**Inventory Notification Arquitectura**

Este sistema está compuesto por dos microservicios independientes:

1. **Inventory Microservice**

**Responsabilidades**

* CRUD de productos.
* Emisión de eventos a RabbitMQ cuando ocurre alguna acción sobre los productos.
* Persistencia de eventos en una tabla Outbox antes de publicarlos.

**Proyectos:**

* **Inventory.API**: Exposición HTTP y background processing (Outbox).
* **Inventory.Application**: Casos de uso, puertos, eventos de dominio.
* **Inventory.Domain**: Entidades y contratos puros.
* **Inventory.Infrastructure**: Integración con RabbitMQ.
* **Inventory.Persistence**: Persistencia relacional (EF Core).

**Base de Datos**

* Tabla Products con información del producto (Id, Name, Category, Price, Stock).
* Tabla OutboxMessages para persistir los eventos antes de publicarlos a RabbitMQ.

**Patrones utilizados:**

* **Clean Architecture**.
* **Transactional Outbox**: los eventos se escriben en la base en la misma transacción que los datos del dominio (Products), evitando inconsistencias. Se reintentan periódicamente.
* **Background Publisher**: un BackgroundService (OutboxPublisherService) se encarga de leer mensajes no publicados y enviarlos al broker.
* **Dead Letter Queue:** Mensajes que no se procesan correctamente se redirigen a colas "dead-letter" para posterior revisión.
* **Polly Retry Policy:** Reintentos con backoff exponencial al publicar mensajes.
* **Circuit Breaker:** Si RabbitMQ falla repetidamente, corta la publicación temporalmente para evitar sobrecargar el sistema.

**Mecanismos de Fiabilidad en la Publicación de Eventos**

El componente encargado de publicar eventos en RabbitMQ es el OutboxPublisherService, un BackgroundService que periódicamente consulta la tabla OutboxMessages por mensajes no publicados.

Para garantizar confiabilidad, implementa los siguientes mecanismos:

* **Try-catch estructurado**: cada intento de publicación está encapsulado en bloques try, lo que evita fallos inesperados y permite capturar excepciones específicas (conectividad, canal cerrado, timeout, etc.).
* **Política de Reintentos con Polly**: si una publicación falla, se reintenta automáticamente hasta 3 veces usando Polly, con **backoff exponencial** (2s, 4s, 8s). Esto reduce presión en el sistema ante errores temporales.
* **Circuit Breaker**: si el servicio detecta múltiples fallos consecutivos (umbral configurable), activa el circuito cortado por 30 segundos. En ese lapso no se intenta publicar, lo que protege a la infraestructura de sobrecarga y da tiempo para recuperación.
* **Confirmación del canal RabbitMQ (WaitForConfirms)**: antes de considerar exitoso un envío, el canal espera confirmación explícita del broker. Si no llega, se registra como fallo y el mensaje permanece en la tabla.
* **Persistencia segura en Outbox**: los mensajes se insertan en OutboxMessages como parte de la misma transacción que afecta datos de negocio (Products). Esto asegura que un cambio en el dominio no se pierda aunque falle la publicación.
* **Estado de publicación**: cada registro Outbox contiene campos como:
* Published: booleano indicando si ya fue enviado.
* Retries: cantidad de intentos realizados.
* LastError: último mensaje de error capturado.
* PublishedAt: timestamp del último intento exitoso.

1. **Notification Microservice**

**Responsabilidades**

* Escuchar los eventos desde RabbitMQ emitidos por InventoryService.
* Procesar los eventos según el tipo: product.created, product.updated, product.deleted.
* Almacenar los eventos en una base de datos local para trazabilidad.

**Proyectos:**

* **Notification.API**: Servicio Background para consumir mensajes.
* **Notification.Application**: Procesamiento y persistencia de eventos.
* **Notification.Infrastructure**: (vacía por ahora, puede contener adaptadores externos en el futuro).
* **Notification.Persistence**: Base de datos local de registros.

**Base de Datos**

* Tabla InventoryEvents que almacena:
  + Id, EventType, ProductId, Payload (evento completo en JSON), ReceivedAt.

**Patrones utilizados:**

* Hexagonal Architecture (Ports & Adapters).
* **Dead Letter Queue**: para mensajes que no pueden ser procesados luego de varios intentos.
* **Event Processor**: clase InventoryEventProcessor centraliza la lógica de negocio según el tipo de evento.
* Circuit Breaker con Polly.

**Procesamiento Resiliente de Eventos Entrante**

El consumidor implementa los siguientes mecanismos para garantizar fiabilidad:

* **Try-catch estructurado**: captura excepciones durante el procesamiento y evita que el mensaje se confirme si hay errores.
* **Reintentos**: los errores transitorios pueden ser reintentados automáticamente mediante políticas configuradas con Polly.
* **Manual Ack/Nack**: el consumidor utiliza confirmación manual. Solo se envía Ack cuando el evento fue procesado exitosamente. Si falla, se envía Nack y el mensaje queda disponible para reentrega.
* **Dead Letter Queue (DLQ)**: si un mensaje falla repetidamente, se redirige a una cola de tipo dead-letter para posterior inspección.
* **Circuit Breaker (Polly)**: si el servicio alcanza un umbral alto de fallos, se corta temporalmente el flujo de procesamiento para proteger el sistema.

Este diseño permite alta resiliencia en el procesamiento de eventos, evita pérdidas silenciosas y facilita la auditoría de eventos no procesados.

