# PFS001 - SIGEA

Ordoñez Silva, Yonel Jr.

## Indice



# I. Propuesta de Estructura de Archivos del Proyecto

## 1.1. Estructura General del Proyecto

Se opto por una estructura Monolitica Monorepo, donde todos los servicios se desarrollan en un unico respositorio, el flujo de cambios y versiones se hara a traves de `git flow`, cada integrante tendra una rama especifica para su servicio y solo tocara lo que le corresponde.

Si se desea interactuar con otro servicio se tiene que realizar una fusion (`merge`) hacia la rama develop del servicio solicitante y el servicio objetivo.

Para coordinar los componentes fuera de `services/` se realizaran reuniones generales donde se discuta y observe el avance de cada integrante, con el objetivo de poder realizar la orquestacion y configuracion necesarias de acuerdo a la configuracion inidividual de cada servicio desarrollado por cada integrante.

```
app/

  services/                  # Microservicios individuales
     user-service/          # Microservicio de usuarios
     document-service/      # Microservicio de documentos
     notification-service/  # Microservicio de notificaciones
     ...

  shared/                   # Código común y reutilizable
     libs/
        common-utils/         # Funciones y helpers comunes
        event-models/         # Definición de eventos compartidos
        security/             # Autenticación, JWT, etc.
     configs/
         logback.xml          # Logging centralizado

  infra/                    # Infraestructura común
     docker/
     k8s/
     terraform/

  docs/                            # Documentación técnica y de arquitectura
     openapi.yaml                 # Documentación OpenAPI/Swagger de la API
     architecture.md              # Explicación de la arquitectura del microservicio
     README.md                    # (Opcional) Introducción a la documentación, estructura
  gateway/                  # API Gateway (Spring Cloud Gateway, etc.)
     ...
  discovery/                # Service discovery (Eureka, Consul, etc.)
     ...
  config-server/            # Centralized config (Spring Cloud Config)
     ...
  api-docs/                 # OpenAPI specs o contratos
     user-service.yaml
     document-service.yaml
     ...
  .gitlab-ci.yml / pipeline.yml   # Pipeline de CI/CD
  docker-compose.yml       # Desarrollo local (opcional)
  README.md
```

## 1.2. Estructura Interna de Cada Microservicio

### 1.2.1. Estructura N°1

Esta arquitectura es una solucion mas generica para abordar la arquitectura propuestas (Microservicios Orientado a Eventos).

No existen divisiones complicadas.

```
user-service/
  src/
     main/
         java/com/example/userservice/
             config/                # Configuración del servicio
             controller/            # Controladores REST
             service/               # Lógica de negocio
             domain/                # Entidades del dominio (POJOs)
             repository/            # Repositorios JPA/Mongo/etc.
             events/                # Publicación y consumo de eventos
             messaging/             # Configuración de Kafka/RabbitMQ
             dto/                   # DTOs para request/response
             exception/             # Manejo de errores y excepciones
             mapper/                # Conversores entre entidades y DTOs
             security/              # Seguridad (opcional)
             UserServiceApplication.java   # Main **class**
         resources/
             application.yml        # Config por entorno
             bootstrap.yml          # (si usas Spring Cloud Config)
             logback-spring.xml     # Logging
             ...
         test/
             unit/               # Test de servicios, lógica interna (sin contexto Spring)
             integration/        # Test con contexto Spring completo
             contract/           # Test de contratos con otros servicios (si usas Pact)
             resources/          # Configs de test, data mock, testcontainers, etc.
  Dockerfile
  build.gradle / pom.xml
  README.md
```

#### 1.2.1.1. Descripcion de Componentes para la Estructura N°1

| Carpeta | Propósito |
|---|---|
| `config/` | Beans de configuración (CORS, Swagger, Kafka, mappers, etc.) |
| `controller/` | Endpoints expuestos por el servicio |
| `service/` | Lógica de negocio (orquesta repos, eventos, etc.) |
| `domain/` | Entidades del modelo de dominio (no deben tener lógica de infraestructura) |
| `repository/` | Interfaces para JPA o acceso a datos (implementadas por Spring Data) |
| `events/` | Publicadores y consumidores de eventos del microservicio |
| `messaging/` | Configuración de Kafka/RabbitMQ y serialización de eventos |
| `dto/` | Objetos de entrada/salida (sin lógica) |
| `mapper/` | Conversores entre entidades y DTOs |
| `exception/` | Clases para manejar errores de negocio o del sistema |
| `security/` | Filtros, autenticación, autorización (si aplica) |

