**5.1 Vista de Building Blocks – Nivel 0: Contexto General del DTO Middleware**

**Descripción General**

Esta vista muestra el ecosistema de alto nivel en el que opera el DTO Middleware. Presenta cómo múltiples sistemas fuente publican eventos que son procesados por el middleware para construir un objeto unificado de cliente (CustomerPosition). También destaca que el middleware no solo actúa como generador de este DTO, sino que además puede consumir APIs internas, particularmente del ecosistema GLUON, para complementar o enriquecer la información antes de su persistencia.

### **Componentes Principales**

| **Bloque** | **Descripción** |
| --- | --- |
| Data Sources | Conjunto genérico de sistemas origen que publican eventos relevantes. Incluye:• APIs GLUON u otras APIs internas que publican eventos operativos del cliente.• Mainframe, cuyas salidas (como eventos CDC o procesos batch) se traducen a eventos relevantes para el middleware.• Bases de Datos que envían eventos estructurados o semiestructurados.• Otros subsistemas que generan información transaccional o de perfil del cliente. |
| Event Broker | Componente intermedio (ej. Apache Kafka, Redpanda, IBM MQ) que desacopla publicación y consumo de eventos. Los eventos se almacenan en tópicos desde donde el middleware los consume. |
| DTO Middleware | Núcleo del proceso de generación del CustomerPosition. Dentro de este middleware:• Se consumen eventos del broker.• Se realizan agregaciones y enriquecimientos.• Se aplican estrategias de fallback, consolidación y generación de nuevos eventos.• También puede consumir directamente APIs GLUON, como CarSecurity y CarJWKS, cuando se requiere información adicional que no viene en los eventos (por ejemplo, conversión de PAN a CardID).• Toda esta lógica es implementada de forma modular y altamente configurable usando Spring Integration y Spring Cloud Stream. |
| Repositorio CustomerPosition | Representa el almacenamiento final del DTO generado. Es flexible en cuanto a tecnología e incluye:• Redis JSON, como caché transaccional (L1).• Oracle Converged Database, como base de datos persistente (L2).El diseño permite incorporar nuevos motores de almacenamiento sin cambiar la lógica del middleware. |
| GLUON APIs (consumidas por el Middleware) | Conjunto de APIs internas que pueden ser invocadas por el middleware para obtener datos complementarios. Por ejemplo:• CarSecurity: para obtener información segura de tarjetas.• CarJWKS: para extraer claves o identificadores relacionados con seguridad de tarjetas.Este consumo ocurre en paralelo al procesamiento de eventos, y sus respuestas enriquecen el DTO. |

**Interacciones Clave**

* Envío de eventos (send events): Todos los sistemas origen publican eventos hacia el broker.
* Consumo de eventos (pull events): El middleware consume eventos desde el broker y los transforma.
* Consultas de fallback y enriquecimiento (balances/fallback query, API consume): El middleware puede hacer llamadas síncronas a APIs GLUON (como CarSecurity) o a sistemas fuente para completar información faltante.
* Persistencia del DTO (Set/Update DTO): El middleware guarda el DTO generado en los repositorios correspondientes.
* Consumo indirecto del DTO: Las APIs externas (por ejemplo, de canal o GLUON) consumen el DTO desde la base de datos o la caché, sin depender de la lógica del middleware.

**Consideraciones Técnicas**

* El middleware opera de manera desacoplada, siendo capaz de integrarse con cualquier sistema que publique eventos compatibles.
* Puede realizar consultas directas a APIs del ecosistema GLUON, siempre que estén disponibles, para enriquecer la información.
* Su arquitectura modular permite extender nuevas capacidades (como nuevos agregadores o repositorios) por configuración, sin necesidad de reescribir la lógica principal.
* Se respeta el modelo BIAN como marco conceptual para CustomerPosition, aunque la estructura del JSON puede simplificarse para balancear performance y mantenibilidad.

