway (Punto de Entrada)

El Gateway expone endpoints HTTP REST hacia el cliente (Frontend/Postman) y actúa como fachada y enrutador hacia los microservicios.

Microservicio A (Entidad Maestra/Recurso)

Este microservicio posee su propia base de datos independiente y publica o consume eventos de dominio a través de RabbitMQ.

Microservicio B (Entidad Transaccional)

Este microservicio también posee su propia base de datos independiente. Se comunica con el Microservicio A obligatoriamente a través de RabbitMQ para operaciones de escritura, validación o actualización de estado.

⚠ **Restricción Crítica:** No debe existir comunicación HTTP directa entre el Microservicio A y B para el flujo crítico de negocio.

4. El Reto: Selección de Estrategia Principal

⚠ **Restricción Importante:** Los estudiantes NO deben implementar las estrategias básicas vistas en el repositorio de clase (Circuit Breaker Simple, Sagas Manuales, Outbox con Polling).

Deben elegir, investigar e implementar **UNA (1)** de las siguientes estrategias recomendadas:

Opción A: Transactional Outbox + CDC (Change Data Capture)

El Problema: El patrón Outbox convencional usa un Cron Job que consulta la base de datos constantemente (SELECT * FROM outbox). Esto genera latencia y carga innecesaria, conocido como "Polling Hell".

La Estrategia: Implementar un Outbox reactivo usando CDC. En lugar de consultar la tabla repetitivamente, el sistema escucha el Log de Transacciones de la base de datos o utiliza disparadores en tiempo real (ej. LISTEN/NOTIFY en Postgres) para enviar el evento a RabbitMQ inmediatamente después de la transacción.

Opción B: Idempotent Consumer (Consumidor Idempotente)

El Problema: RabbitMQ garantiza "At-least-once delivery". Si la red falla antes del ACK, el mensaje se duplica. Procesar un pago dos veces puede ser catastrófico.

La Estrategia: Implementar deduplicación estricta en el consumidor utilizando claves de idempotencia (Idempotency Keys) almacenadas en Redis o tablas de control, garantizando que el efecto en la base de datos ocurra exactamente una vez aunque el mensaje llegue múltiples veces.

Opción C: CQRS (Command Query Responsibility Segregation)

El Problema: Las operaciones complejas de escritura bloquean las bases de datos, ralentizando las consultas críticas para el usuario.

La Estrategia: Separar físicamente los modelos de lectura y escritura. El Microservicio A maneja la escritura y el Microservicio B mantiene una Proyección

(vista materializada) que se actualiza vía RabbitMQ, permitiendo lecturas rápidas incluso si el servicio de escritura cae.

Opción D: Workflow Orchestration (Temporal.io / Sagas Orquestadas)

El Problema: Las Sagas manuales son frágiles; si el servidor se reinicia a mitad del proceso, el estado se pierde.

La Estrategia: Utilizar un motor de orquestación como Temporal.io (o BullMQ Flows avanzado) que garantice la persistencia del estado paso a paso, manejando automáticamente reintentos y timeouts como infraestructura.

4.1 Otras Opciones de Investigación Aceptadas

Si el grupo desea explorar otros enfoques, las siguientes estrategias también son válidas para la entrega, siempre que se justifique su complejidad "Enterprise":

Patrón Bulkhead (Mamparo): Aislar los recursos del sistema (pools de hilos o conexiones a BD) para que si un consumidor de RabbitMQ se satura o bloquea, no consuma toda la memoria del microservicio, permitiendo que otros procesos sigan funcionando.

Distributed Rate Limiting (Throttling): Implementar un control de flujo distribuido (usando Redis) en el Gateway o en la entrada de la cola, para proteger a los microservicios de picos de tráfico que podrían tumbar la base de datos (Backpressure).

Cache Fallback / Graceful Degradation: Estrategia donde, si el Microservicio Maestro no responde a tiempo (timeout) o la cola está saturada, el sistema automáticamente sirve datos "viejos" (stale) desde una caché o degrada funcionalidades no esenciales para no mostrar un error 500 al usuario.

Dead Letter Exchange (DLX) con Replay Automático: Configuración avanzada de RabbitMQ para manejar "Poison Messages" (mensajes que rompen al consumidor), enviándolos a una cola muerta y activando un mecanismo de reparación y reintento automático bajo ciertas condiciones.

5. Guía de Implementación Paso a Paso

Para completar este taller exitosamente, se recomienda seguir el siguiente flujo de trabajo, adaptando la lógica del tutorial de referencia a sus propias entidades:

Paso 1: Diseño y Definición

Definan claramente cuál es su Entidad Maestra (Recurso limitado) y cuál es su Entidad Transaccional (Acción). Diseñen la estructura JSON de los mensajes que viajarán por RabbitMQ (ej. producto.stock.reservado, cita.solicitada).

