# LMS Shared Library v1.0.0 – Developer Guide

## Overview

The **LMS Shared Library (v1.0.0)** is a collection of reusable modules that provide foundational features for all LMS microservices. Its goal is to enforce consistent design practices – including multi-tenancy, security, messaging, validation, and observability – across the platform. By centralizing common concerns (such as tenant context handling, JWT authentication, error handling, and event publishing), the shared library reduces boilerplate in each service and ensures that cross-cutting policies (like tenant isolation and logging standards) are uniformly applied. The library is internal to the development team (not a public SDK), and it forms the basis of our microservices’ infrastructure. It is versioned with semantic versioning (starting at **v1.0.0** for the initial release), and updates are managed to minimize breaking changes while clearly documenting any modifications in the changelog (see **Versioning and Release Management** below).

**Modules:** The shared library is organized into multiple Maven modules, each targeting a specific concern area:

* **shared-core:** Core utilities and base classes (domain base entities, request context, id generators, common exceptions, etc.).
* **shared-security:** Authentication/authorization integration (JWT parsing, role-based access control helpers, multi-tenant security filters).
* **shared-db:** Database and persistence support (DataSource configuration, multi-tenant filters, base repository, migration utilities).
* **shared-messaging:** Kafka messaging patterns (producer/consumer config, Outbox pattern, event wrapper, idempotent consumers).
* **shared-observability:** Logging, metrics, and tracing setup (Micrometer metrics, OpenTelemetry tracing, correlation IDs, audit logging).
* **shared-validation:** Custom validation annotations and validators (for domain-specific formats and rules).
* **shared-rest:** Web-layer utilities (exception handlers producing standard error responses, request filters like correlation and idempotency, OpenAPI docs auto-config).
* **shared-test:** Testing utilities for integration tests (Testcontainers support, data fixtures, base test classes).

Together, these modules provide a **foundation** for building LMS services that are **tenant-aware, secure, consistent, and observable**. New microservices can be bootstrapped quickly by leveraging the shared library, which ensures common patterns (such as enforcing tenant IDs on all database queries, or emitting logs/traces in a standard format) are applied out-of-the-box.

## Module Structure and Dependencies

Each module in *lms-shared* has a clear purpose and carefully controlled dependencies to avoid cyclic references. The modules are arranged in a layered fashion (see **Figure 1**), where lower-level modules provide primitives that higher-level ones build upon. **Figure 1** illustrates the allowed dependency flow:

* **shared-core** stands at the base: it has no dependencies on other shared modules. Higher-level modules may depend on *core* for common utilities and contexts.
* **shared-security, shared-db, shared-validation** are foundational modules that may depend on *core* but not on each other (to keep concerns separated). For example, *shared-security* uses core’s context utilities for tenant and user info, and *shared-db* uses core’s BaseEntity classes. These modules do not have circular dependencies – e.g., *core* does not depend on *security*, etc.
* **shared-messaging, shared-rest, shared-observability** are higher-level modules that can depend on core and on certain foundational modules. For instance, *shared-rest* depends on *core* (for request/response wrappers and idempotency utilities) and may interact with *security* (for injecting security info into OpenAPI docs). *shared-messaging* depends on *core* and *db* (it defines JPA entities for the Outbox pattern and thus relies on *shared-db*’s Hibernate configurations) and uses core’s utility classes. *shared-observability* depends on *core* (for correlation context) and may tie into *messaging* (to propagate tracing info to Kafka). Importantly, there are **no cyclic dependencies** – each module only depends on those below it in the hierarchy.
* **shared-test** is a special-case module used only in testing scopes. It can depend on many of the other modules (e.g. it uses *shared-db* for spinning up a test database, *shared-messaging* for Kafka containers, etc.) but since it’s not used at runtime by the services, this does not affect production dependency flow.

The project uses a **Bill of Materials (BOM)** module (*shared-bom*) to manage dependency versions centrally. The BOM ensures all microservices use consistent versions of third-party libraries (Spring Boot, Spring Cloud, Kafka clients, etc.) and shared components. This prevents version drift and incompatibilities across services. All modules share a parent Maven POM that references the BOM for dependency management.

To enforce the architectural rules, we use automated checks (via [ArchUnit](https://www.archunit.org/) tests). These tests assert that no cyclic dependencies exist between packages/modules and that higher-level modules don’t depend on implementation details of lower layers (only on their intended APIs). For example, an ArchUnit rule ensures that *shared-core* never depends on any other *shared-* module (it should remain the lowest layer). Another rule might enforce that *shared-rest* and *shared-observability* do not depend on *shared-messaging* (since web/observability concerns are separated from messaging). These checks help maintain a clean modular structure as the library evolves. If a developer accidentally introduces a disallowed dependency, the build will fail, prompting a refactoring to maintain the correct layering.

**Figure 1: Module dependency structure.** Lower-level modules (e.g., *core*) provide foundations used by higher-level modules (*security*, *db*, *messaging*, *observability*, *rest*). The arrows indicate allowed compile-time dependencies (e.g., *shared-security* → *shared-core* means security depends on core). No cyclic dependencies are permitted, and *shared-core* remains independent.

*Figure 1: LMS Shared Library modules and their dependency flow (arrows point from dependent module to the module it uses). The shared-core module is the foundation for most others. Higher-level modules like shared-rest, shared-messaging, and shared-observability depend on core (and sometimes on other base modules) but not vice versa. This layering enforces a one-directional dependency graph.*

## Shared Core Module

**Purpose:** The *shared-core* module provides the building blocks and common utilities that all other modules (and microservices) rely on. It has no external dependencies on other shared modules, making it a lightweight foundation. Key responsibilities of *shared-core* include:

* **Base Domain Entities:** Abstract base classes for JPA entities, such as BaseEntity (which provides a primary key ID, tenantId, and audit timestamps), AuditableEntity (extends BaseEntity to also include createdBy/updatedBy fields for audit), and SoftDeleteEntity (adds a deleted flag and a Hibernate filter for soft deletes). These are meant to be extended by actual entity classes in services so that every table automatically has tenant scoping and timestamps. For example, a service’s Member entity would extend BaseEntity, inheriting id, tenantId, createdAt, and updatedAt fields. This design choice ensures **consistency**: all entities have standard fields and are identifiable across services. It also centralizes the multi-tenancy mechanism – e.g., the base classes and associated Hibernate filters use the tenantId to enforce row-level isolation (see **Shared DB** module).
* **Request Context Propagation:** A RequestContext class (backed by a thread-local or similar) is provided to hold information about the current request: e.g. the tenant ID, user ID, and correlation/trace ID. The core module also includes a RequestContextFilter that runs for each incoming HTTP request to populate the RequestContext and MDC (Mapped Diagnostic Context) with these values. This allows any part of the code to retrieve RequestContext.getTenantId() or RequestContext.getUserId() at runtime to know who is performing the operation and under which tenant context. The design strategy here is to avoid passing context information through multiple method signatures; instead, it’s globally accessible in a controlled manner. **Usage Example:** If a service method needs the current tenant for a business check, it can simply call String tenantId = RequestContext.getTenantId(); – this value was set at the start of the request by the filter (which extracted it from the JWT or request headers). At the end of the request, the filter clears the context to avoid any leakage between requests. This pattern greatly simplifies multi-tenant development: developers don’t need to manually thread the tenant ID through every service and repository call – it’s automatically available.
* **ID Generation Utilities:** *shared-core* provides utilities for generating unique identifiers, such as an ULID or Snowflake ID generator. These are used to produce resource IDs that are globally unique and sortable. For example, the library might include a Ulid class to generate 26-character lexicographically sortable IDs, and a SnowflakeId class (if we use Twitter’s Snowflake algorithm) for numeric IDs. The IdGeneratorConfig in core standardizes how IDs are generated across services (perhaps configuring node/datacenter bits for Snowflake, etc.). The intention is to avoid relying on auto-increment DB IDs (which can conflict in multi-tenant environments or across microservices) and have a unified approach to identifiers.
* **Exception Hierarchy and Global Handler:** Core defines a standard exception hierarchy and error response model. For example, it might have a base BusinessException or DomainException for known business errors, and other runtime exceptions for infrastructure issues. The *shared-core* (or shared-rest, see below) provides a **Global Exception Handler** (likely a Spring @ControllerAdvice) that catches uncaught exceptions in any service and converts them to a standardized HTTP JSON response. The standard error model might follow the [RFC 7807 Problem+JSON](https://datatracker.ietf.org/doc/html/rfc7807) format or similar. Core defines classes like ErrorResponse or ProblemDetails to represent these errors. The global handler ensures every error payload includes useful info like a trace ID, timestamp, and an error code, so that debugging is easier and clients get consistent messages. (The actual implementation of the Problem+JSON builder and handler is in *shared-rest*, but core holds some base exception classes and perhaps a basic handler that *rest* enhances).
* **Utility Classes:** Common utilities such as string or JSON helpers are in core for broad use. For instance, a Jsons utility might wrap an object to JSON conversion (using Jackson) for consistent configuration (ensuring, say, the same ObjectMapper settings across services), or Preconditions might provide assertions for method arguments, etc. Date/time utilities (like ClockConfig that sets the default ZoneOffset.UTC for the JVM) are also here. By centralizing these, we avoid duplicating helper code in each microservice.
* **Pagination and Envelope Objects:** Core may define generic response envelope classes like ResponseEnvelope<T> and PageEnvelope<T>. These wrap API responses with metadata (e.g., wrapping a result in a standard JSON structure that includes status or trace info) and standardize pagination format for list endpoints. Services can return a PageEnvelope (with items and pagination details) from repository queries and the rest layer will directly use it, ensuring all services paginate data in the same way.

**Design Choices:** The core module is deliberately kept free of heavy framework configurations – it mostly contains plain Java classes and simple Spring components. This makes it safe for any module to depend on core without risk of circular initialization or large footprint. For example, RequestContext is a simple POJO with static access, not requiring Spring to be functional (though the filter that populates it is Spring-managed). Similarly, base entities in core define common fields but actual JPA configurations (like the @Entity annotation or @MappedSuperclass) can be handled in each service or in *shared-db* to keep core decoupled from JPA specifics. This separation means core can be reused in contexts outside of a Spring Boot web app if needed (e.g., perhaps in a standalone batch process, the RequestContext might be set manually).

**Using RequestContext and GlobalExceptionHandler:** To illustrate how *shared-core* components are used, consider a simple flow: A controller calls a service method that needs to ensure an operation is within the user’s tenant. Inside the service method, without needing to pass tenantId around, the developer writes:

String currentTenant = RequestContext.getTenantId();  
if (!currentTenant.equals(resource.getTenantId())) {  
 throw new AccessDeniedException("Cross-tenant access not allowed");  
}

If this exception is thrown, the GlobalExceptionHandler (configured via *shared-rest*) catches it and converts it to a 403 Forbidden HTTP response with a JSON body like:

{  
 "status": 403,  
 "error": "ACCESS\_DENIED",  
 "message": "Cross-tenant access not allowed",  
 "traceId": "<trace-id>"  
}

Here we see the interplay: *shared-core* provided the RequestContext and base exception; *shared-rest* turned the exception into a uniform HTTP response. Developers do not need to manually craft error responses – they simply throw exceptions or use the Problems utility (discussed later), and the library does the rest. This significantly **simplifies error handling** and yields consistent results across all microservices.

## Shared Security Module

**Purpose:** The *shared-security* module integrates **authentication and authorization** concerns into our microservices in a consistent manner. It builds on Spring Security (or the chosen security framework) to provide JWT validation, role-based access control, and multi-tenant awareness in the security layer. Rather than each service writing its own security config, *shared-security* offers auto-configuration and helpers that every service can use to require JWT auth on incoming requests and enforce roles/permissions. Key features include:

* **OIDC/JWT Resource Server Config:** The module likely provides a Spring Boot auto-configuration that sets up the service as a JWT **resource server**. This means it configures the JWT decoder (with the public key or JWKS URL of the identity provider) and requires that all requests have a valid JWT access token. The configuration may be as simple as importing *shared-security* – the service will automatically reject requests without a valid Authorization header. This encapsulates the standard security setup so teams don’t have to copy-paste it. For example, *shared-security* might define a SecurityAutoConfiguration that adds a Spring Security filter chain requiring authentication on all API paths (except maybe health checks) and scopes/roles as appropriate.
* **JWT Claims Extraction (Tenant and Roles):** We embed multi-tenancy in JWTs via a claim (e.g., tenant\_id). *Shared-security* includes a custom JWT converter (JwtAuthConverter) that maps JWT claims to Spring Security authorities[[21]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=SecurityAutoConfiguration). For example, it reads the tenant\_id claim and might store it in the SecurityContext (or directly into our RequestContext). It also maps token claims for roles or scopes into GrantedAuthority objects (prefixing roles with ROLE\_ as needed)[[21]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=SecurityAutoConfiguration). This way, within service code, you can use Spring’s @PreAuthorize or our own annotations to enforce that a user has a certain role or scope. The *JwtAuthConverter* ensures that if a user’s token says they have role ADMIN, it’s available as ROLE\_ADMIN in the security context. It also likely extracts user ID, etc., and could place the tenant ID into the RequestContext at filter time.
* **RBAC and Custom Annotations:** To make authorization checks easier, *shared-security* provides annotations like @RequiresScope("SCOPE\_NAME") or @RequiresRole("ROLE\_NAME")[[22]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=JWT%20parser%20to%20extract%20tenant_id%2C,roles%2C%20scopes)[[23]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=JwtAuthConverter%20%28maps%20roles%20%E2%86%92%20ROLE_,scopes%20%E2%86%92%20authorities%2C%20reads%20tenant_id) that can be placed on REST endpoints or service methods. Under the hood, these might be meta-annotations on Spring’s @PreAuthorize with SpEL expressions that check the authorities. For example, @RequiresScope("loyalty.read") could translate to @PreAuthorize("hasAuthority('SCOPE\_loyalty.read')") given how the JWT is converted. The module likely defines constants (e.g., ScopeConstants and RoleConstants) to avoid typos in these strings[[24]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=PurposeOfUse%20header%20support%20,hook). By using these annotations, we ensure consistent enforcement of permissions across services. (If tomorrow the roles or scopes logic changes, we update it in *shared-security* in one place).
* **Multi-Tenancy Enforcement in Security:** An important aspect is **tenant isolation at the request level**. *Shared-security* might include a filter or interceptor that runs after JWT authentication to ensure the authenticated user is only operating within their tenant. For example, a **Correlation/Tenant Filter** in this module could compare the tenant\_id claim from JWT against the path parameter or request header to verify a user from tenant A isn’t trying to act on tenant B’s data (though primarily data isolation is done in the DB layer). More centrally, *shared-security* works with *shared-db* by configuring a TenantFilterInterceptor that sets the Hibernate tenant filter parameter at the start of each session[[25]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=Method%20security%20setup%20with%20Spring,Security). Specifically, after a request is authenticated, *shared-security* can set RequestContext.tenantId (with JWT data) and then *shared-db* sees this and applies session.enableFilter("tenant").setParameter("tenantId", currentTenant). This synergy ensures that **every query is automatically scoped** to the tenant (see *Shared DB* for details). The security module might also enforce tenancy on a higher level – e.g., by injecting a tenantId into outbound RestTemplate calls or Kafka messages headers for tracing multi-tenant flows.
* **JTI (Token Revocation) Support:** To enhance security, *shared-security* follows JWT best practices by respecting the jti (JWT ID) claim for token revocation. While JWTs are stateless, if a need arises to revoke a token (say an admin manually invalidates a token or a user logs out), the service should check a revocation list for JTI. *Shared-security* could provide an in-memory or Redis-based **TokenBlacklist** store. On each request, after JWT validation, a filter can check Jwt.getId() against a blacklist. If it’s found (meaning the token was revoked before its expiration), the filter will reject the request (perhaps returning 401 Unauthorized). This is a strategy to handle scenarios like compromised tokens or user logout in systems that otherwise rely on JWT’s expiration. The library might include utilities to add JTI entries to a blacklist (with TTL up to token expiration). For example, a central auth service or an admin tool could call an endpoint to invalidate a token by its JTI, and *shared-security* filters in each service would enforce that. This practice, combined with **short-lived tokens**, mitigates risk from stolen tokens. By having this logic in the shared lib, all services uniformly implement revocation checks rather than each doing it differently.
* **Audit Logging (Purpose of Use):** Often, security is tied to compliance. The module supports propagating a *Purpose-of-Use* or *Reason* header for privileged actions[[26]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=JwtAuthConverter%20%28maps%20roles%20%E2%86%92%20ROLE_,scopes%20%E2%86%92%20authorities%2C%20reads%20tenant_id). For example, if an admin accesses sensitive data, the request might include a header like X-Purpose: audit which is captured. *Shared-security* might provide an audit hook or simply ensure this header is passed into logs via *shared-observability*’s AuditLogger. This ensures that whenever someone uses an elevated token or special claim, we log why that access was performed (useful for compliance with data protection laws).

**Design and Usage:** *Shared-security* is designed to minimize the security configuration burden on each microservice. A typical service’s security config class can simply import or enable *SecurityAutoConfiguration* from the library, and perhaps use provided annotations on controller methods. For example, to restrict an endpoint to admins of the tenant, a developer can write:

@RequiresRole(RoleConstants.TENANT\_ADMIN)  
@GetMapping("/reports/financial")  
public Report getFinancialReport() { ... }

The library ensures that if a non-admin calls this, a 403 Forbidden is returned (Spring Security will throw AccessDeniedException, which our global handler turns into a neat error response). The developer doesn’t need to manually parse JWTs or check roles – the shared filter and annotations cover it. In code, one can also obtain the current user or roles via Spring’s Authentication object or directly from RequestContext (if we synchronized these, e.g., *shared-security* might set RequestContext.setUserId(principal) so that information is accessible outside of Spring Security classes).

One crucial design choice is how to **enforce tenant isolation**. We rely more on the data layer to prevent cross-tenant data access (via query filters), but we also practice *defense in depth* at the service layer. For instance, if a request includes a path parameter for tenantId, the service can verify it matches RequestContext.getTenantId(). We could provide an aspect or interceptor to do that check automatically for endpoints that contain {tenantId} in their URL (e.g., ensure a user from tenant X can’t call /tenants/Y/resource for Y != X). This might be implemented via an annotation like @TenantRestricted on controllers or a convention that any method with parameter tenantId triggers a check. While not explicitly stated, it aligns with our security practice of **never trusting client-supplied tenant IDs** – always derive it from the token. The library’s inclusion of tenant info in JWT and filters ensures the backend predominantly relies on that token-derived tenant.