Perfecto. A continuación te presento el texto replanteado del Nivel 1 de la arquitectura del middleware DTO, siguiendo el marco Arc42, sección 5.1 – Building Blocks. Esta versión amplía significativamente el detalle de cada componente y flujo, incluyendo las plataformas específicas, repositorios, frameworks, y consideraciones de diseño técnico que has señalado.

**🧱 Sección 5.1 – Building Blocks**

**Nivel 1 – Arquitectura lógica del Middleware DTO**

Este nivel representa una descomposición estructural completa del DTO Middleware, ilustrando la interacción de componentes tecnológicos y funcionales entre los sistemas fuente, la plataforma de eventos, los microservicios de integración y los repositorios de persistencia. Todo el diseño responde a una arquitectura desacoplada basada en eventos, altamente configurable, orientada a la consolidación del objeto CustomerPosition.

**🔹 1. Sistemas Fuente (**

**DataSources**

**)**

Los eventos que alimentan al middleware provienen de distintos orígenes heterogéneos, los cuales se describen por dominio funcional o tecnología base:

* Mainframe OB2: Se identifican los principales contextos funcionales tradicionales:
  + Pampa
  + Captación
  + Cartera

Cada uno genera eventos a través de mecanismos CDC (Change Data Capture), principalmente vía IBM CDC. Los eventos contienen el nombre de la tabla origen, valores anteriores y nuevos, y metadatos como el timestamp y la clave primaria.

* Bases Oracle:
  + Fenix, VisualTime, Impact

Estas fuentes utilizan Oracle GoldenGate CDC para emitir eventos estructurados JSON, compatibles con transformaciones posteriores.

* Otros subsistemas:

Bases adicionales o plataformas como Rector y PLARD (AWS) también pueden enviar eventos CDC mediante conectores uniformados o APIs.

**🔹 2. Plataforma de Eventos (Confluent – Cluster C)**

Todos los eventos generados se publican hacia Kafka, dentro del Cluster C de la plataforma Confluent. Esta instancia de Kafka actúa como bus de eventos centralizado y opera con los siguientes componentes:

* Kafka Core / Event Bus: Sistema principal para la distribución de eventos. Todos los microservicios escuchan desde este bus según los tópicos definidos.
* Kafka Connect:
  + Conectores de entrada y salida para integraciones con otros sistemas.
  + S3 Sink Connector conectado a Scality, utilizado para persistencia de eventos y auditoría.
* Scality (S3 compatible): Repositorio de almacenamiento de eventos procesados. Aquí se mantiene una trazabilidad completa en objetos tipo JSON.

**🔹 3. Middleware de Integración (**

**DTO Middleware**

**)**

Este bloque contiene los microservicios de orquestación y procesamiento de eventos, desplegados en OpenShift como parte de la estrategia Caas (Container as a Service).

**Microservicios Principales:**

1. customer-position-ms

Enfocado en la generación y mantenimiento del objeto CustomerPosition. Implementa los siguientes componentes:

* + Event Consumer

Entrada única de eventos desde Kafka. Identifica la naturaleza del evento y lo redirige al componente adecuado.

* + DTO Composer

Encargado de enrutar el evento a uno o varios agregadores configurables. Cada agregador representa una función lógica como:

* + - BalancesAggregator
    - CardAggregator
    - AccountsAggregator

El DTO Composer permite recibir:

* + - Un evento que actualiza un campo.
    - Varios eventos para un solo campo.
    - Un evento que impacta múltiples campos.
    - Múltiples eventos que actualizan múltiples campos.
  + DTO Generator

Una vez que todos los agregadores han resuelto sus valores, genera el objeto CustomerPosition completo.

* + DTO Persist

Se encarga de guardar el objeto tanto en Redis (L1) como en Oracle (persistencia definitiva).

Integra la lógica de backpressure, consolidando múltiples actualizaciones en una sola operación si son cercanas en el tiempo.

1. conciliator-ms

Especializado en resolver inconsistencias, reintentos y trazabilidad de eventos fallidos.