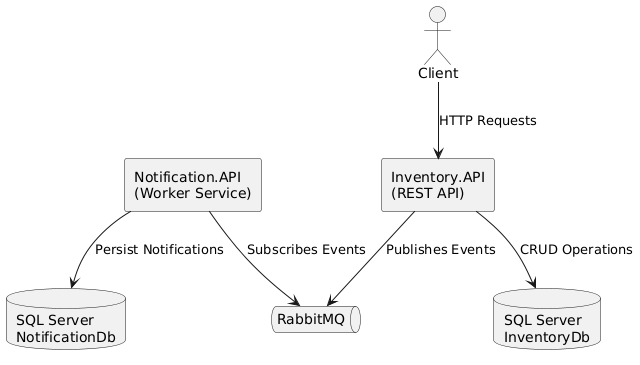
**Consumo de eventos desde RabbitMQ**

El microservicio Notification suscribe a los mensajes publicados por Inventory usando un consumidor conectado a RabbitMQ. Los eventos se enrutan mediante una **routing key** específica:

* product.created
* product.updated
* product.deleted

Cada mensaje contiene el evento en formato JSON dentro del campo Payload, que se almacena íntegro para trazabilidad.

**Inventory Notification Diagrama de Arquitectura**



**Descripción breve:**

**Inventory.API**

* Ofrece endpoints REST (productos).
* Publica mensajes en RabbitMQ.

**RabbitMQ**

* Encola eventos de inventario.
* Proporciona interfaz web.

**Notification.API**

* Worker service que consume mensajes.
* Persiste notificaciones en su base.

**Bases SQL Server**

* Cada microservicio tiene su propia base.

**Documentación de la API**

**1. Información general**

Controlador: **ProductsController**

Ruta base: **api/products**

**2. Endpoints**

* **Obtener todos los productos**

**Verbo HTTP:** GET  
**Ruta:** /api/products  
**Descripción:** Obtiene todos los productos disponibles en el sistema.  
**Parámetros:**

* CancellationToken (query)

**Respuestas posibles:**

* 200 OK → IEnumerable
* **Obtener producto por ID**

**Verbo HTTP:** GET  
**Ruta:** /api/products/{id:int}  
**Descripción:** Devuelve un producto específico según su ID.  
**Parámetros:**

* id (int, ruta)
* CancellationToken (query)

**Respuestas posibles:**

* 200 OK → ProductDto
* 404 NotFound
* **Crear producto**

**Verbo HTTP:** POST  
**Ruta:** /api/products  
**Descripción:** Crea un nuevo producto en la base de datos.  
**Parámetros:**

* CreateProductDto (body)
* CancellationToken (query)

**Respuestas posibles:**

* 201 Created → ProductDto
* 400 BadRequest → ValidationProblemDetails
* **Actualizar producto**

**Verbo HTTP:** PUT  
**Ruta:** /api/products/{id:int}  
**Descripción:** Actualiza los datos de un producto existente.  
**Parámetros:**

* id (int, ruta)
* UpdateProductDto (body)
* CancellationToken (query)

**Respuestas posibles:**

* 204 NoContent
* 400 BadRequest → ValidationProblemDetails
* 404 NotFound
* **Eliminar producto**

**Verbo HTTP:** DELETE  
**Ruta:** /api/products/{id:int}  
**Descripción:** Elimina un producto por su ID.  
**Parámetros:**

* id (int, ruta)
* CancellationToken (query)

**Respuestas posibles:**

* 204 NoContent
* 404 NotFound

**3. Parámetros de entrada**

* **CreateProductDto**
* **Name** (string, obligatorio)
* **Description** (string?, opcional)
* **Price** (decimal, obligatorio)
* **Stock** (int, obligatorio)
* **Category** (string, obligatorio)
* **UpdateProductDto**
* **Id** (int, obligatorio)
* **Name** (string, obligatorio)
* **Description** (string?, opcional)
* **Price** (decimal, obligatorio)
* **Stock** (int, obligatorio)
* **Category** (string, obligatorio)

**4. Modelos de salida**

* **ProductDto**
* **Id** (int)
* **Name** (string)
* **Description** (string)
* **Price** (decimal)
* **Stock** (int)
* **Category** (string)

**5. Flujo de cada acción**

1. **GetAll**
   * Llama a IProductService.GetAllAsync(ct).
   * Mapea cada entidad Product a ProductDto.
   * Devuelve 200 OK con lista de DTOs.
2. **GetById**
   * Llama a IProductService.GetByIdAsync(id, ct).
   * Si no existe, retorna 404 NotFound.
   * Si existe, mapea a ProductDto y retorna 200 OK.
3. **Create**
   * Valida ModelState; si invalido, retorna 400 BadRequest.
   * Construye Product a partir de CreateProductDto.
   * Llama a IProductService.AddAsync(product, ct).
   * Retorna 201 Created con ubicación /api/products/{id} y el ProductDto resultante.
4. **Update**
   * Valida ModelState y que id == dto.Id; si falla, 400 BadRequest.
   * Verifica existencia con GetByIdAsync; si nullo, 404 NotFound.
   * Actualiza propiedades de la entidad y llama a UpdateAsync.
   * Retorna 204 NoContent.
5. **Delete**
   * Verifica existencia con GetByIdAsync; si nullo, 404 NotFound.
   * Llama a DeleteAsync.
   * Retorna 204 NoContent.