Overall, *shared-security* standardizes authentication (with JWT) and authorization (with roles/scopes and tenant context) across the LMS. It reduces misconfiguration risks – e.g., ensures every service uses the same token validation rules (signature, expiry, allowed issuers) and that any change (like rotating a public key or adding a new required scope) can be made in one place.

## Shared Database Module

**Purpose:** The *shared-db* module addresses database connectivity, multi-tenancy at the data level, and common persistence patterns for all services. Its responsibilities include data source configuration, entity manager customization (for tenant filtering), base repository definitions, and database migration support[[27]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=shared). In essence, this module encapsulates how our microservices interact with PostgreSQL (our chosen DB) in a consistent, safe way. Key features:

* **DataSource and Connection Pool Config:** *Shared-db* likely provides a pre-configured **HikariCP** DataSource tuned for our environment[[27]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=shared). For example, it can set connection pool sizes, timeouts, etc., suitable for our 99.9% uptime and performance requirements[[28]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=Availability%3A%2099.9,for%20portal%20UI). It centralizes the JDBC URL and credentials retrieval (potentially from environment or Vault) so individual services don't each define it. This ensures that all services use the same best-practice settings (like TCP keepalive, proper Unicode, etc.). If, for instance, we need to enable a new PostgreSQL parameter, we update *shared-db* config and all services pick it up on next release. Additionally, *shared-db* might support multi-tenancy at the DB level by either using separate schemas or a single schema with tenantID column – in our design we use a **single schema approach with row-level tenant isolation**[[29]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=Security%3A%20MFA%20for%20admins%3B%20RBAC%2FABAC%3B,trails%20immutable%3B%20data%20encryption%20KMSmanaged) (since the core has tenantId in BaseEntity). Thus, the DataSource is not multi-tenant itself (we’re not using schema-per-tenant switching at runtime), but the queries are filtered for tenant.
* **Hibernate Tenant Filter:** A cornerstone of *shared-db* is the Hibernate/JPA configuration that automatically adds tenant\_id = :tenantId to all queries on tenant-scoped entities. This is accomplished by defining a global Hibernate filter and enabling it per session. Specifically, *shared-db* defines a @FilterDef(name="tenant", condition="tenant\_id = :tenantId") on BaseEntity (or an abstract class) and a TenantFilterConfig to register it. There is also a TenantFilterInterceptor (likely a Spring SessionInterceptor or Hibernate EmptyInterceptor) that runs whenever a new EntityManager session is opened, and if a RequestContext.tenantId is present, it enables the filter with that value. In practice, this means developers do not have to manually add where tenant\_id=... in queries; the library does it. For example, if a repository calls findAll() on MemberRepository, under the hood Hibernate will append ... where tenant\_id = 'TENANT-001'. Similarly, on inserts, it’s important to set the tenant\_id field. Our BaseEntity might mark tenantId as updatable = false and not-null, and we rely on setting that field automatically. The interceptor could also intercept any persist operation: if an entity extends BaseEntity and tenantId is null, the interceptor sets it from RequestContext before flush. Alternatively, we might rely on application logic to set tenantId (e.g., when creating new entities, the service sets entity.tenantId = RequestContext.getTenantId()). Either way, *shared-db* enforces that no row goes into the database without a tenant\_id, and no query reads data across tenants[[29]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=Security%3A%20MFA%20for%20admins%3B%20RBAC%2FABAC%3B,trails%20immutable%3B%20data%20encryption%20KMSmanaged). This effectively implements **application-level row-level security**, analogous to PostgreSQL’s RLS feature but enforced by Hibernate. The design is such that multi-tenancy is transparent to developers: as long as they extend BaseEntity and call repository.save(), the entry gets the right tenant, and any find queries only return that tenant’s data.
* **Base Repository:** The module may provide a BaseRepository<T,ID> interface that extends Spring Data JPA’s JpaRepository[[30]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=per%20session). This can include some common methods or custom behaviors. For instance, if we support **soft deletes**, BaseRepository might override the delete methods to actually perform a soft delete (set deleted=true) and provide a findAllActive() to get only not-deleted records. It could also include utility methods such as findByIdOrThrow(id) which throws a standardized exception if not found (so services can reduce boilerplate). By using a base repo, we ensure consistent patterns (like how we check for entity existence or how we handle optional returns).
* **JSONB and Custom Types:** Given PostgreSQL’s support for JSONB, *shared-db* likely integrates **Hibernate Types** for JSONB fields. It might include a JsonbAttributeConverter or use the hibernate-types library to allow mapping Java objects or Map<String,Object> to JSONB columns[[31]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=BaseRepository). In our domain, we often store dynamic JSON (e.g., voucher features or tenant settings) – *shared-db* ensures there is a consistent way to map these to entities. It could also handle other database types like Postgres enums or arrays if needed. The inclusion of these converters in one place means all services handle JSON data in the same way (for example, always using Jackson with certain settings to serialize, thus ensuring no loss of data).
* **Database Migrations and Flyway Hooks:** Managing schema migrations is critical in microservices. The module includes Flyway (our chosen migration tool) configuration and possibly base SQL migrations. For example, *shared-db* might apply baseline migrations for common tables like idempotency\_keys or outbox if those are expected in every service. It could also contain Flyway callbacks to enforce rules – e.g., a callback that tags each migration with the service name or prevents certain risky operations. The snippet mentions *Flyway base migrations & callback hooks*[[32]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=JSONB%20support%20via%20Hibernate%20types)[[33]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=JsonbAttributeConverter%20for%20Map), which suggests that *shared-db* might have a migration version 1\_\_Base.sql that creates some shared infrastructure tables (like audit log table, idempotency key table, etc., that every service should have). The callback could, for instance, automatically record the Git commit or build version at the time of migration (for traceability). Also, it might include an *Anti-drift* mechanism: a callback that checks the database for unexpected changes (maybe comparing a hash of schemas or ensuring all migrations have been applied in order). By centralizing migration config, we ensure all services use the same Flyway settings (like naming conventions, clean disabled, etc.).
* **Soft Deletes:** Soft delete functionality is often needed (to mark data as deleted without physically removing it). The core provided SoftDeleteEntity and *shared-db* likely defines a global @Where(clause="deleted = false") on that entity and any subclass, plus a filter similar to tenant filter for deleted flag[[34]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=TenantJpaFilter%20to%20inject%20tenant_id%20on,insert%20and%20filter%20queries)[[35]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=Flyway%20helpers%3A%20FlywayConfigurer%2C%20callback%20to,tag%20builds%20%26%20block%20drift). It means queries automatically exclude soft-deleted rows unless explicitly requested. The BaseRepository might provide softDelete() or so. This ensures uniform handling of logically deleted records across services (e.g., membership records or transactions might be soft-deletable for audit).

**Design Choices:** The approach taken by *shared-db* is to implement multi-tenancy at the application level (via filters) rather than at the physical DB (no separate schema or separate DB per tenant). This was chosen for simplicity and resource efficiency – maintaining separate schemas for potentially many tenants can complicate deployments and migrations. Instead, a tenant\_id column is present in all multi-tenant tables[[29]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=Security%3A%20MFA%20for%20admins%3B%20RBAC%2FABAC%3B,trails%20immutable%3B%20data%20encryption%20KMSmanaged) and the library guarantees it’s always used in queries. This choice relies on developer discipline to always use the JPA repositories (which apply filters) rather than writing native SQL that could bypass them. To enforce this, code reviews and static analysis can be used (for example, an ArchUnit rule might forbid usage of @Query(nativeQuery=true) without special approval, to avoid accidental cross-tenant queries).

*Shared-db* works closely with *shared-core* and *shared-security* to fully implement multi-tenancy. For example, upon each web request, after *shared-security* extracts the tenant\_id from JWT, it calls something (perhaps sets a thread-local) that *shared-db*’s entity manager interceptor picks up to set the filter. This coordination is internal and hidden from service code.

**Using the Tenant Filter:** Suppose a developer writes a Spring Data repository query like List<Order> findByStatus(String status). They expect to get orders for the current tenant. Thanks to *shared-db*, the generated SQL will effectively be SELECT \* FROM orders WHERE status = ? AND tenant\_id = ?. The tenant\_id parameter is injected automatically (no need for the method signature to include it). When the query executes, the *TenantFilterInterceptor* has already set the value of :tenantId to (say) 'TENANT-001'. So the developer just gets back the current tenant’s orders. If by mistake they attempted to bypass and, for example, use EntityManager.createQuery("FROM Order") outside the repository, the filter still applies because it’s at the session level. Only if someone explicitly disables the filter or writes raw SQL would they see other tenants – and doing so is against our architecture guidelines (and would be caught in tests for tenancy leaks).

Another feature is **Idempotency Keys** at the DB level (discussed more in *shared-rest*). Often these are stored in a table in each service’s DB. *Shared-db* could define the JPA entity for an IdempotencyKey record and even a repository for it. In practice, we have an idempotency\_keys table (with columns: tenant\_id, key, first\_seen\_at, response\_hash, status)[[36]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=Idempotency). *Shared-db* might not implement the full logic, but providing the schema and entity ensures each service has this table and can use it via *shared-core*’s IdempotencyKeyStore.

**Migration Example:** When a new microservice is created, it can include *shared-db* in its dependencies. Immediately, that service gains the base entities and the filter config. Also, if *shared-db* includes a base Flyway migration, that migration will run on service startup to create common tables. For instance, migration V1\_\_baseline.sql might contain:

CREATE TABLE idempotency\_keys (  
 tenant\_id UUID NOT NULL,  
 idempotency\_key VARCHAR(100) NOT NULL,  
 first\_seen\_at TIMESTAMP NOT NULL,  
 response\_hash VARCHAR(500),  
 status VARCHAR(20),  
 PRIMARY KEY (tenant\_id, idempotency\_key)  
);

and

CREATE TABLE outbox\_messages (  
 id UUID PRIMARY KEY,  
 tenant\_id UUID NOT NULL,  
 topic VARCHAR(100) NOT NULL,  
 key VARCHAR(100),  
 payload JSONB NOT NULL,  
 status VARCHAR(20) NOT NULL,  
 created\_at TIMESTAMP NOT NULL  
);

(This is a guess, but likely since *shared-messaging* deals with outbox, that table’s creation might also be automated). The above ensures idempotency and outbox patterns (see next sections) are ready to use with minimal setup in each service.

In summary, *shared-db* enforces **data layer rules**: each insert or query is automatically tenant-scoped and consistent with soft deletes, JSON handling, etc. It frees developers from writing repetitive code for these concerns and significantly reduces the risk of multi-tenant bugs (like forgetting to filter by tenant and accidentally exposing data). We also implement **tenancy-leak tests** (see **CI/CD and Testing**) to validate that our filters work – e.g., test cases may create data in tenant A and ensure that when running under tenant B context, none of A’s data is accessible.

## Shared Messaging Module

**Purpose:** The *shared-messaging* module provides a standardized way for microservices to publish and consume events, particularly via **Apache Kafka** (our chosen event bus). It implements patterns like the **Transactional Outbox** for reliable event delivery and **idempotent consumer** logic for safe processing of events. By using *shared-messaging*, all services follow the same conventions for topic names, message envelopes, and error handling in the event pipeline[[37][38]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=shared). Key components include:

* **Kafka Configuration and Factories:** The module likely autoconfigures Kafka Producer and Consumer factories with sensible defaults[[37][39]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=shared). For producers, it might enable **transactions** so that we can use the outbox pattern (ensuring atomic DB + Kafka writes). For consumers, it sets up listener containers with error handling and maybe manual ack mode (to better control commit after processing). It also centralizes Kafka connection properties (bootstrap servers, serializers, etc.) to avoid duplication. Additionally, *shared-messaging* can define a **topic naming convention** – for instance, topics are named as ${ENV}.${tenantDomain}.events or ${env}.${service}.{entity}.events[[40]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=Kafka%20producer%2Fconsumer%20configuration)[[41]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=ConsumerOffset). A convention mentioned is {env}.{domain}.events with message keys as {tenantId}:{entityId}[[41]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=ConsumerOffset). This way, if we see a topic like prod.loyalty.events, we know it’s carrying domain events for loyalty service in production, and keys like TENANT-001:MEMBER123 indicate the tenant and entity. The library might provide helper methods to format topic names and keys to reduce mistakes.
* **EventEnvelope<T>:** This is a generic wrapper object for events dispatched on Kafka[[42]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=shared)[[43]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=KafkaProperties%20%26%20KafkaAutoConfig%20,with%20transactions). Instead of publishing raw JSON or avro records, services use an EventEnvelope<T> that includes metadata like tenantId, correlationId, traceId, and a schema version for the payload[[43]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=KafkaProperties%20%26%20KafkaAutoConfig%20,with%20transactions). The payload T could be a domain event object (e.g., MemberCreatedEvent with specific fields). By using an envelope, we ensure every message on the bus has the necessary context to trace it and route it. For example, when an event is published, *shared-messaging* will automatically populate the envelope’s traceId from the current request (so that if event processing later fails, we can link it back to the originating request trace) and correlationId for chaining events. tenantId in the envelope ensures the consumer knows which tenant the event belongs to (this is redundant if topics are isolated by tenant, but we usually use multi-tenant topics with the tenant in message key or header). The schemaVersion field helps with evolution of event payloads – if we update an event structure, we bump its version, and consumers can handle compatibility. The envelope concept is crucial for observability and debugging: every event log can be tied to a service trace.
* **Event Publisher (Outbox Pattern):** To avoid the dual-write problem (writing to DB and sending a Kafka message in one go), *shared-messaging* implements the **Transactional Outbox** pattern[[44]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=Topic%20naming%20convention%20%28)[[45]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=REST%20writes%20require%20Idempotency,tenant_id%2Ckey%2Cfirst_seen_at%2Cresponse_hash%2Cstatus%29%20with%20TTL). The library provides an OutboxMessage JPA entity representing an event to be sent (with fields like id, topic, key, payload, status, createdAt)[[46]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=Outbox%20pattern). When a business event occurs (say an Order is placed), instead of producing to Kafka directly, the service writes an OutboxMessage to its DB within the same transaction as other changes. A separate *Outbox Relay* component then reads these outbox rows and publishes to Kafka asynchronously[[47]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=JPA%20OutboxMessage,createdAt%2C%20status).

*Shared-messaging* likely includes the logic for this Outbox Relay. Possibly it offers two modes: polling or DB trigger-based. Polling means periodically (e.g., every few seconds) checking the outbox table for new messages with status “PENDING”, and sending them. Trigger-based might use PostgreSQL’s LISTEN/NOTIFY to get immediate notifications of inserts. The snippet suggests support for NOTIFY/LISTEN[[48]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=JPA%20OutboxMessage,createdAt%2C%20status). In either case, the actual send to Kafka is done in a **transactional producer** – meaning the KafkaProducer is configured with transactional.id and all sends are wrapped in a transaction that also marks the DB row as sent. Typically, the algorithm is: fetch PENDING outbox messages → for each, begin Kafka transaction → send all messages in this batch to Kafka → commit Kafka transaction → update those outbox rows to status “SENT”. This ensures if the process crashes mid-way, uncommitted Kafka messages won’t appear but the DB remains unchanged (so it can retry). If it sends and crashes before marking SENT, on restart it may resend the same event, but that’s okay because consumers are idempotent or deduplicate by message ID. Each OutboxMessage has a unique ID (likely the DB primary key which we also use as Kafka message key or part of it). Consumers can use that to detect duplicates (see next point).

The *shared-messaging* module provides an EventPublisher abstraction to simplify the service code. A service simply calls something like eventPublisher.publish(new MemberCreatedEvent(member)). Under the hood, *shared-messaging* will wrap that event into an EventEnvelope, serialize it (perhaps to JSON or Avro), and persist an OutboxMessage entity. It may even flush the persistence context to ensure the outbox insert is done. The developer doesn’t have to write any Kafka code; they just call publish and commit their transaction. The actual Kafka publishing happens in the background. This design was chosen to **guarantee atomicity**: either both DB and event are saved, or neither, eliminating inconsistent states[[49]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=enabled%20via%20an%20interceptor%20that,reads%20JWT%20tenant_id). It trades a bit of complexity (running an outbox relay thread or component) for reliability. The library likely runs the Outbox Relay on a separate thread pool within each service or uses a lightweight Spring scheduler for it. It might also be smart about backpressure, etc.

* **Idempotent Consumer:** In a distributed system, consumers might see duplicate messages (if producer retried or broker delivered twice). Also, with the outbox pattern, if the relay crashes after sending but before updating DB, a duplicate send can occur on restart. Therefore, *shared-messaging* provides an **Idempotent Kafka Listener** base class or aspect[[50]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=Outbox%20pattern%20utilities%20with%20transactional,producers)[[51]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=Idempotent%20consumer). When services consume events, they should extend or use this to automatically handle deduplication. The scheme is usually: maintain a **ConsumerOffset** (or *ProcessedMessage*) store in the database, recording the unique message IDs that have been processed[[52]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=Idempotent%20consumer). This could be a table with columns: message\_id (maybe the outbox GUID or a combination of topic+partition+offset), processed\_at timestamp. On receiving an event, the IdempotentListener checks this store – if the message\_id is already present, it ignores the event (or immediately commits offset). If not, it processes the event and then inserts the message\_id into the store, all within a local transaction around the consumer logic. To assist this, *shared-messaging* might provide a template method or an annotation that wraps the consumer method in a transaction and automatically does the book-keeping. For example, a service could declare a listener like:

@KafkaListener(topics = "loyalty.events", groupId="loyalty-service")  
public void onMemberEvent(EventEnvelope<MemberEvent> event) {  
 // business logic  
}

By annotating the class with something like @IdempotentKafkaListener, the library could intercept this call: before invoking, it checks if event.getHeaders().getMessageId() is in DB. If yes, skip; if no, proceed. After successful processing, it records the ID. This ensures **at-least-once** processing is safe – duplicates don’t lead to double side effects. The ConsumerOffset table basically serves as an **exactly-once** mechanism at the application level (within a single service instance). It’s worth noting this is needed even if Kafka is in *at-least-once* mode; if we ever used *exactly-once* Kafka features, we might still keep this for safety or for cross-instance deduping.