* + Conciliator

Evalúa si un evento debe reenviarse o escalarse. Utiliza la tabla ConciliatorChannel en Oracle para almacenar comandos fallidos bajo el patrón durable channel.

* + Event Generator (Fallback)

Emite nuevos eventos o comandos correctivos para que el DTO Composer reprocese una actualización específica.

**🔹 4. Repositorios de Persistencia**

* Oracle Database
  + CustomerPosition (Tabla): Persiste el DTO completo como un objeto JSON, usando funciones nativas de Oracle para indexación sobre atributos clave.
  + ConciliatorChannel (Tabla): Actúa como canal duradero para almacenar eventos o comandos en espera de reprocesamiento o intervención manual.
* Redis
  + Motor: Redis con módulo RedisJSON.
  + Estructura: La vista CustomerPosition se materializa como un objeto JSON, indexado por CustomerID.
  + Propósito: cache de primer nivel (L1) para respuestas rápidas desde APIs o GLUON.

**🔹 5. APIs Externas Consumidas**

El middleware también actúa como consumidor de APIs externas para enriquecer la información que no puede ser obtenida vía eventos:

* Card Security API:

Permite convertir el PAN en un CardID y consultar reglas de seguridad asociadas.

* Card JWKS API:

Permite validar y obtener claves públicas para firma o encriptación.

Ambas se acceden vía API Connect (Gateway Gluon) y son llamadas desde DTO Composer durante la generación del DTO cuando es necesario completar campos específicos.

**🔹 6. Tecnologías y Frameworks**

* Spring Cloud Stream:

Utilizado para el binding de eventos Kafka a canales lógicos.

* Spring Integration:

Framework principal para orquestar flujos de mensajes, transformaciones, y agregadores configurables.

* OpenShift (Caas):

Todos los microservicios están desplegados sobre contenedores en esta plataforma.

* Kafka Connect / Scality:

Base para la arquitectura de auditoría y trazabilidad de eventos.

**🧠 Consideraciones Clave**

* Todos los agregadores del DTO Composer son configurables, permitiendo extender la solución sin codificación adicional.
* El diseño permite materializar cualquier objeto BIAN (no solo CustomerPosition) en los mismos repositorios.
* La lógica de persistencia, cacheo, y reintento está desacoplada de los componentes de composición, lo que facilita su extensión.
* El patrón de eventos nuevos derivados también está implementado, permitiendo que un evento recibido pueda generar uno nuevo para notificación o propagación.

¿Te gustaría que convierta esta misma sección también en formato tabla (como el Nivel 0), o la dejamos únicamente en esta narrativa?

Perfecto. A continuación te presento la descripción completa del componente Event Consumer (nivel 2) del microservicio customer-position-ms, utilizando la redacción más precisa que acabas de validar, e integrando todos los elementos mostrados en ambas imágenes (filtros, enriquecimiento de headers, ruteo, y el ejemplo):

**🔷 Nivel 2 – Componente**

**Event Consumer**

**(microservicio**

**customer-position-ms**

**)**

**Descripción General**

El componente Event Consumer es el único punto de entrada para los eventos recibidos desde el broker de eventos (Kafka) dentro del DTO Middleware. Su propósito principal es recibir, filtrar, enriquecer y enrutar los eventos que serán procesados por los distintos subcomponentes del Middleware, asegurando que solo los eventos relevantes sean propagados en el sistema.

**Funcionalidad Detallada**

**1.**

**Recepción de eventos**

El componente cuenta con binders externos de entrada (Input External Binders) que se conectan al broker y reciben eventos de tipo E-Events (CDC) y Command Events. Estos binders están implementados mediante Spring Cloud Stream, usando canales enlazados a tópicos de Kafka.

