Proyecto Integrador 202610

Asignatura: Integración de Sistemas

Trabajo en equipo + implementación práctica + defensa con demo funcional (sin IDE)

Duración	5 semanas
Modalidad	Trabajo Grupal
Entrega final	Repositorio + evidencias + defensa
Ejecución	Docker Compose (one command), demo principalmente en navegador

1. Contexto del caso

Una empresa mediana (retail/servicios) creció y hoy opera con sistemas heterogéneos. Necesita integrar su flujo Order-to-Cash: toma de pedidos (canal digital), inventario, pagos, logística, notificaciones y analítica. La integración debe ser robusta ante fallos, segura y con trazabilidad.

El foco del proyecto no es "hacer un e-commerce", sino diseñar e implementar una solución de integración empresarial usando patrones y prácticas vistas en el curso.

2. Objetivo general

Diseñar e implementar una plataforma de integración ("IntegraHub") que permita ejecutar el flujo de pedidos end-to-end combinando:

- APIs para integración síncrona.
- Mensajería/eventos para integración asíncrona.
- Integración por archivos (legado).
- Integración de datos (batch/ETL y/o streaming).
- Seguridad, resiliencia e idempotencia.
- Evidencia visible y defendible mediante demo "tipo deploy".

3. Requisitos funcionales mínimos (MVP obligatorio)

3.1 Flujo A – Creación y procesamiento de pedido (E2E)

1. Un cliente crea un pedido vía API (POST /orders).
2. Se genera un evento OrderCreated y se procesa asíncronamente.
3. Se valida y reserva inventario.
4. Se procesa el pago (servicio simulado o real, pero controlado).
5. Se confirma el pedido y se genera el evento OrderConfirmed (u OrderRejected).

3.2 Flujo B – Notificaciones (Pub/Sub)

Cada cambio de estado relevante del pedido debe notificar a:

- Canal de “Operaciones” (webhook a Slack/Discord o servicio simulado).
- Servicio de notificaciones al cliente (simulado).

3.3 Flujo C – Integración por archivos (legado)

Se recibe diariamente un archivo CSV (inventario/catálogo). Debe existir un proceso automatizado que:

- Ingesta el archivo desde una carpeta “inbox”.
- Valida formato y datos.
- Transforma y carga a la persistencia.
- Gestiona errores sin detener el flujo (registro + manejo de mensajes inválidos).

3.4 Flujo D – Analítica (mínimo una de las dos)

Deben implementar al menos una alternativa:

- Streaming: eventos a un bus (Kafka o equivalente) y consumo para métricas.
- Batch/ETL: extracción desde API/DB y carga a Postgres/DB analítica.

4. Requisitos técnicos obligatorios

4.1 Patrones de integración (mínimos obligatorios)

Cada equipo debe implementar y justificar al menos los siguientes patrones:

1. Point-to-Point (cola).
2. Publish/Subscribe (tópico/fanout).
3. Message Router (p. ej. content-based routing o reglas de ruteo).
4. Message Translator (mapeo entre modelos/DTOs/eventos).
5. Dead Letter Channel (DLQ) y estrategia de reintentos.
6. Idempotent Consumer (evidencia de no duplicación ante reintentos/dobles envíos).

4.2 Resiliencia (obligatorio)

Debe existir evidencia de:

- Timeouts.
- Retries (con backoff si aplica).
- Circuit breaker o mecanismo equivalente (debidamente justificado).
- Manejo de fallos demostrable en la defensa (ver sección 7).

4.3 Seguridad (obligatorio)

- Al menos 1 API debe estar protegida con OAuth2 + JWT (o mecanismo equivalente aprobado por el docente).
- Se debe evidenciar control de acceso (token válido vs token inválido).

4.4 Contratos y gobierno mínimo

- APIs documentadas con OpenAPI/Swagger.
- Colección de Postman para pruebas (happy path + error cases).
- Estrategia mínima de versionado o compatibilidad (justificada).

5. Reglas explícitas para la defensa (demo “sin IDE”)

5.1 No aplica demo desde IDE

- No se permite “mostrar que funciona” corriendo código desde el IDE.
- No se permite editar código para corregir issues durante la defensa.

5.2 Uso mínimo de terminal permitido

Se permite terminal únicamente para:

- Levantar el sistema con un comando: docker compose up -d
- Opcional: apagar el sistema: docker compose down

No se permite ejecutar comandos adicionales para “hacer funcionar” componentes, ni scripts manuales para simular resultados.