* **Schema Registry and Serialization:** The module could integrate with a schema registry if we use Avro or JSON schemas for event payloads[[53]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=Idempotent%20consumer%20handler%20with%20dedupe,store). It might provide a client that fetches schemas and configures Kafka ProducerFactory/ConsumerFactory to use the registry (e.g., Confluent Schema Registry). If using simple JSON, it may not be needed. But the mention of *Schema registry client (Avro/JSON Schema)*[[53]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=Idempotent%20consumer%20handler%20with%20dedupe,store) suggests support for Avro messages. Possibly, *shared-messaging* can automatically register schemas for event types and encode/decode the EventEnvelope accordingly. This again encourages consistency: all events of type X share the same schema and version, known to producers and consumers. The library handling this means developers focus on the content of events, not the serialization details.

**Event Processing Flow:** The typical flow with *shared-messaging* in action is depicted in **Figure 2**. It shows how an event generated in Service A’s database is delivered to Service B, with the outbox and idempotency steps.

【40†source】[[45]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=REST%20writes%20require%20Idempotency,tenant_id%2Ckey%2Cfirst_seen_at%2Cresponse_hash%2Cstatus%29%20with%20TTL)

* In Service A, during a transaction (for example, creating an order), the application calls eventPublisher.publish(OrderCreatedEvent) instead of directly hitting Kafka. The library creates an OutboxMessage in Service A’s DB with status “PENDING”. The transaction commits, so now the outbox table has a record of an event to send (along with the newly created order data).
* The Outbox Relay (running in Service A) picks up the new OutboxMessage (it might get a DB notification or just see it on the next poll)[[47]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=JPA%20OutboxMessage,createdAt%2C%20status). It starts a Kafka transaction, sends the event to topic (say, prod.order.events with key containing tenant), and upon success, marks the outbox row as “SENT”. The row might remain for auditing or be cleaned later. Now the event is in Kafka with a unique message ID (perhaps the outbox UUID).
* Service B has a Kafka listener for order.events. When Kafka delivers the message, the *shared-messaging* IdempotentListener in Service B intercepts. It checks Service B’s ConsumerOffset table to see if this messageId was seen. If not, it records it (often one would record after processing, but it can also record before to guard against parallel processing of same message, depending on design). It then unwraps the EventEnvelope, sets up the RequestContext in Service B (so that if Service B needs to call another service or log, it has the original correlation and tenant info carried in the envelope), and calls the user’s listener logic. The business logic in Service B executes (e.g., increment a user’s points balance on receiving an OrderCreatedEvent). If that logic triggers further events, Service B might also use its outbox to emit them – the correlationId from the envelope ensures trace continuity. After successful processing, it finalizes by inserting the messageId into ConsumerOffset (if not done earlier) and committing the Kafka offset. If the processing failed, the framework can retry or send to a dead-letter topic depending on config, but importantly, the ConsumerOffset not being saved means it will try again on restart (preventing message loss).

This flow ensures **guaranteed delivery and processing**: either Service B will process the event exactly once (under normal operation), or if something fails, the system is designed to either retry safely or handle duplicates gracefully. No event is lost between Service A and B as long as at least one of them remains operational.

**Figure 2: Outbox event processing and idempotent consumption.** Service A saves an event to its database outbox as part of a transaction, and later a relay service publishes it to Kafka. Service B’s consumer receives the event, checks if it has seen the event ID before (idempotency check), and if not, processes it and records the ID to avoid duplicate processing.

*Figure 2: Transactional Outbox and Idempotent Consumer flow. Service A’s Outbox Relay reads new events from the Outbox table in the database and publishes them to Kafka (ensuring that database changes and events stay in sync). Each event is wrapped with metadata (tenant, traceId, etc.). Service B’s consumer receives the EventEnvelope, uses the embedded messageId to check against a ConsumerOffset store (to avoid duplicates), and processes the event. This pattern guarantees reliable, exactly-once processing of cross-service events*[*[49]*](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=enabled%20via%20an%20interceptor%20that,reads%20JWT%20tenant_id)[*[54]*](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=Outbox%20pattern)*.*

* *Note:* In our implementation, we use at-least-once Kafka semantics with application-level deduplication for exactly-once processing. The Outbox ensures events are only published for committed transactions, and the ConsumerOffset ensures consumers don’t apply the same event twice[[49]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=enabled%20via%20an%20interceptor%20that,reads%20JWT%20tenant_id). We chose this over Kafka’s own Exactly-Once Delivery mainly for flexibility and transparency – the data of pending events is in a regular table we can monitor, and it decouples the services via persistent storage.
* **Topic Management:** The library might also assist with creating topics (maybe via an admin client on startup). For example, if a service knows it will publish topic prod.loyalty.events, *shared-messaging* could ensure that topic exists with the correct partitions and replication factor, possibly through config properties. If the infrastructure team pre-creates topics, this may not be needed, but having the code to do it is useful for integration tests (where topics can be auto-created with Testcontainers Kafka). We enforce naming conventions as mentioned, partly to avoid conflicts and to logically group events. The environment prefix (dev/test/prod) prevents cross-env mistakes. Domain grouping (like loyalty.events) allows wildcard consumption if needed and easier monitoring (one can watch all loyalty events).
* **Error Handling & Retries:** *Shared-messaging* likely sets a default error handler on Kafka listeners. For instance, if a consumer throws an exception, we might retry a few times or send the message to a Dead Letter Topic (DLT). The library could configure Spring Kafka’s SeekToCurrentErrorHandler or the newer DefaultErrorHandler to perform, say, 3 retries with backoff, then route to a {topic}.DLT. It might also log the error with the correlationId for debugging. Standardizing this means all services treat poison messages uniformly.

**Usage:** From a developer’s perspective, using *shared-messaging* is straightforward. To publish events, you inject an EventPublisher bean (provided by the lib) and call methods on it. For example, in a service method:

eventPublisher.publish("loyalty.events", new EventEnvelope<>(new PointsExpiredEvent(expiredPoints)));

The library handles transaction synchronization to ensure this call results in an outbox record. To consume, you simply define a method with @KafkaListener and the library’s infrastructure wraps it with dedup checks. If you need custom logic on dedup (maybe some events can be processed multiple times), you can opt out by not using the Idempotent base class, but by default we encourage using it.

By adhering to these patterns, we’ve achieved **loose coupling** between services: they communicate via events asynchronously, and thanks to the outbox, we don’t lose events even if a service or Kafka is temporarily down. For example, if Kafka is down, the outbox messages remain PENDING and will be retried until Kafka is back (ensuring no data loss). And if a consumer service is down, Kafka retains the events until it comes back, and then it processes them once. Our architecture, as implemented by *shared-messaging*, thereby supports eventual consistency between services in a robust way.

## Shared Observability Module

**Purpose:** The *shared-observability* module centralizes logging, metrics, and tracing configuration so that all services emit observability data in a consistent format[[55]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=shared)[[56]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=OpenTelemetry%20exporter%20autoconfig). It helps our SREs and developers to monitor the system health and debug issues by providing standardized logs (with JSON structure and correlation IDs), metrics instrumentation (via Micrometer), and distributed tracing (via OpenTelemetry). Key features include:

* **OpenTelemetry Auto-Instrumentation:** *Shared-observability* integrates **OpenTelemetry (OTel)** for tracing[[55]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=shared)[[56]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=OpenTelemetry%20exporter%20autoconfig). It likely configures the OpenTelemetry SDK with exporters (e.g., to Jaeger or Zipkin or an OTLP endpoint). It might include the OTel Java Agent or manual instrumentation of key points. For instance, it can automatically trace HTTP server requests and Kafka message processing by using Spring Cloud Sleuth or OTel instrumentation libraries. The configuration ensures that a trace ID is generated at the entry point of each service (if not already present) and propagated to outgoing calls. The module might provide beans for Tracer and setup context propagation. It definitely ensures that the *Correlation ID* we use in logs is tied to the trace ID or at least carried along (possibly the same value or mapped). The result is that for any given transaction (say an API call in Service A that triggers an event to Service B), we can correlate logs and metrics across service boundaries by a common trace or correlation ID. This is crucial when debugging a multi-service issue – one can search in the log aggregation system for a specific correlationId and get the complete story across services.
* **Micrometer Metrics Registry:** All services use Micrometer for metrics, and *shared-observability* provides a pre-configured **MeterRegistry** (for example, integrating with Prometheus or Elastic APM)[[57]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=OpenTelemetry%20exporter%20autoconfig). It likely auto-registers common tags on all metrics, such as serviceName, environment, and maybe tenant if applicable[[58]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=OpenTelemetry%20exporter%20autoconfig). It also might define **SLA-focused timers** for key operations. For instance, it could use the @Timed annotation on controllers or specific methods, and a MeterRegistryCustomizer to ensure those metrics have uniform naming (http.server.requests with tags like outcome, status, etc.). By including this module, services automatically collect: HTTP request counts/latencies, Kafka consumer lag, DB connection pool metrics, JVM stats, etc. The module may expose internal metrics like *tenant counts* or others if needed. With unified metrics, our dashboards and alerts can be consistent across services (we know that http.server.requests{service="member-service", status="500"} exists for all services and can be aggregated). The module also sanitizes metrics for multi-tenancy – e.g., we might tag metrics by tenant for internal monitoring, but be cautious not to produce high-cardinality metrics if tenant IDs are numerous. More often, tenant is not a tag in metrics (to avoid cardinality blow-up), but if we have some partition (like environment or region) those are included.
* **Structured Logging with PII Scrubbing:** Logging is configured via *shared-observability* using a JSON encoder for Logback (or Log4j) so that each log event is a JSON object with predefined fields[[59]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=Logback%20JSON%20encoder%20%2B%20PII,scrubber%3B%20CorrelationIdFilter%3B%20AuditLogger%20facade). These fields likely include timestamp, level, logger, message, thread, etc., as well as our **custom context**: correlationId, traceId, tenantId, userId (if available), and possibly spanId from OTel. The module might supply a Logback configuration file or programmatic config to output logs in JSON. It also implements a **PII (Personally Identifiable Information) Scrubbing** filter[[59]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=Logback%20JSON%20encoder%20%2B%20PII,scrubber%3B%20CorrelationIdFilter%3B%20AuditLogger%20facade). This could be a Logback TurboFilter or a custom appender that scans log messages for sensitive data patterns (like email addresses, phone numbers, national IDs) and masks them. Alternatively, it can provide utilities for developers to mark certain log fields as sensitive, which the logger will then redact. For example, if a service logs an object containing a field ssn or password, the output JSON might show "ssn": "[REDACTED]". This is important for compliance (PDPL/GDPR)[[60]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=Compliance%20%26%20Security) and also to avoid leaking secrets in logs. *Shared-observability* likely contains a list of regex patterns (like \d{14} for IDs, or credit card regex) to mask, and ensures that all logs pass through this filter. The result is that developers can log freely, confident that the framework will scrub out any inadvertent sensitive info before it lands in log storage.
* **Correlation ID Propagation:** We use a **Correlation ID** to track a request across services (this is similar to a trace ID, but sometimes used as a business transaction ID). *Shared-observability* provides a CorrelationIdFilter that generates a correlationId for incoming requests (if the client hasn’t sent one) and stores it in MDC[[59]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=Logback%20JSON%20encoder%20%2B%20PII,scrubber%3B%20CorrelationIdFilter%3B%20AuditLogger%20facade). This correlationId is then propagated: the module ensures any outgoing HTTP calls (using RestTemplate or WebClient) carry this ID, and any outgoing Kafka events include it in the EventEnvelope headers[[59]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=Logback%20JSON%20encoder%20%2B%20PII,scrubber%3B%20CorrelationIdFilter%3B%20AuditLogger%20facade). On the consumer side, it picks it up from the message and puts it in the MDC again. Thus, all log entries across all microservices for a given high-level action share the same correlationId. This is invaluable for troubleshooting distributed processes. For example, a user action that flows through 3 services can be traced by searching logs for one ID. The correlationId may be the same as the traceId from OTel, or we might separate them (some teams do both: use traceId for internal tracing and correlationId for external linking, but often they are unified). In our case, likely we use one ID for simplicity. The *CorrelationIdFilter* in *shared-rest* (mentioned in *shared-rest* module) and the similar logic in *shared-observability* work together to ensure every request/response has this header (perhaps using X-Correlation-ID). Also, *shared-observability* could add this ID to response headers so that the client calling the service can know the correlationId (useful if they need to report an issue with that ID).
* **Audit Logging:** In addition to application logs, we maintain **audit logs** for security-critical events. *Shared-observability* might include an AuditLogger facade[[59]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=Logback%20JSON%20encoder%20%2B%20PII,scrubber%3B%20CorrelationIdFilter%3B%20AuditLogger%20facade) that writes structured audit events (perhaps to a separate appender or Kafka topic). For example, when a user updates a setting or an admin views PII, we call AuditLogger.log("EVENT\_NAME", details...) and the library formats it as a JSON audit event with who, what, when, and possibly stores it to an audit\_events index. The idea is to have a tamper-evident trail of important actions (which is different from debug/operational logs). This can be integrated with *shared-security* (where certain actions automatically trigger audit logs, e.g., a purpose-of-use header triggers an audit event that user X accessed sensitive data for reason Y). Observability module could define the schema of audit events and the output channel (file, DB, or Kafka).
* **PII Masking in Traces/Metrics:** Besides logs, PII could accidentally appear in traces (like span names or attributes) or metrics (like username as a metric label). The module can include guidelines or automatic filters to avoid that. For instance, we do not put raw customer data as metric tags. If needed, we hash or categorize them. This may not be a concrete feature in code but a documented practice enforced via code review or ArchUnit.
* **Centralized Logging & Monitoring Integration:** The *shared-observability* might configure exporters: e.g., push logs to Fluentd/ELK, metrics to Prometheus, traces to Jaeger. In code, that might mean just ensuring the right dependencies and properties are set. It could also expose health check endpoints for monitoring (though that might be more *shared-rest* domain). For instance, *observability* might add an actuator endpoint that scrubs any sensitive info in metrics for general exposure.

**Design Choices:** We chose to implement observability through standardized libraries (OTel, Micrometer) but wrapped in our configuration to ease usage. The alternative was each team individually deciding how to log or measure things, which leads to inconsistency. By using *shared-observability*, we ensure: - Every log line in every service can be parsed by the same JSON parser and has key fields (trace, correlation, tenant, etc.). - Every service has a baseline of metrics (like jvm.memory.used, http.server.requests) out of the box, and if they add custom metrics, they use the provided MeterRegistry (so all metrics end up in the same monitoring backend with proper tags). - All services participate in distributed tracing, which is essential for our microservice architecture to diagnose latency or failures across service boundaries.

One important practice is instrumenting **PII scrubbing** comprehensively. This shows the value of doing it in the library: we write the scrubbing logic once and it applies to all logs from all services. The scrubbing might target things like credit card numbers, national IDs, phone numbers (for example, replacing middle digits with \*). We ensure things like passwords are never logged in the first place (by policy) and if they were, our filter would catch any field named “password”.

Another design is linking **observability with multi-tenancy**. We generally avoid per-tenant logs or metrics because of scale, but we do ensure tenant context is present as a field. This allows filtering logs by tenant if needed (e.g., if tenant ACME reports an issue, we can filter logs where tenantId=ACME in the log aggregator). But we might not push tenantId into every Prometheus metric due to potential high cardinality (unless number of tenants is small).

**Usage:** Most of *shared-observability* is behind the scenes – developers do little to use it except ensure their service includes the module and maybe some minimal config. For logging, they simply log using SLF4J as usual (log.info("Processed order {}", orderId)), and the output will automatically include the context fields. They should prefer structured logging (logging key=value pairs) since it’s JSON, e.g., log.info("Processed order id={} for member={}", orderId, memberId). The library’s JSON encoder will output as {"message":"Processed order id=123 for member=456", "orderId":123, "memberId":456, "level":"INFO", "tenantId":"TENANT-001", "traceId":"..."} depending on configuration (some logback encoders allow extracting numbers like that). If not automatically, the developer can explicitly log structured info by constructing a map or using log arguments.

For metrics, if a developer wants to add a custom metric (say count how many vouchers expired), they can use the autowired MeterRegistry from Micrometer (which *shared-observability* set up). For example, Counter counter = meterRegistry.counter("lms.vouchers.expired"); counter.increment();. They don’t worry about where it goes – the library’s config ensures it’s scraped or pushed appropriately. If they annotate a method with @Timed, the library ensures that annotation is picked up and a Timer is registered.

For tracing, if a developer needs to create a custom span (maybe for an external call not automatically instrumented), they can inject the Tracer or use OTel’s APIs. But often, built-in instrumentation suffices (HTTP, Kafka, JDBC are usually auto-traced by OTel if configured). The main practice we encourage is to always pass along the correlation/tracking IDs: e.g., if making an HTTP call manually, include X-Correlation-ID from RequestContext. However, our *shared-observability* plus *shared-rest* likely provide an *HTTP client interceptor* to do this automatically, so manual intervention is minimal.

In summary, *shared-observability* ensures that each microservice is a well-behaved citizen in our monitoring ecosystem, emitting rich telemetry data. When combined with centralized tools (like ELK for logs, Prometheus/Grafana for metrics, and Jaeger/Zipkin for traces), we get a comprehensive view of the system’s behavior. This module reflects our commitment to reliability and easier maintenance – issues can be pinpointed quickly using the uniform logs and traces, and performance can be tracked consistently via metrics.