**2.**

**Filtrado por sistema fuente**

Cada evento recibido es enviado a un filtro específico según su sistema fuente. Los filtros actuales incluyen:

| **Filtro de Entrada** | **Fuente del evento** | **Propósito** |
| --- | --- | --- |
| DB2-MAINFRAME CDC | Mainframe (DB2) | Filtra eventos de tablas relevantes del mainframe. |
| PLARD Filter | AWS PLARD | Filtra eventos aplicables del entorno PLARD. |
| PGI Investment Filter | Oracle (PGI) | Filtra eventos de inversión. |
| Fenix Filter | Oracle (Fenix) | Filtra eventos desde el sistema Fenix. |
| Rector Filter | Oracle (Rector) | Filtra eventos de Rector. |
| VisualTime Filter | Oracle (VisualTime) | Filtra eventos asociados a ese sistema. |

Cada filtro puede aplicar una lógica adicional de retención, por ejemplo, incluyendo o excluyendo ciertos eventos por tabla (ej. PEDT001).

**3.**

**Enriquecimiento de Headers**

Cada evento filtrado es enviado a un enriquecedor (Enricher) que incorpora los headers requeridos por el Middleware. Esto asegura estandarización en el tratamiento de mensajes, agregando trazabilidad y contexto.

**Ejemplo de Headers agregados:**

**4.**

**Ruteo hacia componentes de negocio**

Una vez enriquecido, el evento es enviado a un canal directo (routingChannel) que utiliza un HeaderValueRouter para dirigir el mensaje a su destino:

| **Header (DTO-MESSAGE-TYPE)** | **Canal de destino** | **Propósito** |
| --- | --- | --- |
| DTOGENERATOR-COMMAND | dtoGeneratorCommandChannel | Se envía al componente generador de DTOs. |
| CONCILIATOR-COMMAND | conciliatorCommandChannel | Se envía al componente Conciliador. |
| EVENT | eventsChannel | Se distribuye mediante Publish-Subscribe. |

**Estructura Técnica**

| **Elemento** | **Descripción** |
| --- | --- |
| Spring Integration | Framework de integración usado para definir filtros, enricher y canales. |
| DirectChannel | Canales internos usados para comunicar filtros y enriquecedores. |
| HeaderValueRouter | Permite ruteo dinámico basado en headers configurados. |
| RecipientList Router | Distribuye eventos a múltiples destinos si es necesario. |
| Binders externos | Se conectan a Kafka para recibir los mensajes. |

Perfecto, a continuación te presento el Nivel 3 (Building Block Level 3) del componente Event Consumer, con base en tu imagen actualizada y tus instrucciones, redactado en formato Arc42.

**🧩 Arc42 – Building Block View: Nivel 3 –**

**Event Consumer Component**

**1. Descripción General**

El componente Event Consumer es el único punto de entrada de eventos externos provenientes del Event Broker (Kafka) hacia el middleware DTO Middleware. Este componente es compartido por múltiples microservicios, incluyendo customer-position-ms y conciliator-ms.

Su principal responsabilidad es:

* Recibir eventos desde múltiples tópicos Kafka, agrupados por fuente.
* Aplicar filtros por sistema fuente y tabla.
* Enriquecer los mensajes con headers de control estandarizados (DTO-\*).
* Rutar de forma interna, según tipo de evento o comando, hacia componentes consumidores como DTO Generator o Conciliator.

El diseño se basa en Spring Integration, utilizando DirectChannel, HeaderEnricher, RecipientListRouter, y HeaderValueRouter.

**2. Estructura Interna**

Event Broker (Kafka)

│

├── Binders [E-Events] ──▶ [Event Consumer Component]

│ ├─ Filters por sistema y tabla

│ ├─ Enriquecedores de headers

│ └─ Ruteadores (Recipient List, Header Router)

│

└── Binders [Commands] ─▶ (igual tratamiento)

Cada evento pasa por los siguientes pasos:

1. Filtro por fuente (DB2-Mainframe, PLARD, PGI, etc.).
2. Filtro de tabla (si aplica): solo tablas relevantes (ej. PEDT001).
3. Enriquecimiento de headers de control estándar.
4. Envío a canal común de ruteo (routingChannel).
5. Distribución vía RecipientListRouter a:
   * Canal de eventos (Events),
   * Canal de comandos para DTO Generator,
   * Canal de comandos para Conciliator.