5.3 Demo debe ser “tipo deploy”

La demostración debe ejecutarse principalmente en navegador, usando:

- Swagger UI (o Portal/API UI) para invocar APIs.
- UIs de operación (por ejemplo RabbitMQ Management UI y/o Kafka UI).
- Pantalla propia del equipo (Demo Portal) o endpoints visibles que permitan seguimiento.

5.4 Demo Portal (obligatorio)

Cada equipo debe entregar un Demo Portal web mínimo (simple; estética no evaluada) que permita:

- Crear pedidos (formulario o botón “Crear pedido demo”).
- Ver lista de pedidos y su estado.
- Ver el Correlation-ID / Order-ID y el último evento asociado (trazabilidad mínima).

Objetivo: que el sistema se vea como un producto operativo, no como un conjunto de servicios aislados.

5.5 Health/Status (obligatorio)

Debe existir un mecanismo visible de “sistema vivo”:

- Página “System Status” en el Demo Portal, o
- Endpoint consolidado /health o /status que muestre estado por servicio.

6. Entregables obligatorios

6.1 Repositorio

Un repositorio por equipo (GitHub/GitLab) con:

- Código completo.
- Instrucciones claras en README.md.
- Arquitectura y evidencias en carpeta /docs.

6.2 Docker Compose (obligatorio)

- Archivo docker-compose.yml (o compose.yml) que levante todo el sistema.
- Puertos definidos; sin configuraciones manuales post-arranque.
- Variables sensibles en .env.example (no subir secretos reales).

6.3 Evidencias y documentación

- Diagrama C4 (Context + Container).
- 2 diagramas de secuencia: (a) “Create Order (E2E)”, (b) “Fallo + Retry + DLQ + recuperación”.
- Matriz de patrones (tabla): Patrón → dónde se usa → por qué → trade-off → evidencia (logs/UI/screenshot).
- Postman collection: happy path y casos de error (token inválido, pago falla, inventario insuficiente, etc.).
- Evidencia de seguridad (JWT) y resiliencia (fallo controlado).

6.4 Presentación final

- 8 a 10 slides máximo (claras y ejecutivas).
- Debe incluir: problema, arquitectura, patrones, demo flow, riesgos, lecciones aprendidas.

7. Defensa (estructura y demo mínima)

Duración total por equipo: 15–20 minutos

- 2–3 min: Contexto + arquitectura + decisiones clave.
- 8–10 min: Demo guiada (en navegador).
- 2–3 min: Preguntas del docente.

Demo mínima obligatoria (se evaluará en vivo):

1. Crear pedido desde Demo Portal/Swagger.
2. Mostrar en RabbitMQ UI (o equivalente) la mensajería (cola/tópico) y el enrutamiento.
3. Mostrar cambio de estado del pedido en el Portal.
4. Inducir fallo controlado (pago o inventario) y evidenciar: retries, DLQ e idempotencia.
5. Mostrar seguridad (token válido vs inválido).
6. Mostrar analítica (streaming o ETL), con evidencia visible.

8. Criterios de evaluación (rúbrica)

Criterio	Peso
Arquitectura y decisiones (trade-offs)	20%
Patrones implementados con evidencia	25%
Resiliencia + DLQ + idempotencia (demo de fallo)	15%
Seguridad (OAuth2/JWT)	10%
Integración de datos (ETL/streaming)	10%
Calidad de entrega (compose, README, Postman, docs)	10%
Defensa (claridad, dominio, respuestas)	10%

9. Reglas de trabajo en equipo

- Grupos con los que se ha venido trabajando durante el semestre
- Definir roles mínimos: Integración/Mensajería, APIs, Datos/ETL, Portal/Demo, Calidad/DevOps (pueden combinarse).
- Cada integrante debe evidenciar contribución (commits, issues o bitácora).

10. Restricciones y buenas prácticas

- No usar soluciones sin justificar; se evaluará comprensión y decisiones.
- Debe existir manejo de errores y mensajes inválidos.
- Se calificará coherencia de la solución, no la cantidad de tecnologías.

11. Checklist de aceptación (mínimo para aprobar)

- Levanta con docker compose up -d sin pasos manuales adicionales.
- Demo Portal operativo con creación y tracking de pedidos.
- Mensajería visible: P2P + Pub/Sub y DLQ en RabbitMQ (o equivalente).
- Evidencia de idempotencia.
- API protegida con JWT y demo de acceso denegado.
- Documentación completa (C4 + secuencias + matriz de patrones + Postman).
- Demo en navegador, sin IDE (terminal solo para levantar/bajar).