## Shared Validation Module

**Purpose:** The *shared-validation* module defines custom validation annotations and logic specific to our LMS domain, complementing Java’s standard Bean Validation (JSR 380) features[[61][62]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=shared). By centralizing these, we ensure that common data format rules (like phone number formats or business rules on values) are uniformly enforced across all services. Key validations provided might include:

* **Custom Constraint Annotations:** Examples mentioned are @MobileKSA, @NationalIdKSA, @Money, @EnumValue, @NotFuture[[62]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=shared). These likely correspond to:
* @MobileKSA: Validates that a phone number conforms to Saudi Arabia’s mobile number format (e.g., length and prefix). Instead of each service duplicating this regex, the annotation and its validator reside in shared-validation. So any DTO with a mobile field can just add @MobileKSA and automatically get validation error if format is wrong.
* @NationalIdKSA: Ensures that a given string or number matches the Saudi national ID/Iqama format (which might be a 10-digit number with specific checks).
* @Money(scale, precision): Validates monetary values for correct scale/precision (e.g., at most 2 decimal places, not negative when not allowed, etc.). Potentially ensures a currency value is within expected range.
* @EnumValue: Checks that a string or number is one of the defined constants in a given enum. This is handy for validating input strings (like “gender” or “status”) against an enum type. The annotation might take a class literal of an enum and ensure the value is in Enum.values(). This prevents invalid values from passing in (especially important if not using automatically bound enums).
* @NotFuture: Possibly used for dates to ensure a date field is not in the future (if expecting only past or present, e.g., birth date should not be future).
* **Validator Implementations:** For each annotation, *shared-validation* provides a ConstraintValidator. E.g., MobileKSAValidator that checks a phone number with a regex or maybe a library. These validators are registered via Java’s ServiceLoader or through Spring’s validation config when the module is included. Additionally, the module could include **Normalization** logic as part of validation. For instance, before validating a mobile number, it might normalize by removing non-digits or adding country code if missing. They mentioned *Normalizers for phone numbers and currency amounts*[[63]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=%40NotFuture). It’s possible the validation annotations not only validate but also normalize the value (if they have access to a mutable reference or via a custom binder). More likely, separate util methods (like PhoneNumberNormalizer.normalizeKSA(String phone)) are provided and can be used by services before saving data. We might integrate that by having the validator accept multiple formats but then output in a standard format.
* **Reusable Error Messages:** The module probably defines standardized error messages for these validations, possibly with localization support. So if @MobileKSA fails, the error message might be "Invalid Saudi Arabia mobile number" or a code like error.mobile.invalid. By centralizing, we ensure consistency: the error message for an invalid phone is the same regardless of which service validated it. This is good for client UI/UX (they see a uniform style of error) and easier to maintain (just change in one place if needed). We could integrate it with a message bundle for internationalization if needed (perhaps for Arabic support given KSA context).
* **Cross-Field and Object Validations:** The examples given are mostly single-field, but *shared-validation* could also provide some class-level constraints if needed, for common patterns. For example, maybe an annotation @ValidDateRange that checks if startDate is before endDate, etc., used in scheduling contexts. Or a cross-field check that if one field is X then another must be Y (these would be more domain-specific, not sure if included). But likely, it sticks to field validations.

**Design Choices:** We use Bean Validation (hibernate-validator under the hood likely) which is declarative and integrates with Spring easily (e.g., automatically validating @RequestBody in controllers). By packaging our custom constraints in *shared-validation*, we avoid duplicating logic. It also fosters **correctness** – if, for example, the national ID format rules update (like a new prefix introduced by government), we update the validator in one place, and all services using @NationalIdKSA are instantly using the updated logic after they pull the new lib version. Without this, each service might drift or forget to update, causing inconsistent validation.

Another point is that *shared-validation* might have no external dependencies except perhaps the Bean Validation API. It’s likely purely an API module with annotation definitions and classes, plus maybe some small util dependencies for things like libphonenumber for phone number validation. Keeping it separate means services that don’t need any custom validation might even omit it (though usually it’s small enough that including it is fine).

**Usage:** Using these constraints in a service is straightforward – just annotate DTO fields or entity fields and let Spring handle it. For example:

public class SignupRequest {  
 @MobileKSA  
 private String mobileNumber;  
 @NationalIdKSA  
 private String nationalId;  
 // ... other fields ...  
}

Then in a controller:

@PostMapping("/signup")  
public Response registerMember(@Valid @RequestBody SignupRequest request) {  
 // ...  
}

Spring will automatically invoke the validators. If mobileNumber doesn’t match MobileKSA pattern, the request will be rejected with a 400 Bad Request and a validation error message. We likely integrate this with *shared-rest*’s exception handler to produce a nice **constraint violation response** (perhaps as part of the Problem+JSON). In fact, *shared-rest* might catch MethodArgumentNotValidException and format the violations. The violation messages would come from our annotations. For example, the response could be:

{  
 "status": 400,  
 "error": "VALIDATION\_FAILED",  
 "message": "Validation failed for one or more fields",  
 "violations": [  
 {"field": "mobileNumber", "error": "Invalid Saudi Arabia mobile number format"}  
 ],  
 "traceId": "abc123"  
}

Where "Invalid Saudi Arabia mobile number format" came from *shared-validation*’s message for MobileKSA. This uniform handling means front-end developers can rely on similar JSON structure for any validation errors across microservices.

Additionally, *shared-validation* might include some **utilities for data cleaning** – for example, a method to standardize phone numbers (remove spaces, add country code +966 if missing). They hint at *Normalizers*[[63]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=%40NotFuture). Possibly, we could use them in services whenever accepting input. Perhaps behind the scenes our @MobileKSA validator could call a normalizer that, say, if a number is 10 digits, prepend +966, or ensures it starts with 05 or +9665, etc. However, validators typically shouldn’t mutate the value; instead, the service might call normalizer before saving. This is more of a convention, not enforceable by Bean Validation. We might document that “You can use PhoneNumberNormalizer on inputs before persisting to unify format”.

**Integration with Multi-Tenancy and Domain Rules:** Some validations might even depend on tenant settings (like if each tenant can configure allowed formats or something). But given the examples, these are general for all tenants. If needed, one could inject a service into a ConstraintValidator (Spring allows that) to do advanced checks (like ensure a referenced ID exists in DB, etc.). But we likely avoid heavy logic in validators to keep them fast.

In summary, *shared-validation* is a smaller but useful piece that improves data integrity at the boundaries of our services. It encodes business knowledge (e.g., “Saudi phone numbers must be 9 digits and start with 5 when in local format”) in one spot. As new common validation needs arise (maybe validating coupon codes or membership IDs), we add them here and share across services.

## Shared REST Module

**Purpose:** The *shared-rest* module focuses on the **web layer concerns** of our microservices. It provides standardized handling of API errors, filters for incoming requests (like correlation and idempotency), and common configuration for API documentation. Essentially, it ensures that all RESTful APIs in LMS have a uniform behavior in how they handle exceptions, idempotency, and documentation. Key features:

* **Problem Details & Global Exception Handling:** *Shared-rest* likely implements a global REST exception handler (possibly extending the one in *shared-core*) that produces error responses in a structured format (e.g., Problem+JSON as per RFC 7807). It may define a ProblemException class and a RestExceptionHandler as a @ControllerAdvice[[64]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=). The ProblemException could be a runtime exception that carries an HTTP status and error code. The handler catches all exceptions (including Spring’s validation exceptions, security exceptions, etc.) and maps them to a ProblemResponse JSON. For example:
* If a service throws new ProblemException(domain="Member", code="NotFound", status=404), the handler will return {"type": "/problems/member/not-found", "title": "Member not found", "status": 404, "traceId": "...", "detail": "Member ID ABC not found"}. The *Problems* builder mentioned allows constructing these easily, e.g. Problems.of("member","not\_found", 404).detail("Member not found").
* If a generic exception occurs (null pointer, etc.), the handler likely wraps it in an InternalServerError problem with traceId.
* It also adds the current traceId and tenantId to every error response for debugging[[64]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=). This means clients get a traceId they can report to us, and we can find the log. The tenantId in error might help if the error is tenant-specific (though leaking tenantId to client is usually fine since client is that tenant).
* Validation errors can be aggregated in a violations list as described earlier.

By using this standardized error model, we avoid ad-hoc error responses (like different services writing different JSON for 404 or 500). It also integrates with our logs: the traceId in the response is the same as in logs, so support teams can easily correlate.

* **Request Filters (Correlation and Idempotency):** *Shared-rest* defines servlet filters (or Spring OncePerRequestFilters) that run for each HTTP request to handle cross-cutting concerns:
* **CorrelationIdFilter:** As noted, this generates or extracts a correlation ID for the request and puts it in MDC[[65]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=ProblemException%2C%20RestExceptionHandler%20). If the client sent an X-Correlation-ID, it uses that; if not, it generates a new UUID (or traceId). It then attaches it to the response header so the client knows it, and logs will contain it. This filter is crucial for distributed tracing correlation at the application level (complementing OTel).
* **IdempotencyFilter:** This filter enforces the usage of **Idempotency-Key** header on unsafe (typically POST/PUT) requests[[65]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=ProblemException%2C%20RestExceptionHandler%20). The idea is: for any state-changing REST call (like create/update operations), the client should send a unique Idempotency-Key: someUUID. The filter will check if this key has been seen before for the current tenant. If not, it allows the request to proceed and stores the key. If yes (meaning a duplicate request, possibly retry), it short-circuits: either returns the same response as last time (replaying from a cache) or simply ignores the duplicate. Our implementation likely stores a hash of the response for that key to return on duplicates[[36]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=Idempotency). The IdempotencyKeyStore (or IdempotencyRepository) from *shared-core* is used here[[66]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=idempotency%2F).
  + On first request: the filter intercepts, sees key K not in DB, so it marks it (perhaps inserts a row tenantId, key, first\_seen=now, status=IN\_PROGRESS) and lets the request through. The request is processed, the response is about to be sent. The filter (likely in an after-completion hook) then stores the response’s essential parts (HTTP status, maybe body hash) in the idempotency\_keys table and marks status COMPLETED.
  + On a retry: the filter sees key K already exists with a completed status, so it fetches the stored result (status code and maybe cached body) and immediately returns that to the client without hitting the controller again. This prevents duplicate side effects on the server (e.g., creating two orders instead of one).
  + If the previous request was still IN\_PROGRESS (e.g., if the client retries very quickly), the filter might wait or reject the concurrent request with 409 Conflict (depending on strategy, but likely we assume only one in-flight).
  + The store is likely with a TTL so that after a certain time (maybe 24 hours) those keys expire to avoid unbounded growth.
* The design choice to implement idempotency in a generic filter means every service automatically gets this capability for all relevant endpoints, rather than each implementing it. It addresses network issues where clients might retry on timeout but the server actually processed the request. Now they won’t create duplicates. Developers just need to ensure clients provide the header; we could enforce it by the filter sending an error if a safe call (GET) has an idempotency key (maybe we ignore it) or if a POST is missing the header, possibly we could warn or still allow it (depending on strictness – we might decide to enforce it strictly to ensure clients adopt it).
* *Example:* A client calls POST /members without Idempotency-Key. Possibly our filter could generate one (less ideal, as client won’t know to reuse it). More likely we either allow it (meaning not idempotent) or we respond with 400 Bad Request: Missing Idempotency-Key to enforce best practice. We might lean toward enforcement for external clients, but internal scenarios might not use it. Since it’s internal use by dev team, we can decide. For now, let’s assume it’s required for critical endpoints.
* **OpenAPI (Swagger) Auto Configuration:** Documenting APIs is made easier by *shared-rest*. It can integrate **Springdoc OpenAPI** (if we use that) to automatically configure global aspects like the JWT security scheme and common response codes[[67]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=CorrelationIdFilter%2C%20IdempotencyFilter). For example, it might add an *Authorize* button in Swagger UI by defining a bearer token scheme globally, so each service doesn’t need to do that config. It could also add common response descriptions (like 401, 403, 500) to all endpoints so we don’t manually annotate them everywhere. Additionally, it might group endpoints by tags or include the *module’s domain name in tags, etc., for consistency. Essentially, any repetitive Swagger configuration is handled once. If there are custom converters or OpenAPI custom schemas (for our Problem response, etc.),* shared-rest\* can define them so that the generated OpenAPI JSON includes those models.
* **Response Wrappers:** The *shared-rest* module likely uses the ResponseEnvelope<T> from core for all responses[[68]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=IdempotencyKey%20). It might have controllers return ResponseEntity<ResponseEnvelope<X>> for instance. To simplify, *shared-rest* could employ a controller advice that wraps responses automatically. However, often teams decide to only wrap where needed. If we do envelope everything, the Problems we send might also be enveloped unless excluded. This might be too deep, so maybe only normal responses are enveloped with traceId, etc., and error responses are separate.
* **CORS and Other Filters:** The module might also configure things like CORS (if needed for UIs calling the APIs) or compression or other web filters at a global level. Since not mentioned explicitly, we focus on the main two: correlation and idempotency.

**Design & Usage:** Once *shared-rest* is on the classpath, the service will have these filters registered (likely via Spring Boot auto-config). The service developer doesn’t need to manually add them. They should however be aware of their presence: for example, know that Idempotency-Key is expected and design clients accordingly. Also, when writing controllers, they can throw exceptions (like ProblemException) directly and trust the global handler to do the right thing, rather than catching and forming responses manually.

**Example of Global Error Handling:** A service might have code:

Member member = memberRepo.findById(id).orElseThrow(() ->   
 Problems.notFound("member", "not\_found").detail("No member with id " + id)  
);

Here Problems.notFound(domain, code) is a static method likely from *shared-rest* (or core) returning a ProblemException or similar. Throwing it triggers *shared-rest*’s handler which knows that “member/not\_found” corresponds to a 404 with a standardized error body. The developer doesn’t have to construct the HTTP response themselves, just indicate the error condition via exception. This improves readability and consistency.

**Idempotency Example:** Suppose a *shared-rest* IdempotencyFilter intercepts POST /orders with Idempotency-Key XYZ-123. It uses *shared-core*’s IdempotencyKeyStore (which could be a service that calls a JPA repository for the IdempotencyKey entity). On first call, store returns “not found” for XYZ-123, so filter will proceed and also save an entry that this key is in use. The controller executes and returns, say, 201 Created with body Order{id=1001}. The filter then updates the idempotency entry for XYZ-123 with status completed and perhaps stores a hash of the response body or the entire response in a cache. If the client retries the call (maybe they didn’t get the first response due to a network glitch), now the filter finds XYZ-123 exists. It can either: - Re-execute the controller but that might create duplicate order (so not good). - Directly return a response. It has the last response cached; it sets status 200 (or 201?) accordingly and returns the same body Order{id=1001}. The client gets the result without creating a new order. Alternatively, we might decide to prevent re-execution by returning a special code (202 Accepted with no body or 409 Conflict) to signal "already processed". But a better UX is to return the same success response, which some idempotency designs do. Our snippet in architecture doc mentions *persist idempotency\_keys with TTL* and *response\_hash*[[36]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=Idempotency). That implies we store something to compare responses. Possibly we only store a hash of the response to detect if a different response would occur (but ideally, a properly designed operation should always yield the same result for same input if executed again). So likely we do return the original response.

From a developer perspective, none of this is coded in their service – it’s entirely handled by *shared-rest*. They just need to document that their endpoint uses Idempotency-Key and let the filter do its job. If the filter is applied globally, maybe it has configuration to only apply to certain HTTP methods or paths (we might not want it on *all* posts – maybe only on those configured as idempotent operations). Possibly an annotation @Idempotent on a controller class or method could signal the filter to enforce idempotency on that route. The snippet in doc shows *IdempotencyFilter* listed as always included[[65]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=ProblemException%2C%20RestExceptionHandler%20), so maybe it triggers on all POST/PUT automatically.

* **OpenAPI Usage:** If a developer runs the service with *shared-rest*, they likely get an automatically available Swagger UI or at least an OpenAPI JSON at /v3/api-docs. It will already know about the bearer token and maybe common responses. This saves each service from writing a config like:

@OpenAPIDefinition(  
 security = @SecurityRequirement(name="bearerAuth"),  
 info = @Info(...),  
 components = @Components(securitySchemes = {  
 @SecurityScheme(name="bearerAuth", type=HTTP, scheme="bearer", bearerFormat="JWT")  
 })  
)

Because *shared-rest* might do that for them. If the library has a dependency on springdoc-openapi-ui, it might even serve the Swagger UI. This encourages every service to have interactive API docs ready with minimal effort.

**Conclusion:** *Shared-rest* aligns the behavior of our REST endpoints across microservices. Clients interfacing with multiple LMS services will see consistent error formats and can use the same strategies for retries (thanks to idempotency). It also plugs in seamlessly with the other modules: it uses *shared-core* for context and exceptions, *shared-security* for attaching security info to docs or checking roles (some error flows might involve security exceptions, which are also handled globally), *shared-observability* for logging the correlation ID filter actions, etc. The cohesive design means the top-layer concerns (HTTP specifics) are managed here, allowing service developers to focus on business logic.

## HTTP Request Handling Flow

To illustrate how all the above pieces come together for an incoming API call, **Figure 3** shows the flow of an HTTP request through the various shared library components (filters, context, etc.) before reaching the application’s business logic, and the flow back out with the response:

【31†source】 The sequence is as follows:

1. **Client Request:** A client (could be a frontend app or another service) calls a microservice endpoint. It includes an Authorization: Bearer <JWT> token and typically an X-Correlation-ID if it’s continuing a trace (or the client might omit it if it expects server to generate). If the request is a POST/PUT, it will include an Idempotency-Key header as recommended.
2. **Security Filter Chain:** When the request hits our service, Spring Security (configured by *shared-security*) is the first to act. It will validate the JWT – checking signature, expiry, and scopes. If invalid, it short-circuits with 401. If valid, it establishes the SecurityContext with the authenticated principal (user details) and authorities (roles/scopes). As part of this, our custom JwtAuthConverter runs, extracting the tenant\_id claim and user info and storing them. At this point, the user is authenticated and we know the tenant ID from the token.
3. **CorrelationIdFilter:** Next, *shared-rest*’s CorrelationIdFilter executes[[65]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=ProblemException%2C%20RestExceptionHandler%20). It looks for X-Correlation-ID. If present, it validates/uses it; if absent, generates a new unique ID (e.g., a UUID). It then stores this in MDC (for logging) and also in our RequestContext (since that holds correlationId as well). It sets an HTTP response header X-Correlation-ID with this value so the client can know or use it for further calls. This filter doesn’t block anything; it enriches context.
4. **Tenant Context Setup:** Around this time (could be done in the security filter or a dedicated step), *shared-security* ensures the **TenantContext** is set. This means putting the tenantId (from JWT) into RequestContext.tenantId[[25]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=Method%20security%20setup%20with%20Spring,Security). It could be done by a filter or as part of Security context establishment. Now our static RequestContext has tenant, user, correlation info all populated. Also, *shared-db* likely hooks in via an OpenEntityManagerInView interceptor or on the first DB call, enabling the Hibernate filter with that tenantId. Essentially, before any repository is used, the current session knows to restrict by tenant = X.
5. **IdempotencyFilter:** If the request method is POST/PUT, *shared-rest*’s IdempotencyFilter runs[[65]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=ProblemException%2C%20RestExceptionHandler%20). It checks for the Idempotency-Key header.
6. If missing and the filter is configured to require it, it may immediately return a 400 error (the GlobalExceptionHandler would formulate a Problem JSON saying “Missing Idempotency-Key”). Assuming the header is present, it proceeds.
7. It queries the IdempotencyKeyStore (probably using *shared-db* JPA) for an entry with this key and current tenant.
   * If found with status COMPLETED, it means this is a retry of a previously successful call. The filter will then fetch the saved response data. It will short-circuit the chain and return that data to the client directly, skipping calling the controller again. This would come out as a normal HTTP response (200/201) with the cached body. The filter might also set a header like Idempotent-Replay: true to indicate this was a replayed response (not necessary, but could be useful).
   * If found with status IN\_PROGRESS (rare, maybe if a concurrent duplicate request), the filter might either wait a tiny bit or return a 409 Conflict indicating “Request already in progress”. For simplicity, let’s say we disallow concurrent duplicates and treat it as conflict.
   * If not found, it will create an entry (status IN\_PROGRESS) and allow the request to continue to the controller.
8. In our flow diagram, we assume it’s a new request, so it goes through.
9. **Controller & Service Logic:** Now the request reaches the application’s controller (REST endpoint). Spring will have deserialized JSON to a DTO, validated it (via *shared-validation* annotations if any) before entering the controller. If validation failed, an exception is thrown which is caught by GlobalExceptionHandler later (we’ll cover that on the way out). If validation passed, inside the controller any @PreAuthorize or @RequiresScope annotations are checked (thanks to *shared-security*). Assuming the user is authorized, the controller calls service methods to perform business logic. The service might use repositories, which due to *shared-db* are already filtering by tenant. So e.g., orderRepository.save(order) automatically sets tenantId, etc. The service might also publish events via *shared-messaging* (which means it writes to Outbox within the transaction). All of this is business as usual; the shared library pieces are mostly passive here except ensuring things like tenant filter on queries. The RequestContext is available throughout – if the service needs current user or tenant, it pulls from there (populated in step 4).