**3. Subcomponentes**

| **Subcomponente** | **Tipo** | **Descripción** |
| --- | --- | --- |
| Input External Binders | Spring Cloud Stream | Vinculan tópicos Kafka a canales internos. Separados por Events y Commands. |
| System Filters | Integration Filter | Aplica lógica por nombre de tópico o metadata para cada sistema. |
| Retention Filters | Integration Filter | Deja pasar solo tablas relevantes (ej. PEDT001). |
| Header Enrichers | HeaderEnricher | Agrega headers estandarizados DTO-\*. |
| RoutingChannel | DirectChannel | Canal intermedio común para todos los flujos. |
| RecipientListRouter | Router | Envía mensajes hacia DTO Generator, Conciliator, etc., según header. |

**4. Flujos Principales**

**4. Flujos Principales**

| **Flujo** | **Descripción técnica** |
| --- | --- |
| Flujo 1: CDC Mainframe | Recibe evento → Filtra por tópico PEDT001 → Enriquecer headers DTO → Enviar a routingChannel → Enrutamiento hacia DTO Generator |
| Flujo 2: Evento PLARD | Evento ingresa por binder PLARD → pasa por filtro PLARD → enriquecedor específico → Routing channel → routed via RecipientListRouter |
| Flujo 3: Comando Conciliator | Comando entra por tópico de tipo CONCILIATOR-COMMAND → enriquecido → routed via HeaderValueRouter → canal ConciliatorCommand |
| Fallback interno | Si un evento no cumple reglas (tabla no válida o headers malformados) → se enruta al canal de error con manejo especial definido en configuración técnica. |

**5. Consideraciones Técnicas**

* Cada tipo de evento CDC tiene su propio filtro y enriquecedor para aislar responsabilidades.
* La estandarización de headers es obligatoria y validada antes del ruteo.
* Los RecipientListRouter permiten que múltiples microservicios reciban eventos simultáneamente si cumplen con criterios comunes.
* Todos los canales son internos (DirectChannel) y no usan colas persistentes, salvo cuando se conectan a canales durables desde Conciliator.

**6. Headers estandarizados (DTO Middleware)**

| **Header** | **Descripción** | **Ejemplo** |
| --- | --- | --- |
| DTOM-type | Tipo de mensaje: EVENT, DTOGENERATOR-COMMAND, CONCILIATOR-COMMAND | EVENT |
| DTOM-origin | Sistema origen del mensaje. | DB2-MAINFRAME-CDC |
| DTOM-origin-timestamp | Fecha y hora de creación del evento en el sistema origen. | 2025-07-14T08:34:20.987Z |
| DTOM-origin-id | ID lógico del objeto afectado. | ACC12345 |
| DTOM-origin-table | Tabla que originó el evento. | DB2-MAINFRAME-CDC\_PEDT001 |
| DTOM-user-id | Identificador del usuario (opcional si no aplica). | BUC87654321 |
| DTOM-transaction-id | ID único de transacción de la fuente. | TXN-20250714-0001 |
| CORRELATION\_ID (Spring) | Correlación técnica interna. | TXN-20250714-0001 |
| TRACE\_ID | ID de traza para observabilidad (opcional). | 00-4bf92f3577b34da6... |

Estos headers se agregan en el subcomponente de Header Enricher según el tipo de evento. En el ejemplo de imagen, se aplica en DB2-MAINFRAME-CDC Enrich.

**7. Frameworks y tecnologías utilizadas**

* Spring Integration: Core del patrón de integración (filtros, canales, enricher, routers).
* Spring Cloud Stream: Binders para conectarse con tópicos Kafka.
* Kafka: Event broker principal. Topics diferenciados por fuente y tipo (CDC/Command).
* Java DSL de Spring Integration: Define ruteos lógicos, header routers, recipient lists, etc.