Paso 2: Inicialización de Microservicios

Generen los proyectos NestJS independientes (o configuren un Monorepo). Instalen dependencias clave: @nestjs/microservices, amqplib, typeorm. Configuren variables de entorno (.env) para RabbitMQ y Bases de Datos.

Paso 3: Construcción del Microservicio "Maestro" (Servicio A)

Creen la Entidad y el Repositorio con TypeORM. Implementen @MessagePattern o @EventPattern para escuchar y procesar solicitudes asíncronas. Si aplica CDC, configuren la emisión de eventos ante cambios en BD.

Paso 4: Construcción del Microservicio "Transaccional" (Servicio B)

Al crear una transacción local, deben iniciar la comunicación con el Servicio A vía RabbitMQ. Aquí deben codificar la lógica de la Estrategia Avanzada elegida (ej. configurar el Worker, los interceptores de Idempotencia, el servidor de Temporal o los Rate Limiters).

Paso 5: Construcción del API Gateway

Configuren la conexión a los microservicios (ClientsModule). Expongan los Endpoints HTTP (REST) que recibirán las peticiones del cliente y las enrutarán a los servicios correspondientes.

Paso 6: Orquestación y Pruebas

Creen un docker-compose.yml unificado **opcional** (RabbitMQ, BDs, Servicios). Ejecuten pruebas de caos: simulen el fallo específico que su estrategia resuelve (ej. duplicar mensajes, apagar un servicio, desconectar la BD) para validar la resiliencia.

6. Entregables y Sistema de Evaluación

Repositorio de Código (Obligatorio)

Proyecto completo dockerizado que incluya la infraestructura adicional requerida por la estrategia (ej. Redis, Temporal Server, configuración de triggers). El repositorio debe contener un README.md con instrucciones claras de instalación y ejecución.

Modalidad de Evaluación

Opción A: Presentación en Clase (100% de la nota)

El estudiante debe presentar y demostrar en vivo al docente durante la clase el funcionamiento completo de su implementación. La presentación debe incluir:

- Explicación breve de la arquitectura implementada
- Demostración funcional del flujo completo (Happy Path)
- Prueba de resiliencia en vivo: simulación del fallo específico que la estrategia resuelve y demostración de la recuperación automática o protección del sistema
- Respuesta a preguntas técnicas del docente sobre decisiones de diseño e implementación

Opción B: Video Demostrativo (50% de la nota)

Si el estudiante NO logra demostrar exitosamente el funcionamiento en clase, deberá elaborar un video de 3-5 minutos que incluya los mismos elementos de la presentación en vivo. El video debe subirse a una plataforma (YouTube, Vimeo, Google Drive) y la URL debe incluirse en el archivo README.md del repositorio.

 **Importante:** Esta opción solo califica para el 50% de la nota total del taller.

7. Rúbrica de Evaluación

Criterio	Peso	Descripción
Arquitectura Híbrida	30%	Correcta separación de responsabilidades, uso de Gateway REST y RabbitMQ obligatorio para comunicación interna.
Complejidad de Estrategia	40%	Implementación técnica correcta de una estrategia avanzada (de la lista principal o las opciones de investigación).
Demo de Resiliencia	30%	La demostración prueba inequívocamente que el sistema soporta fallos y mantiene la consistencia de datos.

Nota: Se permite utilizar librerías de terceros para facilitar la implementación (ej. nestjs-throttler, bull, drivers de CDC), siempre que se explique su configuración y funcionamiento en el informe.

8. Enlaces de Interés

Documentación Oficial

[NestJS Microservices](#) - Documentación oficial de NestJS para microservicios

[RabbitMQ Tutorials](#) - Tutoriales oficiales de RabbitMQ

[TypeORM Documentation](#) - Documentación de TypeORM para manejo de bases de datos

Patrones de Resiliencia

[Transactional Outbox Pattern](#) - Explicación detallada del patrón Outbox

[CQRS Pattern](#) - Artículo de Martin Fowler sobre CQRS

[Saga Pattern](#) - Patrón Saga para transacciones distribuidas

[Circuit Breaker Pattern](#) - Fundamentos del Circuit Breaker por Martin Fowler

Herramientas y Tecnologías

[Temporal.io](#) - Plataforma de orquestación de workflows

[Debezium \(CDC\)](#) - Plataforma de Change Data Capture distribuida

[Redis](#) - Base de datos en memoria para caché e idempotencia

[BullMQ](#) - Sistema de colas y jobs para Node.js basado en Redis

Recursos Educativos

[Microservices.io](#) - Catálogo completo de patrones de microservicios

[Building Microservices \(Sam Newman\)](#) - Libro de referencia sobre arquitectura de microservicios

[Distributed Systems Patterns](#) - Colección de patrones para sistemas distribuidos