Assume everything goes well and the service prepares a response object.

1. **Returning Response & Post-Processing:** The controller returns a response (or perhaps just the object and Spring wraps it in ResponseEntity). Before sending it out:
2. The *RestExceptionHandler* isn’t involved for success responses, but *shared-rest* might wrap the response in a ResponseEnvelope (depending on our design, it might be manual or automatic). Let’s say we opted to wrap. So a success body {"orderId":1001, "status":"CREATED"} might be wrapped as {"data":{...order...}, "traceId":"<id>", "tenantId":"TENANT-001"}. This could be done by a ResponseBodyAdvice that adds those fields. If we chose not to wrap (some prefer not to for normal responses), then nothing happens here.
3. The IdempotencyFilter, if this was a idempotent request, will intercept on the way out. It now has the response (and status). It will store in the database the outcome: e.g., for key XYZ-123 mark as COMPLETED, and possibly store a hash of response body (to detect if a future execution would differ). In some implementations, it might store the full response or a reference to it (but storing full payloads could be heavy; a hash and status code might suffice, since on retry we might just re-run the operation or perhaps we cached it in memory for quick return). In our approach, likely we store at least status code and maybe location header or something minimal, and use that to respond on duplicates. The filter then allows the response to go to client.
4. The CorrelationIdFilter might add the correlationId header to the response here (if not already done).
5. The Security context filter might clean up (SecurityContext is cleared after request).
6. The RequestContext filter definitely clears the thread-local at the end so no data leaks to next request on same thread.
7. **Client Receives Response:** The client gets the response with potentially the correlationId header. If it was a duplicate call that got short-circuited, they still get a valid response (maybe without knowing it was from cache, unless we indicated). They also have the same Idempotency-Key echo (some systems echo it back or not – not necessary).
8. **Logging and Tracing:** Throughout this flow, our *shared-observability* instrumentation has been at work:
9. A trace was started at the beginning (OTel created a span for the HTTP request named like HTTP GET /orders).
10. The traceId is propagated to logs via MDC. So any log statements in controller/service automatically have traceId, tenant, user if we put that in MDC as well.
11. When the response is about to be sent, the trace/span is ended, and OTel exports it (with info like HTTP status, duration).
12. Our logging of request/response info might be done at filter level. Possibly *shared-rest* logs an access log: e.g. INFO [correlationId] 201 POST /orders (tenant TENANT-001 user=U123) 45ms. If we do, it uses the context we set.
13. The outbox event (if any) that was recorded will be picked up asynchronously but that’s separate from the request thread.
14. **Exception Scenario:** If any exception occurred in controller or service (like thrown ProblemException or runtime), the GlobalExceptionHandler in *shared-rest* would catch it:
    * If it’s a ProblemException (our own), it builds the Problem JSON. If it’s validation errors, it builds that response with violations. If it’s an unexpected exception, it likely logs the error (with stack trace and traceId) and builds a generic 500 response.
    * It sends that as response. The filters (correlation, etc.) still run to add headers and clear context. IdempotencyFilter would, if an error occurred after we reserved a key, mark the key as maybe FAILED. Depending on our design, we might treat error responses as non-idempotent (i.e., allow a retry to actually retry the operation). Possibly if status != 2xx, we clear the idempotency entry so that a retry is allowed. The stored status could be “ERROR” with maybe the response hash, but then on retry we might not want to instantly return the same error – perhaps the issue was transient and retry could succeed. For simplicity, likely we do not cache errors as idempotent results (with the exception of conflict errors maybe).
    * All that logic would be documented for developers: e.g., "If a request fails, the Idempotency-Key is considered unused (or expires quickly) so that a retried request will attempt again".

**Figure 3: HTTP request handling flow through shared library components.** When a request comes in, it passes through security authentication, correlation ID assignment, and idempotency check before reaching the application’s controller. Upon responding, the idempotency filter records the outcome and the correlation ID is included in the response. Throughout, the tenant context is enforced and logs/traces are recorded for observability.

*Figure 3: End-to-end flow of an HTTP request in an LMS microservice using the shared library. The request first goes through security validation (JWT parsing and tenant extraction), then the CorrelationIdFilter and IdempotencyFilter from* shared-rest *apply*[*[69]*](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=)*. The request context (tenant, user, trace) is set*[*[25]*](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=Method%20security%20setup%20with%20Spring,Security) *and passed into business logic. Responses are handled by the global exception handler for errors or returned normally, with the filters adding the correlation ID header and storing idempotency records. All database operations automatically used the tenant filter, and any events were saved to the outbox for later publication. The result is a consistent, safe handling of each request across all microservices.*

This comprehensive request pipeline ensures that every request is authenticated, authorized, tracked, and isolated to the correct tenant with minimal effort by the service developer. It also means cross-cutting rules (like idempotency and logging) are uniformly applied.

## Multi-Tenancy Enforcement Flow

Multi-tenancy is a fundamental concern throughout the LMS architecture. This section visualizes how **tenant isolation** is enforced at different layers when a request is processed, ensuring data from one tenant never leaks to another. **Figure 4** depicts the end-to-end tenant enforcement:

1. **JWT and Security Context:** When a user or service makes a request, the JWT contains a claim for the tenant\_id (e.g., “TENANT-001”). The *shared-security* module extracts this and stores it as the current tenant context[[70]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=JWT%20parser%20to%20extract%20tenant_id%2C,roles%2C%20scopes). This happens right at the gateway of the service (during authentication). The application never trusts any tenant ID coming as a URL param or body field without cross-checking it against this context.
2. **Tenant in RequestContext:** The tenant ID from JWT is put into RequestContext.tenantId (a thread-local storage) by a security filter or an interceptor[[25]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=Method%20security%20setup%20with%20Spring,Security). From this point on, any part of the code can retrieve the current tenant via RequestContext. This is used by *shared-db* and possibly by other checks. For example, if a controller has an endpoint /tenants/{tenantId}/accounts, the code can verify that the {tenantId} path variable matches RequestContext.getTenantId(), rejecting the call if not (so a user from tenant A cannot ask for data under tenant B’s ID). We often design the APIs to not even require the tenantId as a param (since we know it from auth), but some internal endpoints might have it. Regardless, the server will ensure they match.
3. **Hibernate Tenant Filter:** When the service accesses the database, the *shared-db* module’s TenantFilterInterceptor ensures that a Hibernate filter parameter tenantId = 'TENANT-001' is set on the session. So any JPA repository query implicitly adds AND tenant\_id = 'TENANT-001'. If the code tries to access an entity that belongs to another tenant, it simply won’t be returned by queries. Similarly, on insert, either the entity’s tenantId field was explicitly set to 'TENANT-001' (perhaps by the service code or an event listener) or the insert is prevented if it's null. Most likely, BaseEntity’s @PrePersist method (or the interceptor) sets the tenantId before insert if it’s missing. Thus, all new records get stamped with the correct tenant without developers having to remember it each time.
4. **Row-Level Security at DB (optional):** We rely on the application filter, but for defense-in-depth, we could also enable PostgreSQL’s **Row-Level Security (RLS)** on tables with tenant\_id[[29]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=Security%3A%20MFA%20for%20admins%3B%20RBAC%2FABAC%3B,trails%20immutable%3B%20data%20encryption%20KMSmanaged). In a scenario where someone accidentally bypasses Hibernate (like running a raw SQL via JdbcTemplate), if RLS is active (with policy tenant\_id = current\_setting('app.current\_tenant') or similar), the DB itself would restrict results. Implementing that requires setting app.current\_tenant on each connection (which we could do via a PG SET command when obtaining a connection with tenant context). This is an advanced measure; we might not have it now but it’s considered in the architecture[[29]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=Security%3A%20MFA%20for%20admins%3B%20RBAC%2FABAC%3B,trails%20immutable%3B%20data%20encryption%20KMSmanaged). Our current approach is simpler: rely on the app layer filter and static analysis/testing to catch any bypass.
5. **Service Logic Enforcement:** On the service layer, besides the automatic DB filtering, we also ensure any cross-tenant operations are prohibited. For example, if admin tokens (with a special role) can switch context, we explicitly code that. Otherwise, normal service methods never specify tenant outside of their own. If a code path tries to do something like repository.findByTenantId(someOtherId), that’s a red flag – we generally avoid writing repository methods that take tenantId because it should always be the current tenant. If needed (like in a multi-tenant admin service that manages tenants), such methods exist but are only callable by a super-admin role and possibly require manual switching off of the filter (with great care). Additionally, *shared-security* might supply a check that if a user from tenant X tries to access a URL containing tenant Y, an AccessDenied is thrown at the web layer.
6. **Isolation in Caches and In-Memory Data:** Not shown in diagram but worth noting: if we use any caching, we key the cache by tenant as well. The shared library could provide a CacheKeyGenerator that prefixes keys with tenantId to avoid collisions. Similarly, any static in-memory data structures should be tenant-scoped or global read-only. This is more of a guideline for developers.
7. **Response and Logging:** The response going out contains only tenant-specific data. For example, if tenant A user asked for their members, the JSON will obviously only have their members (because that’s all that was fetched). We also include the tenantId in log contexts and maybe in audit logs to clearly tag everything. If a bug or misconfiguration caused data from another tenant to appear, our test and audit processes should catch it (for instance, our **tenancy-leak tests** simulate requests from two tenants and ensure no leakage).

**Figure 4: Multi-tenancy enforcement across request processing.** When a request is made with a JWT containing a tenant claim, the system carries that tenant identity through every layer. The security filter and RequestContext establish the tenant context[[25]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=Method%20security%20setup%20with%20Spring,Security). The Hibernate filter in *shared-db* then restricts all data access to that tenant. Any attempt to access or modify data from another tenant is either filtered out, ignored, or explicitly blocked by the application. Even on producing a response, only tenant-specific information is included. This guarantees isolation: each tenant’s data is invisible to others by design.

*Figure 4: Tenant Isolation End-to-End. The user’s JWT token identifies their tenant (“ACME Corp”), which* shared-security *uses to set the tenant context. As the request flows through, the* shared-db *layer ensures every database query and insert uses the tenant context (e.g., adding WHERE tenant\_id = 'ACME'). The service code naturally operates only on data for ACME. The response contains data only from ACME’s partition. If the user attempted to access another tenant’s resource, they would be denied or simply get no data. Multi-tenancy is enforced at multiple points to provide robust isolation.*

By structuring the system this way, even if a developer forgets a check at one layer, another layer will catch it. For example, if they forgot to check path tenantId vs token tenantId in a URL, the DB filter would still prevent any cross data from being returned. Conversely, if someone directly wrote a SQL without tenant\_id clause, a code review or the RLS (if enabled) would stop it. This layered approach (JWT claims + application filter + database filter) aligns with a strong security posture for multi-tenancy.

## Architectural Guidelines and Constraints

To maintain the integrity of our architecture, we have established several **architectural rules** and use automated enforcement where possible:

* **No Cyclic Dependencies:** As discussed, modules (and even packages within modules) must not form cycles. This keeps the codebase modular and comprehensible. For example, *shared-core* can’t depend on *shared-db* while *shared-db* depends on *shared-core*, as that would entangle them and complicate initialization and testing. We use ArchUnit tests to assert that such cycles do not exist. This applies not only to top-level modules but also to layers within services (though within a single service, we usually follow layered architecture too). ArchUnit rules declare something like: “Classes in package ..shared.core.. should not access classes in ..shared.db..” (except maybe some allowed exceptions). If a violation is introduced, the build fails, forcing a fix (e.g., move a utility from db to core if needed).
* **Layered Dependency Direction:** Dependencies should flow **downwards** from higher-level to lower-level modules, not the reverse. Allowed dependency flow is: shared-core <- (depended on by) others; shared-db and shared-security may depend on core; shared-messaging, shared-observability, shared-rest may depend on core and perhaps on db/security as needed (but e.g., messaging doesn’t depend on rest, rest doesn’t depend on messaging, etc.). We explicitly document allowed pairs. Another ArchUnit test might enforce a whitelist: e.g., classes in *shared-rest* can use *shared-core* and *shared-security* classes, but not vice versa. This prevents sneaky cross-linking. A concrete rule: “No class in com.ejada.lms.shared.rest.. should be accessed by classes in com.ejada.lms.shared.core..” (to ensure core is pure base). We also avoid any module depending on *shared-test* (test is only for test scope).
* **Encapsulation of Concerns:** Each module should encapsulate its concern and not leak it to others inappropriately. For instance, database-related code stays in *shared-db*. If another module needs to do something with the database, it should go through an interface or callback provided by *shared-db*, not implement its own DB logic. Similarly, security details (like role names, etc.) are defined in *shared-security* and used elsewhere, but, say, *shared-core* wouldn’t directly reference Spring Security classes (keeping core free of that). This separation is reviewed in PRs.
* **Consistent Naming and Conventions:** Architectural consistency also means naming conventions for packages, classes, and topics/queues etc. The shared library sets many conventions (topic naming, as we saw, or naming of Spring beans and roles). We have a guideline document that developers follow (e.g., event classes should be named XEvent and placed in events package; ID generators in ids package, etc.). While not strictly enforced by tools, these conventions improve predictability.
* **Stable Interfaces and Backwards Compatibility:** Since multiple microservices depend on the shared library, we aim to keep its interfaces stable or backward-compatible when possible. Architectural rule: avoid breaking changes in public methods of shared modules without a major version bump. For example, if *shared-core* provides IdempotencyKeyStore.save(key) method, we wouldn’t remove or change its signature in a minor update; instead we’d add new methods or mark it deprecated and only remove in next major release. Compatibility is tested by having at least one consumer service (or a sample service) integrated in CI that ensures it still works with the new library version.
* **No Business Logic in Shared Modules:** The shared library is foundation, not business domain. We enforce that shared modules should not contain business-specific logic (like no code deciding how points are calculated or voucher rules). They may contain generic domain-agnostic helpers (even if domain-flavored like KSA ID validation, it’s not core business logic). This keeps the shared lib broadly usable and avoids coupling it to specific service implementations. ArchUnit can’t directly check “no business logic”, but code reviews do. We also avoid referencing specific microservice packages from the shared lib to keep it general.
* **Clear Module API vs Implementation:** Each module should expose a clear API (public classes meant to be used by services) and internal details should be package-private or in impl packages. For example, *shared-messaging* may have EventPublisher interface that services call, but the actual OutboxRelay thread or KafkaTemplate config is internal. Developers using the library should rely on the API and not hack around it. To enforce, we use Java’s access modifiers and possibly jigsaw modules if applicable (not sure if using JPMS). At least, internal packages are documented as such.
* **Coding Standards and Quality:** We apply consistent coding standards (naming, formatting, etc.) across the shared library. This is ensured via lint tools or a style guide. It’s not strictly architectural, but it keeps the code maintainable. We integrate tools like Checkstyle or SpotBugs in the build. SpotBugs might catch code issues (null pointer risks, etc.) and we treat those seriously especially in shared code since a bug here propagates everywhere.
* **Testing of Architectural Assumptions:** We have specialized tests like **tenancy-leak tests** to validate our multi-tenancy enforcement. For example, a test might populate data for two different tenants in a repository (by simulating context switch) and ensure that querying under one tenant never returns another’s data. Another test might ensure that if developer accidentally writes a repository method not filtering by tenant (like a custom @Query without where clause), that test would catch it. Those tests often run via *shared-test* or within each service, but the patterns are defined centrally.
* **ArchUnit Enforcement:** We have ArchUnit test classes dedicated to shared-lms. Example rules we enforce:
* **“Services cannot bypass shared library layers”:** e.g., a service should not call internal JPA EntityManager directly for a multi-tenant table – it should use the repository which has filter. Hard to enforce generally via ArchUnit, but if we find patterns, we add rules. For shared lib itself:
* **“No Spring @Component in shared-core (except config)”:** keeping core mostly free of Spring-managed beans (to avoid early initialization issues).
* **“Controllers should only exist in services, not in shared library”:** ensure no accidental controllers in shared-rest – it should only provide advices and filters, not actual endpoints.
* etc.

These architectural rules are maintained as part of our **Quality Gates** in CI. A violation in architecture (like a forbidden dependency or cycle) breaks the build just like a failing unit test would.

By adhering to these constraints, we maintain a clean architecture that scales: new features or modules can be added without entangling with existing ones, and developers can reason about the system in layers. This discipline also makes onboarding easier: a new developer can be told “module A handles X, and it only depends on module B, never on C” – a clear mental model.