**8. Diagramas relacionados**

* Nivel 1: Despliegue de microservicios donde Event Consumer sirve a customer-position-ms y conciliator-ms.
* Nivel 2: Detalle estructural de todos los flujos internos y subcomponentes del Event Consumer.
* Anexo: Tabla detallada de headers en este documento.

¿Deseas que lo formatee para incluirlo directamente en tu documento final o en Word para presentación ejecutiva?

Perfecto. Ya tomé nota de que en el componente DTO Generator, el enrich header se refiere a la asignación del header PRIORITY usando IntegrationMessageHeaderAccessor.PRIORITY, y que los eventos de balances tienen prioridad mayor que los eventos funcionales o técnicos.

Ahora procedo a redactarte el Nivel 3 completo del componente DTO Generator [CustomerPosition] en formato ARC42, incorporando todas las imágenes y la precisión del punto 4. Dame un momento.

Aquí tienes el Nivel 3 completo del componente DTO Generator [CustomerPosition], redactado con base en el marco Arc42 e incluyendo todos los elementos visualizados en las imágenes, junto con la precisión sobre los headers de prioridad.

**Nivel 3 – DTO Generator Component (CustomerPosition)**

Formato: Arc42 – Building Block View (Nivel 3)

**1. Descripción general del componente**

El componente DTO Generator [CustomerPosition] forma parte del microservicio customer-position-ms y es responsable de procesar eventos estandarizados provenientes del Event Consumer Component. Su objetivo principal es generar y actualizar el objeto CustomerPosition, alineado al modelo BIAN, a partir de eventos funcionales, técnicos y contables, aplicando reglas de negocio, agregación y enriquecimiento.

Este componente está construido sobre el framework Spring Integration, lo que permite implementar flujos orquestados mediante canales, filtros, enricher, routers y activadores de servicio, con un diseño desacoplado y altamente extensible.

**2. Flujos principales del componente**

| **#** | **Flujo** | **Descripción** |
| --- | --- | --- |
| 1 | Recepción de eventos | El componente recibe eventos desde un canal tipo PublishSubscribeChannel, proveniente del Event Consumer. |
| 2 | Filtrado y priorización | Los eventos entrantes son clasificados mediante filtros técnicos/funcionales y de balance. Cada tipo de evento es enrutado a través de distintos DirectChannel o PriorityChannel, estableciendo prioridades mediante encabezados. |
| 3 | Agregación funcional/técnica | Los eventos que no involucran balances (por ejemplo, cambios de datos en cuentas, tarjetas o préstamos) se procesan por agregadores funcionales/técnicos. |
| 4 | Agregación de balances | Los eventos relacionados con saldos se enrutan hacia el agregador de balances. Estos eventos tienen una prioridad mayor (PRIORITY = 3) en comparación con los funcionales (PRIORITY = 0), lo que permite su procesamiento preferente. |
| 5 | Enriquecimiento de headers | Antes de pasar al siguiente paso, se agrega el header IntegrationMessageHeaderAccessor.PRIORITY para establecer la prioridad de procesamiento. |
| 6 | Normalización de CustomerPosition | El flujo funcional pasa por un proceso de normalización del objeto CustomerPosition, que permite limpiar, validar y transformar los datos. |
| 7 | Consulta de fuentes externas | Antes de generar el DTO final, se invoca (si es necesario) a servicios como BalanceQuery y CardSecurity, lo cual se hace mediante Service Activators. |
| 8 | Actualización y persistencia | Los datos normalizados se agregan y procesan por un Update Aggregator, el cual orquesta la escritura tanto en Redis como en Oracle, a través de activadores de servicio. |
| 9 | Publicación de cambios | Finalmente, se emite el resultado a través de un PublishSubscribeChannel para otros sistemas que puedan estar interesados en cambios de CustomerPosition. |