## Security Practices and Policies

Security is woven into every part of the LMS shared foundation. Here we outline the key security practices implemented via the shared library and overall policies:

* **JWT-based Authentication:** All microservices use **JWT (JSON Web Tokens)** for authentication, configured by *shared-security*. We treat the Authorization JWT as the single source of truth for user identity and tenant membership. Each service is a resource server that trusts tokens issued by our Auth service (e.g., Keycloak or a custom IAM). We ensure tokens are signed (RS256) and *shared-security* validates the signature and expiration on each request. The shared config also typically enforces that the token’s audience or issuer matches expected values (to prevent tokens from another context being used). We require **short-lived tokens** (e.g., 15 minutes) and use refresh tokens on the client side to balance security and usability. This limits the window a stolen token is valid.
* **Role-Based Access Control (RBAC):** The JWT contains roles (and possibly fine-grained scopes). *Shared-security* parses these into Spring Security authorities[[21]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=SecurityAutoConfiguration). We define roles like TENANT\_ADMIN, TENANT\_USER, PLATFORM\_ADMIN, etc., and scopes for specific permissions (like members:read, members:write). We use **annotations (@RequiresRole, @RequiresScope)** on controllers or methods to guard them[[71]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=JwtAuthConverter%20%28maps%20roles%20%E2%86%92%20ROLE_,scopes%20%E2%86%92%20authorities%2C%20reads%20tenant_id). If a user lacks the required authority, they get HTTP 403. The shared library ensures that these annotations are consistently enforced. It also provides a central definition of what each role means (via RoleConstants and documentation). Our policy is principle of least privilege: services should check for the narrowest scope needed. For instance, to edit points, require points:write scope, not just any authenticated user.
* **Tenant Isolation:** As detailed in Multi-Tenancy Enforcement, each user/token is tied to one tenant. We do not allow a token to access data from a different tenant. The *shared-security* module’s multi-tenant filter plus the DB layer filter enforce this strongly. Additionally, we design our APIs such that the client doesn’t usually even specify tenant – it’s derived from token to reduce risk of mishandling. For internal admin tools that manage multiple tenants, we use special roles (like PLATFORM\_ADMIN which *shared-security* could recognize and possibly bypass tenant filter, but in those cases the code must explicitly assume responsibility to specify target tenant). Those use-cases are limited and carefully audited.
* **Row-Level Security (RLS) at DB:** While currently our primary enforcement is at the application layer, we have an eye on using Postgres RLS as a backup. The goal is that even if someone gains direct DB access or if an app query slipped through without filter, the database won’t return unauthorized data. We have the schema ready for it (every table has tenant\_id). Implementing RLS might involve adding a policy like CREATE POLICY tenant\_isolation ON table\_X FOR SELECT USING (tenant\_id = current\_setting('lms.tenant\_id')). Then *shared-db* when acquiring a connection sets SET lms.tenant\_id = 'TENANT-001'. This is doable but adds complexity, so we’ve opted for the simpler approach first. Still, our **security policy** is to treat cross-tenant data access as a *critical severity bug*. We regularly test with scenarios to ensure no such bug in code, and if we find any, we consider enabling RLS as a safety net in that service.
* **Audit Logging and Monitoring:** Every sensitive action (login, role change, config change, etc.) is audit-logged. The shared library’s AuditLogger (in *shared-observability*) is used to record events like “User X (tenant Y) granted role Z to user W” or “Tenant admin updated reward settings”. These audit logs are stored in an append-only fashion (either a DB table or sent to a secure log index). This ensures traceability – if data is modified or accessed, we have a record of who did it and when. In particular, any use of high-privilege roles is flagged (e.g., Platform Super Admin actions across tenants are all logged). This practice is crucial for compliance (e.g., ISO27001, and Saudi PDPL which likely requires tracking access to personal data)[[60]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=Compliance%20%26%20Security).
* **Data Encryption:** At rest, we rely on database encryption (Transparent Data Encryption on the DB server or disk encryption). In transit, all service-to-service communication goes over TLS (either because they’re all behind HTTPS in cluster or if using plain HTTP internally, the network is isolated – but generally we do TLS for microservice calls as well, or at least for external calls). For sensitive data like credentials or personal IDs, we could additionally use application-layer encryption if needed, but often we just mark them and handle carefully (e.g., not logging PII as mentioned).
* **JTI (Token Revocation):** We consider token revocation important for scenarios like compromised credentials. *Shared-security* provides the mechanism to honor jti claim. Our policy is that if an admin explicitly revokes a token (for example via an Auth service endpoint), any service should reject that token going forward. Implementation: a distributed cache (like Redis) of revoked JTIs. *Shared-security* could consult this cache in its auth filter. We keep the JTI blacklist entries until the token’s natural expiry. This way, if a user’s account is disabled or they log out, we can prevent further use of their token. It’s a bit against the JWT stateless nature, but required for security. In practice, we rarely revoke often (to keep cache small), usually only when needed. Also, using short-lived tokens limits the necessity – an unrevoked token will expire soon anyway. But for immediate cut-off (say user is terminated), revocation is used.
* **Rate Limiting and Throttling:** While not explicitly a shared lib feature, we often implement rate limiting per tenant or user on sensitive endpoints to mitigate abuse. For instance, *shared-rest* could integrate with a library or provide a filter for rate limiting (like a simple in-memory token bucket per IP or user). If not in the library, our API gateway or sidecar might handle it. But the architecture doc mentions quotas per tenant[[72]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=Multitenancy%3A%20Tenant%20isolation%20,pertenant%20configs%20%26%20rate%20limits) which implies we might have some enforcement of calls per minute, etc. Possibly, *shared-observability* metrics are used to track rates and an external system triggers alerts or blocks. For now, no explicit code in the shared lib for rate limiting has been described, but we might incorporate something like resilience4j’s RateLimiter or bucket4j in the future, configured globally for certain endpoints.
* **Secrets Management:** The shared library (especially *shared-security*) likely gets public keys for JWT verification from a well-known JWKS endpoint or a certificate. We ensure that keys and any secrets (like DB passwords, third-party API keys) are not hard-coded but come from environment or vault and are injected via config at runtime. The BOM centralizes versions of crypto libs to ensure we don’t have vulnerabilities by using outdated versions.
* **SQL Injection and Input Validation:** On top of using JPA (which handles SQL parameters safely), we use *shared-validation* to validate inputs (which prevents malicious inputs from going deeper). For any dynamic queries, we use parameter binding rather than string concatenation, avoiding SQL injection. Also, by restricting DB user privileges (each service DB user can only its schema), even if injection happened, damage is limited to that schema.
* **Cross-Site Scripting (XSS) and Output Encoding:** For our responses (which are JSON), we ensure any user-provided data is properly encoded. This is typically handled by JSON libraries (which will escape special chars in strings). We also might set the appropriate content-type headers. The *shared-rest* module’s global config probably sets Content-Type: application/json; charset=UTF-8 for all responses. If we served any HTML (not typical in these services), we’d use frameworks to sanitize it.
* **Transport Security:** All services communicate either via HTTPS or encrypted channels (Kafka is configured with TLS perhaps, since that’s often done for production, though not strictly necessary if within a secure network, but a good practice if multi-tenant data flows through Kafka to secure it as well). *Shared-messaging* allows configuring security (brokers, maybe SASL if needed). For external calls (HTTP to third-party), we always use TLS and possibly set up mutual TLS if required by partner systems.
* **Dependencies and Patches:** The BOM ensures we control library versions. We keep an eye on security advisories for dependencies (e.g., Spring, Jackson, etc.). We update the BOM regularly to include patched versions (especially for critical vulns). This shared approach means a single BOM update can patch dozens of services at once, rather than each service individually.
* **Penetration Testing and Review:** Although beyond code, as part of practice, we subject the system to pen tests. The shared library’s code gets special scrutiny because a flaw here (like an insecure crypto usage or a bypass in filter) would affect all services. For example, we double-check that the idempotency filter and global exception handling don’t inadvertently reveal sensitive info (like stack traces might be suppressed to avoid info leakage).

In summary, our security approach is multi-layered: **secure the perimeter** (JWT auth, TLS), **secure each request** (role checks, tenant checks), **secure the data** (filters, RLS, encryption), and **monitor everything** (audit logs, alerts on suspicious activity). The shared library provides the tools and defaults so that every microservice adheres to these without a lot of custom code. A developer creating a new endpoint automatically benefits from these security measures (they have to explicitly opt-out or do something very custom to go around them, which would be caught in code review).

## Observability Strategy

We have built observability into the shared foundation so that we can achieve **360° visibility** into the system’s behavior. The strategy encompasses logs, metrics, and traces, with a focus on correlation and data privacy:

* **Centralized Logging (ELK Stack):** All services log in JSON format (via *shared-observability* config) to stdout or a file, which is then aggregated by our logging system (e.g., Filebeat/Logstash sending to Elasticsearch). This allows real-time searching of logs across all services. We tag each log with serviceName and environment (via common tags in the log JSON) so we can filter easily. Because every log also has tenantId (for application logs) and traceId, we can filter by tenant during incidents or follow a transaction across microservices. Our logging level guidelines: debug logs in non-prod for deep dives, info for important business events (like “Member created”), warning for recoverable issues, error for failures. The global exception handler logs errors with stack traces and includes the correlationId so we can tie the error log to the error response that the client got. We avoid logging high-volume or sensitive data (no full request payloads for example in production logs, unless needed and then scrubbed). Instead, we log summary info (like “Created 50 points for member X”).
* **PII Scrubbing:** Given that personal data flows through the system (member names, contact info, etc.), our logs should not store raw PII to reduce compliance risk. *Shared-observability*’s log scrubber ensures that if a developer accidentally logs something like a full Member object which includes email or phone, those fields get masked. We maintain a list of sensitive keys (like “name”, “email”, “phone”, “address”, etc.) that the scrubber looks for in JSON logs and replaces values with “\*\*\*” or a hashed version[[59]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=Logback%20JSON%20encoder%20%2B%20PII,scrubber%3B%20CorrelationIdFilter%3B%20AuditLogger%20facade). This might occasionally over-mask (maybe “name” of something not person), but we prefer caution. We also instruct developers to mark logs appropriately—e.g., for audit logs, some PII might be allowed if it’s necessary (like recording which email was sent an invite, but even then we could store a partial masked email). The scrubbing ensures compliance with PDPL – logs accessible to engineers won’t easily reveal personal data, and any debugging requiring full PII would go through proper approvals to look at DB with audit trail.
* **Distributed Tracing (OpenTelemetry):** We instrument all inter-service calls with tracing. This means if Service A calls Service B (via REST or sends a Kafka event), the trace context goes along. In practice:
* For REST calls, we propagate traceparent and X-Correlation-ID headers. *Shared-rest* automatically sets traceparent on outgoing calls if using a RestTemplate that’s instrumented. We use OTel’s context propagation libraries to ensure it picks up from current span and attaches to HTTP headers.
* For Kafka, *shared-messaging* adds trace context in message headers (could be the W3C Trace Context or a custom correlationId). Our consumer picks it up and resumes the trace. So a flow that goes HTTP -> Service A -> Kafka -> Service B -> DB will show up as one trace in Jaeger with multiple spans (HTTP span in A, processing span in A, Kafka send span, Kafka receive span in B, processing span in B, DB spans, etc.). This helps enormously in diagnosing latency issues or pinpointing which service in a chain caused an error.
* We sample traces intelligently: perhaps we sample a percentage of requests if volume is high, or always sample error requests. The sampling config is centralized (maybe via env var that *shared-observability* respects).
* We ensure that **trace IDs do not carry customer data**, they are random UUIDs. And span metadata should not have PII either (span names are just operation names, attributes might include IDs but not personal info). If we do add any data to spans (like a memberId), we consider if that is allowable since trace data might live in external systems.
* **Metrics and KPIs:** Using Micrometer, each service exports metrics to a Prometheus (or other backend). Key metrics we track:
* **Infrastructure metrics:** CPU, memory, GC (these might come via Kubernetes and not directly from app, but some JVM metrics are exposed via Micrometer’s JvmMeterBinder).
* **Application metrics:**
  + Request rates and durations by endpoint (Micrometer auto-timed Spring MVC controllers can give http\_server\_requests\_seconds\_count and \_sum to derive latency).
  + Error rate per endpoint (via the same metrics with status tag).
  + DB query execution times if we instrument DataSource or use an SQL comment with trace (less out-of-the-box, but OTel can instrument JDBC to produce spans).
  + Kafka consumer lag (Spring Kafka provides metrics on lag which Micrometer can capture).
  + Custom domain metrics: for example, *shared-observability* might not produce them, but each service can register metrics like points.issued.count per tenant. If we do per tenant counts, we must handle cardinality – possibly we aggregate or limit to top tenants. Alternatively, we have a separate usage tracking service (the "Usage Tracking" microservice mentioned in arch doc collects such counts via events)[[73]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=Usage%20Tracking)[[74]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=Count%20system). So metrics focus on system health, and for domain stats we rely on that usage service or queries.

Our observability strategy is to use metrics for **alerting** (e.g., high error rate triggers an alert, high latency triggers SRE investigation) and use logs/traces for **diagnosis**. We set up dashboards: - e.g., an API performance dashboard showing p95 latency per service and endpoint, error count trends, throughput. - a Kafka dashboard showing consumer lags and outbox queue sizes (maybe we expose outbox table pending count as a metric). - a system health dashboard with instance counts, GC times, etc.

We also define **SLOs** (Service Level Objectives) such as “95% of requests < 300ms”[[28]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=Availability%3A%2099.9,for%20portal%20UI). We track these via metrics. If they breach, we have alerts.

* **Correlation of Logs with Metrics:** We often want to jump from an alert (metrics) to relevant logs or traces. Because we propagate traceId/correlationId everywhere, we can, for instance, configure an alert to include a traceId if one is associated (in some systems you can link them). Or at least, when investigating, we can go to Jaeger, find traces during that time with errors, and then correlate to logs via traceId. This cross-linking is part of our incident response playbook.
* **PII in Observability Data:** We treat observability data with nearly the same care as production data. That means the systems storing logs/traces/metrics are secured (accessible only to authorized team members). We scrub PII but still consider that traces might indirectly contain some (like tenant IDs we consider not super sensitive, but still internal, and maybe member IDs could be indirectly personal if guessable). So our monitoring systems are within our secure network, not exposed publicly. We also have retention policies – e.g., logs stored 30 days, traces maybe only 7 days at full detail, metrics aggregated after 13 months, etc., to balance utility vs. risk.
* **Using Observability for Testing and Tuning:** We also leverage these tools in staging environments to do performance tests and see if traces reveal slow queries, etc. The uniform instrumentation means we can simulate production loads and gather the same metrics to identify bottlenecks. For example, if a certain DB query is slow, OTel trace will show a long span for that query. Or if an external call is causing delays, we’ll see it clearly.
* **Analytics vs Observability:** We distinguish between metrics for ops vs. data for analytics (like BI). The latter (points earned, etc.) is handled by domain events and the usage tracking service, not by Micrometer metrics (to avoid heavy tagging). Observability metrics focus on system performance and correctness. Domain events serve business analytics (with their own pipelines to a data warehouse, per arch doc sections on Reporting and Usage tracking).

Our observability strategy is ultimately about achieving **quick detection and diagnosis** of issues in a complex microservice environment. By building it into the shared platform, we ensure every new service is born with these capabilities – there’s no excuse for a “black box” service. This dramatically reduces MTTR (Mean Time to Repair) when incidents happen, since engineers can swiftly find what went wrong and where.

## Versioning and Release Management

We maintain a disciplined approach to versioning the LMS Shared Library and managing its releases to ensure stability across all microservices:

* **Semantic Versioning (SemVer):** The shared library follows SemVer (Major.Minor.Patch). **Major** versions (e.g., 2.x.x) include breaking changes or big new features that might require service code modifications. **Minor** versions (e.g., 1.1.0, 1.2.0) add new features or improvements in a backward-compatible manner. **Patch** versions (e.g., 1.0.1, 1.0.2) include bug fixes or small enhancements, fully backward-compatible. For example, moving from 1.0.0 to 1.1.0 might introduce a new validator or a new method in EventPublisher, but existing methods/classes remain unchanged. Jumping to 2.0.0 would indicate something like dropping support for an old approach (say, removing a deprecated class or requiring a new Java version). We communicate the level of changes clearly through version numbers.
* **Release Frequency:** We aim to batch changes into planned releases rather than continuous trickle (unless a critical fix is needed). A typical cycle might be: development of new features in a feature branch, then when enough or at a schedule (e.g., end of sprint), we bump the minor version and release. Patch releases for urgent bug fixes or security vulnerabilities can happen as needed (even out-of-band). We avoid frequent major releases to minimize churn – ideally major releases are infrequent (maybe yearly or less), whereas minors could be every few sprints with enhancements.
* **Changelog:** Each release is documented in a **CHANGELOG.md** file, which is maintained in the repository. The changelog entries are human-readable and categorized by version. For example:

## [1.1.0] - 2025-09-01  
### Added  
- Introduced @EmailKSA validator in shared-validation.  
- EventEnvelope now includes `sourceService` field for debugging.  
- New ArchUnit tests to enforce naming conventions.  
  
### Changed  
- Upgraded Spring Boot to 3.1.0 (from 3.0.5) in shared-bom.  
- IdempotencyFilter now returns 409 instead of 400 when key is reused concurrently.  
  
### Fixed  
- Resolved issue where CorrelationIdFilter didn't add header on error responses.  
- Fixed NPE in TenantFilterInterceptor when no tenant context (like in batch jobs).

This format (inspired by *Keep a Changelog*) allows developers to quickly see what’s new, what might affect them, and any action needed. In particular, any breaking changes are called out explicitly (in a "Breaking Changes" section or by marking them).

* **Backward Compatibility Testing:** Before releasing a new version, we test it against a sample of microservices to ensure compatibility. We maintain a **compatibility test suite** that might involve using a dummy service that includes the new library and verifying basic operations (like startup, filter registrations, etc.). If a change might break something subtle (like changed a filter order), we double-check it doesn’t. For major releases, we may do a trial upgrade on a non-critical service in staging environment to catch issues. We also rely on strong unit/integration tests within the shared lib itself to catch regressions.
* **Deprecation Policy:** When we want to change or remove a feature, we deprecate first. For instance, if we had an old CorrelationIdFilter in shared-rest and we replaced it with a new implementation in shared-observability, we’d mark the old one as @Deprecated and mention in changelog "Deprecated CorrelationIdFilter, will be removed in 2.0.0; use new CorrelationContextFilter instead." This gives service teams time to adjust if needed. Usually, since we often provide things transparently, deprecation might not require their action unless they directly used a class that is going away.
* **Release Process:** We use our CI/CD pipeline to release artifacts to an internal Maven repository (nexus/artifactory). The process is: once changes are merged into main and tested, we bump the version in pom (e.g., 1.0.0 -> 1.0.1) and tag the repository. The CI job for release will package all modules and deploy the jars (and a BOM pom) to the repository. We then update the BOM version in all microservices to the new one. We often use the BOM so that microservices just reference lms-shared-bom:1.0.1 and it pulls all modules at that version. Upgrading them is just changing one BOM dependency.
* **Coordinating with Microservice Releases:** Since the library update might need to propagate to many services, we coordinate the timing. For non-breaking minor/patch releases, we usually expect services to upgrade fairly soon (to get fixes and improvements). Possibly we schedule a platform-wide dependency update sprint or automate raising PRs to each service’s repo updating the BOM version. For major versions, we might allow a longer co-existence where some services still use 1.x while others move to 2.x, assuming runtime compatibility (which usually exists since they're separate processes, but any shared data formats have to be compatible—e.g., EventEnvelope changes might need to be backward compatible until everyone updates). We try to avoid having to support two major branches for long. Ideally, we upgrade all services within a maintenance window or similar.
* **Testing and Quality Gates for Release:** Before finalizing a release, our pipeline runs all tests (unit, integration) for the library itself. We also run a **integration test suite** possibly deploying a sample service with the new lib to ensure the filters and context actually work in a running server. Additionally, quality gates like code coverage (we aim e.g. >90% coverage for shared lib code, since its reliability is paramount) and static analysis (no critical Sonar issues) must be passed. Only then do we allow release.
* **Version Compatibility and Breaking Changes:** If a breaking change is absolutely needed (say, we found a design flaw that must be corrected in a non-backward compatible way), we plan it for a major version. We document what needs to change in service code. For example, if we change how idempotency keys are handled and require services to do something differently, we’ll provide a migration guide. Perhaps rarely, we might include a temporary shim to ease migration (like support both old and new way in one version). But since this is internal, we can coordinate changes fairly directly with the team.
* **Example:** Our current version is 1.0.0. Let’s say we want to add a new @Currency validator and also upgrade Kafka client version. That would be a minor release 1.1.0 – adding features, no expected break. If instead we decide to rename a package or remove something deprecated, that’s major 2.0.0 territory. We’ll avoid doing that until necessary, and accumulate multiple breaking changes to do in one major release to minimize frequency.
* **Change Communication:** Even though we have changelogs, we also communicate in internal channels (Slack or email or team meetings) that "Shared Library v1.1.0 is released – it includes X, Y, Z; please update your services when possible, especially to get fix for issue Q." If a security fix is in it, we mark it urgent. The whole dev team is aware of how to consume the new version (just update BOM and test their service).
* **Regression Plan:** If a new library version causes an unforeseen problem in a service (e.g., performance issue or a bug that slipped), we can quickly rollback that service to previous version (since it's just using older BOM, which still exists in repo). Meanwhile, we fix the issue in a patch release. Our CI can maintain multiple versions if needed, but usually we’d patch on top of latest.
* **Compatibility Testing for Events and DB:** We also ensure that changes to things like EventEnvelope or Outbox do not break interoperability. For example, if we decided to add a new field to EventEnvelope, older consumers that ignore unknown JSON fields will be fine (we verify that’s the case). If we changed the event schema significantly, we might need a phased rollout (producer adds both old and new field for a while). Similarly, if *shared-db* introduced a new Flyway base migration (like adding a new table for some feature), each service picking the new lib will apply it – which is fine if independent. But if it changed an existing table’s definition (say idempotency key length), we ensure it's backward compatible with existing usage or coordinate changes such that services on old version aren’t broken by new DB schema. Typically, each service’s DB is separate, so that's less of an issue cross-service; it’s more within the service's context that the lib and DB schema are in sync.

The careful versioning and release process ensures that the shared library remains an asset, not a liability – it accelerates development without causing integration headaches. As a result, teams can confidently use the latest shared features knowing that they have been tested and will behave consistently across the landscape.

## CI/CD and Testing

Our Continuous Integration/Continuous Deployment pipeline is set up to enforce quality gates and thoroughly test changes to both the shared library and its usage in microservices. Key aspects:

* **Automated Testing in Layers:** We maintain several layers of tests for the shared library:
* **Unit Tests:** Every utility class, filter, etc., in the shared lib has unit tests covering expected behavior and edge cases. For example, tests for IdempotencyFilter simulate various scenarios (new key, duplicate key, error handling) to ensure logic is correct. We use JUnit (and maybe AssertJ, Mockito, etc.) to isolate and test functionality. Unit tests run on every commit/push in CI.
* **Integration Tests:** In the context of the shared library, integration tests spin up components together. For instance, using Spring’s test context, we start a minimal web server with our filters and verify that a dummy request goes through security, correlation, idempotency and yields expected outcome. Or start an in-memory H2 DB to test that TenantFilterInterceptor actually filters data. *Shared-test* module aids this by providing containers for Postgres, Kafka, etc., to test things like outbox pattern end-to-end. We have an integration test where we simulate a service writing to outbox and our OutboxRelay class picks it and a TestConsumer receives it, verifying no duplicates. These integration tests run in CI as well (likely as part of the shared library build, marked differently).
* **Contract Tests:** If we consider the shared lib providing a “contract” to microservices, we could have tests that ensure the library works in a typical service environment. For example, a contract test could be: “When library is included in a service, these beans are present, these endpoints respond as expected (like Actuator health is up, etc.).” This might be partly covered by integration tests. Another contract test aspect is checking that common interactions (like produce event in Service A, consume in Service B with idempotency) hold, which we do as integration scenario tests.
* **Quality Gates – Coverage and Static Analysis:** We treat a high code coverage as a must for shared code. CI fails if coverage drops below e.g. 90%. We use tools like Jacoco to measure and SonarQube or similar to enforce. Static analysis via SpotBugs/PMD or Sonar rules is also enabled. For example, if SpotBugs finds a potential NPE or SQL injection risk or unsecured crypto, we address it immediately. Sonar’s security hotspots (like usage of MD5, or use of System.out printing secrets etc.) are monitored. Because the shared lib is critical, we keep its code quality bar higher than perhaps individual service code (where something might slip but only affect one service; here it affects all).
* **Architecture Tests:** As mentioned, ArchUnit tests run in CI to enforce architecture rules (no cycles, layering, naming). These run with the unit tests and break the build if violated. We keep these tests updated as architecture evolves (for instance, if we add a new module, we add rules for it accordingly).
* **CI Pipeline Workflow:** On each commit/PR:
* Compile and run all tests (unit + integration).
* Run static analysis/linting.
* Build artifacts (but not release them unless on main branch with release process).
* Possibly build a sample microservice (we might have an example service in the repo or separate) to ensure the library works in a real app context.

Only if all pass can code be merged. We also require code review approvals for any changes in shared library – often from senior devs or architects – given the high impact.

* **Test Pyramid in Microservices:** While the shared lib has its tests, each microservice also has tests which indirectly test the shared lib in their context:
* **Unit tests in services** might mock some shared lib behaviors, or just run through (like testing a controller will inherently run through the global exception advice).
* **Service Integration tests:** Many services use *shared-test* which sets up real Postgres, etc., so they are actually exercising the lib’s config (like Flyway migrations from shared-db run, etc.). If a new change in lib broke something, a service’s integration test could catch it. For example, if we changed how the tenant filter param is named and a test in service expected old name, it fails. Thus, our CI for microservices, when they update the library, catches issues. We sometimes proactively run a few representative service test suites against a new library snapshot to ensure nothing obvious fails.
* **Contract Testing for Events (Consumer/Producer):** If microservice A produces an event that B consumes, we ensure the contract (schema) is consistent. Some teams use Pact or Avro schema evolution tests. In our case, because the schema for events is likely centralized (we register event schemas in Schema Registry and share the EventEnvelope definition), we rely on backward compatibility of those schemas. When we change an event schema, we test that old consumer code can still parse it or explicitly version the event type and handle accordingly. The *shared-messaging* integration test might simulate a scenario: Service A using library v1 sends event, Service B using library v1.1 receives – to ensure, for instance, that adding a new field to EventEnvelope doesn’t break an older consumer library (which would just ignore the unknown field if using JSON).
* **Tenancy-Leak Tests:** Specifically, we have a suite of tests to ensure no tenant data leaks:
* For each repository in a sample domain, test that creating data in one tenant and querying in another yields nothing.
* Test that after a request, the tenant context is cleaned up (e.g., run two requests with different tenant IDs on the same threadpool thread to ensure context doesn’t persist).
* Possibly a test that tries to disable the filter (if someone manually calls session.disableFilter("tenant") as a malicious attempt) to see if our design prevents it (like maybe we mark TenantFilterConfig as not removable, though not sure if possible – anyway, normal devs wouldn’t do that).

These tests might be part of *shared-db* integration tests.

* **Performance Tests:** We may not have heavy performance tests for the library in isolation, but we do test that adding the library doesn’t slow down the service unacceptably. For example, measure overhead of filters – correlation and idempotency filters are very lightweight (just a few lookups/inserts), which is fine. Outbox adds some overhead in DB transactions – we test that publishing, say, 100 events adds minimal latency. If any performance issue is found, we optimize in the library (so all services benefit). Sometimes we use JMH (a microbenchmark tool) for critical utility (like ID generation, though ULID and Snowflake are fast enough).
* **CI in Microservices for Shared Updates:** When a new library version is released, microservices run their full CI with that update. We treat their test failures as potential issues either in service or in library. If something common fails across multiple services after a lib update, it likely points to a hidden bug or an overlooked breaking change in the library. We then may need to fix the library and release a patch. To streamline this, we might have a meta-test environment where we run a subset of service tests nightly with the latest library snapshot (continuous integration testing). This way, we catch integration issues early, not only when teams manually upgrade.
* **CD (Continuous Deployment):** For the shared lib, CD means publishing to the Maven repo (not deploying a running app). For microservices, once they pass tests with new lib, they can be deployed to staging/prod as per normal process. Because multiple services might update around the same time for a new lib, we coordinate to avoid a "big bang" unless needed. Usually, each service can be deployed independently with the new lib because the changes are internal – as long as backward compatibility holds for inter-service comms (which we ensure via event/contracts). We might stagger deployments to not change everything at once, easing monitoring (if an issue arises, it’s easier if only a couple of services were updated).
* **Quality Gates Recap:** Summarizing the key quality gates we enforce:
* **Test Pass Rate:** All tests must pass.
* **Code Coverage:** e.g., must remain >= 90%. If new code is added, tests should accompany it.
* **No Critical Sonar Issues:** e.g., no memory leaks, no use of risky functions.
* **Architecture compliance:** ArchUnit rules all green.
* **Documentation updated:** Not a “gate” in CI but in PR review we ensure if a new feature is added, the developer updated this guide or at least the changelog and possibly any README in the repo. Documentation is part of quality – we keep code and docs in sync to avoid confusion.
* **Example CI run for a PR:** Developer adds a new @IBAN validator in shared-validation. They also tweak IdempotencyFilter to handle a new header. They write tests for these. On push, CI runs:
  + Build modules -> compile success.
  + Run ~300 unit tests -> all pass (maybe new tests count increased).
  + Start a Postgres container and run integration tests for shared-db and idempotency -> passes (the idempotency test shows new header scenario passes).
  + Run ArchUnit -> passes (no new dependency issues).
  + Run SpotBugs -> maybe passes or flags something if any (like "you didn't close a resource" – they'd fix and push again if so).
  + Compute coverage -> say 92%, passes threshold.
  + Mark build success. Then a reviewer checks code, approves. On merge to main, a release pipeline might be triggered if we decided this is a release commit (or we manually bump version and tag, then CI publishes version).

The robust CI/CD and testing approach ensures that when the shared library is updated, we have **high confidence** it will not break existing services or, if it introduces a new feature, that feature works as intended in the contexts it will be used. It also enforces architectural consistency as the system grows, keeping technical debt in check.

## Reference

### Key Components and Responsibilities

Below is a quick reference of important components provided by the LMS Shared Library and their responsibilities:

* **BaseEntity (shared-core):** Abstract JPA entity that provides id (UUID or long ID), tenantId (String/UUID), and audit timestamps (createdAt, updatedAt). All persistent domain entities in services should extend this to inherit multi-tenancy and auditing fields. Responsibility: Ensure every row knows its tenant and creation/update time.
* **AuditableEntity (shared-core):** Extends BaseEntity to add createdBy and updatedBy fields (user identifiers). Typically populated via @CreatedBy and @LastModifiedBy annotations, integrated with Spring Security’s user context. Responsibility: Track which user or service account created/modified data for audit purposes.
* **SoftDeleteEntity (shared-core):** Extends BaseEntity to add a deleted flag and possibly a deletedAt. Marked with a Hibernate @Filter or @Where(clause="deleted=false") to exclude soft-deleted records by default. Responsibility: Allow logical deletion of records while keeping them for potential restore or audit.
* **RequestContext (shared-core):** A thread-local context holder for the current request’s key info – tenantId, userId, correlationId, and maybe roles or permissions. Populated at request start (by filters) and cleared at end. Accessible from anywhere in service code via static getters. Responsibility: Provide an easy way to access context without passing it through method signatures.
* **Ulid & SnowflakeId (shared-core):** Utilities/classes for generating unique IDs. Ulid produces a 26-char string sortable by time; SnowflakeId produces a 64-bit number. Often configured with node IDs to avoid collisions. Responsibility: Offer globally unique identifiers for entities and events, ensuring no collisions across services or instances.
* **IdempotencyKeyStore & IdempotencyKey (shared-core):** The store (could be an interface or Spring bean) for persisting idempotency keys. Likely backed by a JPA repository of IdempotencyKey entity (with composite key of tenant + key)[[66]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=idempotency%2F). It provides methods like saveInProgress(key), markCompleted(key, responseHash), find(key). Responsibility: Remember client request keys to prevent processing duplicates.
* **CorrelationIdFilter (shared-rest / shared-observability):** Servlet filter that manages the Correlation-ID. Generates a new one if none provided, adds it to MDC and response header[[59]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=Logback%20JSON%20encoder%20%2B%20PII,scrubber%3B%20CorrelationIdFilter%3B%20AuditLogger%20facade)[[65]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=ProblemException%2C%20RestExceptionHandler%20). This might internally use a CorrelationID class or simply a UUID. Responsibility: Tag each request with a unique ID for tracking across boundaries.
* **TenantFilterInterceptor (shared-db):** Hibernate interceptor or filter enabler that sets the tenant filter parameter on each session/transaction. Possibly implemented as a Spring bean that hooks into entity manager creation. Responsibility: Constrain all database operations to the current tenant’s data.
* **GlobalExceptionHandler / RestExceptionHandler (shared-rest):** A @ControllerAdvice that globally handles exceptions and formulates HTTP error responses. It knows how to handle ProblemException (custom) vs Validation errors vs generic exceptions. Responsibility: Ensure clients receive meaningful, consistent error responses and hide internal stack traces.
* **ProblemException & Problems (shared-rest):** ProblemException is likely a runtime exception carrying problem details (type, title, status) for an error. Problems is a helper class to create these (with methods like .of(domain, code, status)). Responsibility: Simplify throwing API errors and attaching context like detail messages or retry-after headers.
* **IdempotencyFilter (shared-rest):** Filter that intercepts idempotent requests, checks/records the Idempotency-Key using IdempotencyKeyStore[[65]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=ProblemException%2C%20RestExceptionHandler%20). Responsibility: Guarantee that duplicate requests with the same key within a certain period result in a single operation execution.
* **EventEnvelope<T> (shared-messaging):** Wrapper for event payloads with standard metadata (tenantId, correlationId, traceId, event type or schema ref, possibly a messageId)[[43]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=KafkaProperties%20%26%20KafkaAutoConfig%20,with%20transactions). Used both in producers and consumers. It may be a simple POJO or a generic record if using Avro. Responsibility: Carry context with every event to enable traceability and multi-tenant segregation in event processing.
* **OutboxMessage entity (shared-messaging):** JPA entity representing an event waiting to be sent to Kafka[[46]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=Outbox%20pattern). Contains fields: id (UUID), topic, key, payload (JSON/bytes), status (NEW/SENT), createdAt. Responsibility: Temporarily store events as part of local transactions for reliable eventual delivery.
* **OutboxRelay (shared-messaging):** Component that continuously polls or waits for outbox messages and publishes them to Kafka in a transactional manner[[47]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=JPA%20OutboxMessage,createdAt%2C%20status). Could be implemented with Spring’s @Scheduled or a persistent listener on DB events. Responsibility: Bridge the database and Kafka, ensuring atomicity (using Kafka transactions).
* **IdempotentKafkaListener (shared-messaging):** Base class or aspect for Kafka consumers that provides idempotency. It wraps message handling with logic to check the ConsumerOffset store[[51]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=Idempotent%20consumer). Possibly an abstract class where you implement handleEvent(EventEnvelope e) and it handles offset checking. Responsibility: Avoid duplicate processing of events in consumers by tracking processed message IDs.
* **ConsumerOffset / ProcessedMessage (shared-messaging):** Entity/table recording processed event IDs or offsets[[52]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=Idempotent%20consumer). Likely composite key of consumerGroup + messageId, with timestamp. Responsibility: Serve as memory of consumed messages to filter out repeats.
* **SecurityAutoConfiguration (shared-security):** Configuration class that sets up JWT decoding, security context, method security annotations support (via @EnableGlobalMethodSecurity). It may create beans like JwtDecoder (pointing to JWKS) and configure HttpSecurity to require authentication. Responsibility: Provide a plug-and-play security setup aligned with our JWT issuer and claims.
* **JwtAuthConverter (shared-security):** Converter that takes a Jwt and returns a Spring Authentication (like a UsernamePasswordAuthenticationToken with authorities)[[21]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=SecurityAutoConfiguration). It reads claims like roles and scp (scopes) from JWT, maps them to GrantedAuthoritys (ROLE\_ prefix for roles, or custom for scopes), and also possibly stores the tenantId and userId in the details or in RequestContext. Responsibility: Adapt JWT info into Spring Security context and by extension our RequestContext.
* **RequiresScope / RequiresRole (shared-security):** Custom meta-annotations that simplify security expression syntax[[22]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=JWT%20parser%20to%20extract%20tenant_id%2C,roles%2C%20scopes)[[71]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=JwtAuthConverter%20%28maps%20roles%20%E2%86%92%20ROLE_,scopes%20%E2%86%92%20authorities%2C%20reads%20tenant_id). For instance, @RequiresScope("members.write") might internally be @PreAuthorize("hasAuthority('SCOPE\_members.write')"). They improve readability of controller code. Responsibility: Declaratively secure endpoints with minimal boilerplate, leveraging consistent role/scope names.
* **RoleConstants & ScopeConstants (shared-security):** Classes defining constant strings for common roles (e.g., TENANT\_ADMIN = "TENANT\_ADMIN") and scopes (e.g., MEMBERS\_READ = "members.read"). Responsibility: Provide a single source of truth for permission identifiers to avoid typos and mismatches between services.
* **AuditLogger (shared-observability):** Utility to log security-sensitive events (like data exports, config changes) possibly to a separate appender or channel[[59]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=Logback%20JSON%20encoder%20%2B%20PII,scrubber%3B%20CorrelationIdFilter%3B%20AuditLogger%20facade). Might have methods like audit(String action, Map details) which under the hood uses a logger with a special marker. Responsibility: Ensure important actions are recorded in an immutable and searchable way for compliance.
* **CorrelationID and Logging MDC setup (shared-observability):** The combination of the filter and some logging configuration that ensures whenever RequestContext is set, those values (traceId, correlationId, tenant) are put into SLF4J MDC. Possibly accomplished via the RequestContextFilter or Spring’s MDCAdapter. Responsibility: Tie the context information to all log statements on the handling thread, enabling log aggregation by those keys.
* **Micrometer MeterRegistry Customizer (shared-observability):** A bean that adds common tags (service name, environment, maybe region) to every metric[[58]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=OpenTelemetry%20exporter%20autoconfig). Also could configure histogram buckets for timers or any specific metric settings. Responsibility: Standardize metrics across services so that, for example, all of them have service=XYZ tag making it easy to filter per service in Prometheus queries.
* **PIIScrubber (shared-observability):** Likely a custom Logback turbo filter or a layout wrapper that processes each log event to mask sensitive info. It might scan message strings and MDC for patterns (like email regex or digits that look like ID). Responsibility: Prevent sensitive personal or security data from being stored in logs in plain text.
* **Shared BOM (shared-bom):** The Bill of Materials pom that defines versions for all dependencies (Spring Boot, Spring Cloud, Kafka client, OTel, etc.). Services import this BOM so they all use the exact same versions, avoiding classpath conflicts. Responsibility: Lock dependency versions across microservices and simplify dependency management for teams.