**3. Componentes internos**

| **Componente** | **Función** |
| --- | --- |
| Functional/Tecnical Aggregators | Encargados de procesar eventos funcionales (actualizaciones no financieras), aplicando lógica condicional, filtros y almacenamiento temporal en Redis si se requiere. |
| Balance Aggregator | Componente especializado para eventos contables; realiza agregación basada en saldos, gestiona backpressure y garantiza prioridad en su procesamiento. |
| Customer Position Normalization | Estandariza y valida la información del DTO antes de su consolidación final. |
| Update Aggregator | Componente que define reglas para actualizar campos específicos del CustomerPosition solo cuando hay cambios. |
| Redis Customer Position ServiceActivator | Persiste el DTO en Redis, utilizando estructura JSON para permitir proyecciones parciales. |
| Oracle Customer Position ServiceActivator | Realiza la escritura final en la base de datos persistente (Oracle), como fuente de verdad. |
| BalanceQuery y CardSecurity | APIs externas del ecosistema GLUON utilizadas para complementar información antes de materializar el DTO. |

**4. Enriquecimiento de headers**

El header más relevante en este componente es el de prioridad, usado para gestionar el orden de procesamiento entre eventos técnicos/funcionales y eventos de balances.

**Tabla – Headers Enriquecidos**

| **Header** | **Tipo** | **Valor / Ejemplo** | **Descripción** |
| --- | --- | --- | --- |
| IntegrationMessageHeaderAccessor.PRIORITY | Integer | 0 para eventos funcionales / 3 para eventos de balance | Define el orden de prioridad en los canales tipo PriorityChannel. Valor mayor implica prioridad más alta. |
| CoreTransactionID | UUID | 123e4567-e89b-12d3-a456-426614174000 | Identificador único de transacción, utilizado para trazabilidad en la plataforma y WC3. |

**5. Consideraciones técnicas**

* Backpressure: El componente aplica mecanismos de backpressure tanto en los flujos de balance como en los flujos de actualización, utilizando almacenamiento temporal en Redis.
* Almacenamiento intermedio: Redis se emplea como caché de primer nivel (L1), especialmente para eventos complejos o acumulativos.
* Normalización vs actualización: La normalización ocurre antes de decidir si se requiere una actualización real del CustomerPosition, evitando escrituras innecesarias.
* Persistencia final: Oracle es la fuente persistente de largo plazo; Redis se usa como capa de acceso rápido y proyección.
* Tolerancia a fallos: En caso de error en el proceso de generación del DTO, se puede invocar al DTO Fallback (documentado por separado), el cual interactúa con el Conciliator.

**6. Dependencias externas**

| **Sistema** | **Tipo** | **Función** |
| --- | --- | --- |
| Event Consumer Component | Interno | Proveedor de eventos estandarizados. |
| Redis | Interno | Almacenamiento L1, soporta estructuras JSON. |
| Oracle | Interno | Almacenamiento persistente de CustomerPosition. |
| CardSecurity, BalanceQuery | API externa GLUON | Servicios que proporcionan datos adicionales (por ejemplo, ID de tarjeta derivado de PAN). |

Claro. Aquí tienes el texto completo en formato ARC42 – Nivel 3 para el componente DTO Generator [CustomerPosition], considerando las imágenes que compartiste, los headers de prioridad, el uso de Redis y Oracle, y todos los subcomponentes involucrados:

**Arc42 – Nivel 3: DTO Generator Component [CustomerPosition]**

**1. Descripción general del componente**

El componente DTO Generator [CustomerPosition] es el núcleo funcional responsable de componer, transformar y persistir el objeto transversal CustomerPosition, alineado al modelo BIAN. Este componente actúa tras el consumo de eventos estandarizados por el Event Consumer, habilitando la consolidación de información técnica, funcional y contable de múltiples fuentes del core bancario (e.g. mainframe, CDCs, eventos de tarjetas, etc.).

Opera sobre el framework Spring Integration, lo que permite definir flujos desacoplados mediante canales, filtros, enricher, routers y service activators. Su diseño modular permite procesar distintos tipos de eventos en paralelo y aplicar priorización y backpressure cuando es necesario.

**2. Flujos principales del componente**