This reference list is not exhaustive but covers the most critical components. Each of these plays a role in fulfilling the design principles of the platform: **consistency, multi-tenancy, security, reliability, and observability**.

### Example Usage Snippets

To further clarify how developers use the shared library in practice, here are some short code snippets demonstrating key scenarios:

* **Using RequestContext in Service Code:**

@Service   
public class PointsService {  
 public void awardPoints(String memberId, int points) {  
 String tenant = RequestContext.getTenantId(); // Current tenant from context  
 String user = RequestContext.getUserId(); // Who is performing this action  
 log.info("Awarding {} points to member {} in tenant {}", points, memberId, tenant);  
 // ... business logic ...  
 PointsLedger ledger = new PointsLedger(memberId, points);  
 // tenantId on ledger will be auto-set via BaseEntity or manually:  
 // ledger.setTenantId(tenant); (if not automatic)  
 pointsRepository.save(ledger);  
 }  
}

Here, the developer did not need to worry about how tenant or user got there – they trust that *shared-security* and *shared-db* have done their job, and they just use the context.

* **Custom Validator Usage (shared-validation):**

public class SignupRequest {  
 @MobileKSA  
 private String mobile;  
 @Email // using standard annotation for email  
 private String email;  
 @NationalIdKSA  
 private String nationalId;  
 // getters and setters...  
}  
  
// In a controller:  
@PostMapping("/members")  
public ResponseEntity<Void> registerMember(@Valid @RequestBody SignupRequest request) {  
 memberService.register(request);  
 return ResponseEntity.status(HttpStatus.CREATED).build();  
}

If request.mobile is "123456", the MobileKSAValidator will detect it's invalid (not matching KSA format) and Spring will not even enter registerMember – it will throw MethodArgumentNotValidException. The global exception handler catches that and returns a 400 with details like {"field":"mobile","error":"Invalid mobile number format"}. The developer didn't write any of that handling logic – it’s provided by *shared-rest* and *shared-validation*.

* **Publishing an Event with Outbox (shared-messaging):**

@Transactional  
public void createOrder(Order order) {  
 orderRepository.save(order);  
 // Publish domain event for order created  
 OrderCreatedEvent payload = new OrderCreatedEvent(order.getId(), order.getMemberId(), order.getAmount());  
 EventEnvelope<OrderCreatedEvent> event = EventEnvelope.create(payload);   
 eventPublisher.publish("prod.loyalty.events", event);  
 // The EventEnvelope will include tenantId from RequestContext and a new messageId  
 // The publish() method actually writes to OutboxMessage table instead of sending to Kafka directly.  
}

The eventPublisher is a bean likely provided by *shared-messaging*. After the transaction commits, the OutboxRelay will detect the new outbox entry and send it out.

On the consuming side (in another service):

@KafkaListener(topics = "prod.loyalty.events", groupId = "notification-service")  
public void handleLoyaltyEvents(EventEnvelope<?> envelope) {  
 // The listener method can take EventEnvelope as argument because shared-messaging configured MessageConverter  
 // Determine event type and process accordingly  
 if (envelope.getPayload() instanceof OrderCreatedEvent) {  
 OrderCreatedEvent event = (OrderCreatedEvent) envelope.getPayload();  
 // Now we can send a notification email for order creation  
 notificationService.sendOrderConfirmation(event.getMemberId(), event.getOrderId());  
 }  
 // The IdempotentKafkaListener base class (if extended) would automatically handle marking this message as processed.  
}

Because we presumably extended something or configured the container factory to use our idempotent error handler, if this handleLoyaltyEvents is called again for same message, it will quickly exit (not calling notificationService twice).

* **Declarative Security on Controllers:**

@RestController  
@RequestMapping("/admin")  
@RequiresRole(RoleConstants.TENANT\_ADMIN) // Only tenant admins can access any endpoint here  
public class TenantAdminController {  
  
 @GetMapping("/report")  
 @RequiresScope(ScopeConstants.REPORT\_READ) // Must also have report reading scope  
 public Report getTenantReport() { ... }  
  
 @PostMapping("/users")  
 @RequiresScope(ScopeConstants.USER\_WRITE)  
 public ResponseEntity<?> createUser(@Valid @RequestBody UserDto user) { ... }  
}

Thanks to *shared-security*, these annotations will be enforced. If a non-admin calls, they get 403 without any controller code executing. This simplifies securing dozens of endpoints.

* **Global Exception Handling (Problems utility):**

@GetMapping("/members/{id}")  
public MemberDto getMember(@PathVariable String id) {  
 Member member = memberRepo.findById(id)  
 .orElseThrow(() -> Problems.of("member", "not\_found", HttpStatus.NOT\_FOUND)  
 .detail("Member not found: " + id));  
 return memberMapper.toDto(member);  
}

This uses Problems.of(domain, code, status) from *shared-rest* to throw a standardized exception. The global handler will produce a 404 response:

{  
 "type": "/problems/member/not-found",  
 "title": "Member not found",  
 "status": 404,  
 "detail": "Member not found: 12345",  
 "traceId": "abcd-1234...",  
 "tenantId": "TENANT-001"  
}

The developer didn't have to assemble this JSON or set the status code; the framework did it.

* **Audit Logging example:**

if (userRoleChanged) {  
 AuditLogger.log("role\_changed", Map.of(  
 "adminId", RequestContext.getUserId(),  
 "targetUserId", targetUserId,  
 "newRole", newRole  
 ));  
}

This would produce a log (to a special audit log appender or index):

{"event":"role\_changed","adminId":"U111","targetUserId":"U222","newRole":"TENANT\_ADMIN",  
 "timestamp":"2025-08-20T10:15:30Z","tenantId":"TENANT-001"}

It’s separate from normal logs and perhaps immutable (append-only). During a security review, one can query all role\_changed events easily.

These snippets illustrate common patterns. The shared library’s goal is that writing such code is intuitive and requires writing **less code** than without the library, because the heavy lifting (security, error handling, etc.) is abstracted away.

### Glossary of Terms

* **LMS:** In this context, likely "Loyalty Management System" (an assumption based on domain terms like points, vouchers, etc. in the architecture document). It’s the product platform we are building, composed of many microservices and the shared library.
* **Multi-Tenancy:** The ability to host data for multiple client organizations (tenants) in the same application, with strict isolation between them. In our system, each API request and data record is associated with a tenant. Multi-tenancy enforcement ensures one tenant cannot access or affect another tenant’s data[[29]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=Security%3A%20MFA%20for%20admins%3B%20RBAC%2FABAC%3B,trails%20immutable%3B%20data%20encryption%20KMSmanaged).
* **Tenant:** A client organization or customer of the platform. In our system, each tenant has an ID and their own set of data (members, rewards, etc.). E.g., "Acme Corp" could be a tenant. Tenant context is carried via tenantId in tokens and data.
* **Tenant Context / TenantId:** The identifier of the current tenant, carried through the system (in JWT claims, RequestContext, database filters). It’s central to how multi-tenancy is implemented in code.
* **JWT (JSON Web Token):** A compact, URL-safe means of representing claims to be transferred between two parties. We use JWTs for authentication; each JWT has claims like user, tenant, roles, expiry. They are cryptographically signed to prevent tampering.
* **JTI (JWT ID):** A unique identifier for a JWT. We use it for token revocation; by storing the JTI of compromised tokens, we can reject them even if they haven’t expired.
* **RBAC (Role-Based Access Control):** Authorization mechanism using user roles. Each user is assigned one or more roles (like ADMIN, USER) which determine what actions they can perform.
* **Scope:** In OAuth2/OIDC context, a finer-grained permission string included in tokens (like "members:read"). We use scopes to grant specific privileges beyond broad roles. They appear in tokens and are checked in code.
* **Correlation ID:** An ID used to correlate logs and traces of a single request or workflow across multiple services. Often a GUID generated per request (or reused from an external call). Helps in tracking a transaction end-to-end.
* **Trace ID / Span ID:** Part of distributed tracing. A trace ID is like a correlation ID for the entire operation, and each service call is a span with its own ID. We propagate Trace IDs via headers (traceparent) for OpenTelemetry. Typically, correlationId and traceId might be the same or correlationId might map to traceId – conceptually similar, but traceId is more standardized format.
* **Outbox Pattern:** A pattern for ensuring reliable event publishing. Instead of sending events directly to the message broker within a database transaction (risking partial failures), events are written to an outbox table as part of the transaction, and a separate process publishes them to the broker[[49]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=enabled%20via%20an%20interceptor%20that,reads%20JWT%20tenant_id). Guarantees that DB change and event emit happen together or not at all.
* **Idempotency Key:** A unique key provided by the client to identify a particular request (usually a create/update). Used so that if the client repeats the request (due to timeout or retry), the server can recognize it and not perform the action twice[[36]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=Idempotency). Ensures that operations are not accidentally executed multiple times.
* **Idempotent (Operation):** An operation that can be applied multiple times without changing the result beyond the initial application. For example, adding points to a user is not inherently idempotent (doing twice doubles the effect), but using an idempotency key, we make the request effectively idempotent from the client perspective.
* **MDC (Mapped Diagnostic Context):** A logging utility that stores contextual information (like tenantId, userId) in a per-thread context such that any log entries from that thread include those details. We use MDC via SLF4J to automatically enrich logs.
* **Micrometer:** A metrics facade library for Java applications. It allows us to instrument code and emit metrics in a vendor-neutral way. Under the hood, we plug in a backend (Prometheus, JMX, etc.). We use Micrometer to collect metrics like request timings, etc., unified across microservices.
* **OpenTelemetry (OTel):** An observability framework for cloud-native software, providing standardized tracing, metrics, and logging APIs/SDKs. We specifically use it for distributed tracing (and possibly metrics). It replaces older systems like OpenTracing + OpenCensus. OTel defines how trace context is propagated and how spans are recorded.
* **ArchUnit:** A testing library for checking architecture rules (like package dependencies, layering constraints) in Java code. We use ArchUnit tests to enforce no circular dependencies, etc., as part of our build.
* **Testcontainers:** A Java library that provides throwaway instances of common services (like PostgreSQL, Kafka, Redis) in Docker containers for integration testing. *Shared-test* uses Testcontainers to allow each microservice to easily spin up e.g. a fresh Postgres DB for tests, ensuring tests are realistic (using actual DB engine and our migrations). We mention Postgres 16, Kafka 7.x, etc., are used in tests[[75]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=shared).
* **Contract Test:** A test to ensure that two systems (like a producer and consumer of a message, or an API provider and client) agree on a contract (data schema, behavior). E.g., if Service A sends an event, a contract test might use a stub consumer to verify the event format is as expected. This reduces integration issues when things change.
* **CI/CD (Continuous Integration/Continuous Deployment):** Practices of frequently integrating code changes with automated builds and tests (CI), and deploying to production through automated pipelines (CD). We have CI for the shared lib (running tests, quality checks) and then CD releasing the library artifact for use. For microservices, CD means deploying new container images after passing tests.
* **Quality Gate:** Criteria that must be met for code to be considered of acceptable quality. For us, that includes test coverage threshold, no critical static analysis issues, architecture compliance, etc. If any gate fails, the build is considered failed.
* **Coverage (Test Coverage):** The percentage of code lines or branches executed by tests. High coverage implies many code paths are tested. We require a high coverage for the shared lib to ensure reliability.
* **PDPL:** Personal Data Protection Law (specifically Saudi Arabia's PDPL given context)[[60]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=Compliance%20%26%20Security). We mention compliance with PDPL/ISO, which is why we mask PII and log audit events. So PDPL is an important legal framework guiding some of our security and logging decisions.
* **PII:** Personally Identifiable Information. Any data that could identify a person (names, emails, phone, national ID, etc.). We must protect PII by minimizing its use, securing it, and not exposing it inadvertently (hence scrubbing in logs and requiring auth for access, etc.).
* **SLA/SLO:** Service Level Agreement/Objective. SLA is a commitment (often to customers) for availability or performance. SLO is an internal target (like 99.9% uptime, or <300ms latency for 95th percentile)[[76]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=Goals%20%26%20NonFunctional%20Requirements). Our observability and design (like performance budgets for filters) help us meet SLOs which support the SLA.
* **BOM (Bill of Materials):** A Maven concept - a pom file that specifies versions for dependencies so that including the BOM in a project allows you to omit version tags for those deps (and ensures consistency). Our *shared-bom* is used by all microservices to pin dependency versions uniformly.
* **HikariCP:** A popular JDBC connection pool for managing DB connections efficiently[[27]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=shared). *Shared-db* configures HikariCP for Postgres. It's known for good performance.
* **Flyway:** A database migration tool that applies incremental SQL or Java migrations to evolve the schema. We include Flyway in *shared-db* to handle baseline and possibly common migrations (like idempotency and outbox tables)[[32]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=JSONB%20support%20via%20Hibernate%20types). Each service can also have its own migrations for its domain tables.
* **Kafka:** Apache Kafka, a distributed event streaming platform. We use it as our message broker for events between microservices. *Shared-messaging* wraps Kafka producer/consumer config for easier use. Concepts: producers send events to topics, consumers read from topics in consumer groups, at-least-once delivery semantics by default (hence needing idempotency). Kafka topics are partitioned; we likely key by tenant to keep events somewhat partitioned by tenant (could help with ordered processing per tenant).
* **Dead Letter Queue/Topic (DLQ/DLT):** A Kafka topic where messages are sent if they could not be processed by a consumer after retries. Not explicitly mentioned but part of our error handling strategy perhaps. The library may configure error handlers to send to a DLT (like \*.DLT).
* **OpenAPI/Swagger:** Tools for API documentation. *Shared-rest* integrates springdoc-openapi to generate interactive docs for our REST APIs with minimal config[[67]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=CorrelationIdFilter%2C%20IdempotencyFilter). Swagger UI is often used by developers or external integrators to explore and test endpoints.
* **Integration Test:** A test that verifies the interaction between multiple components (as opposed to unit tests which isolate a single class). E.g., starting the whole Spring context and hitting an endpoint is an integration test. We do a lot of these for verifying the shared lib in real scenarios.
* **Tenancy-Leak Test:** Our term for tests designed to ensure that data from one tenant never appears in operations of another. It’s a vital test for multi-tenancy correctness. For example, create an entity for tenant A and ensure that a query under tenant B yields zero results – that’s a tenancy leak test.
* **Resilience4j:** A library for resilience patterns (circuit breakers, rate limiters, retries). Not explicitly mentioned, but the BOM includes it[[77]](file://file-CfaUmypVLUTDocYKrGVgAA#:~:text=Centralizes%20versions%20to%20avoid%20drift,Kafka%2C%20OTel%2C%20springdoc%2C%20Resilience4j%2C%20Testcontainers) which means we have it available. Possibly some shared patterns (maybe not directly used in the library but for services to use). Could be out of scope for the guide, but since BOM lists it, we mention it as part of our stack (services can use it to implement fallback/retry logic around external calls as needed).

Each of these terms/constructs plays a part in our developer guide narrative. The glossary ensures that both new joiners and experienced devs have a clear understanding of what each term means in our context, reducing ambiguity (e.g., knowing the difference between correlationId and traceId, or what exactly "idempotency" implies in HTTP context).

This comprehensive developer guide, combined with the well-structured shared library, is meant to enable developers to quickly onboard and build robust, compliant LMS microservices. They can focus on business logic, trusting that the shared foundation handles cross-cutting concerns consistently across the entire platform.