### 1.2.2. Estructura N°2

En proyectos más grandes, especialmente con arquitectura limpia, hexagonal o DDD, el codigo se separa por capas o contextos, no solo por paquetes técnicos (como service, controller, etc.).Y ahí entra la decisión de mover core, infra, shared, etc. como carpetas de primer nivel dentro de src/main/java/, es decir fuera del dominio de /java/com/…/yourapp

Para poder configurar esta estructrua realizamos lo siguiente:

1. Tener tu Application.java en: src/main/java/com/yourcompany/yourapp/Application.java
2. Y en ese archivo, configurar tu @ComponentScan para que incluya:

```
@SpringBootApplication(scanBasePackages = {
    "com.yourcompany.yourapp",
    "core",
    "infra",
    "api",
    "shared",
    /* Aqui añadimos mas carpetas si la estructura cambia */
})
```

```
user-service/
  src/
     main/
         java/com/yourcompany/yourapp/

         api/                        # Entrypoint HTTP: controllers, handlers, rutas
```

```
            controller/            # @RestController / @Controller
                DocumentController.java
            dto/                   # DTOs (Request/Response)
                DocumentRequest.java
                DocumentResponse.java
            error/                 # Manejo de errores (handlers, excepciones HTTP)
                GlobalExceptionHandler.java
            middleware/            # Filtros, interceptores, middlewares
                LoggingInterceptor.java
            docs/                  # Configuración de OpenAPI/Swagger
                OpenApiConfig.java

        config/                    # Configuraciones del microservicio
            AppConfig.java
            DatabaseConfig.java
            SecurityConfig.java
            EventBusConfig.java
            CorsConfig.java
            Properties/            # Clases con @ConfigurationProperties
                AppProperties.java

        core/                      # Lógica de negocio (Dominio puro)
            model/                 # Entidades del dominio (no JPA)
                Document.java
            service/               # Casos de uso / servicios de dominio
                DocumentService.java
            port/                  # Interfaces (puertos) de entrada y salida
                in/                # Port de entrada (casos de uso invocables)
                    CreateDocumentUseCase.java
                out/               # Port de salida (persistencia, eventos, APIs externas)
                    DocumentRepositoryPort.java
                    EventPublisherPort.java
            exception/             # Excepciones del dominio
                DocumentNotFoundException.java

        infra/                     # Adaptadores (Infraestructura)
            db/                    # Acceso a base de datos
                entity/            # @Entity de JPA
                    DocumentEntity.java
                repository/        # Interfaces JpaRepository y adaptadores de repositorio
                    JpaDocumentRepository.java
                    DocumentRepositoryAdapter.java  # Implementa port.out.DocumentRepository
```

```
            mapper/             # Mappea Entity <-> Dominio
                DocumentMapper.java
        events/                 # Adaptadores para eventos (Kafka, RabbitMQ, etc.)
            publisher/          # Produce eventos
                DocumentEventPublisher.java  # Implementa port.out.EventPublisherPort
            listener/           # Consume eventos (suscripción)
                DocumentCreatedListener.java
            model/              # Modelos del evento (DTOs de eventos)
                DocumentCreatedEvent.java
        client/                 # Integraciones externas (REST, SOAP, etc.)
            UserClient.java
            PaymentClient.java
        config/                 # Beans de infraestructura
            KafkaConfig.java

    shared/                     # Componentes compartidos entre capas
        utils/                  # Utilidades generales
            DateUtils.java
            JsonUtils.java
        constants/              # Constantes globales (no configurables)
            Roles.java
            ErrorMessages.java
            AppConstants.java
        enums/                  # Enums usados globalmente
            DocumentStatus.java

    mappers/                    # MapStruct o manual mappers entre capas
        DocumentDtoMapper.java
        EventMapper.java

    Application.java            # Clase principal (@SpringBootApplication)

    resources/
        application.yml         # Config principal
        application-dev.yml     # Config por entorno
        application-prod.yml
        messages.properties     # Mensajes internacionalizados
        logback-spring.xml      # Config de logs
        static/                 # Archivos estáticos (favicon, docs)
    test/
        unit/                # Test de servicios, lógica interna (sin contexto Spring)
        integration/         # Test con contexto Spring completo
```

```
            contract/           # Test de contratos con otros servicios (si usas Pact)
            resources/          # Configs de test, data mock, testcontainers, etc.
Dockerfile
build.gradle / pom.xml
README.md
```

#### 1.2.2.1. Descripcion de Componentes para la Estructura N°2

1. `api/`

Contiene controladores HTTP y DTOs (entrada/salida). No contiene lógica de negocio. Se conecta con los servicios del dominio (core.port.in).

2. `core/`

Contiene el dominio puro (modelo + lógica de negocio). Usa interfaces para la persistencia y eventos (no depende de frameworks). Alta testabilidad.

3. `infra/`

Implementa los puertos de salida (persistencia, eventos, llamadas HTTP externas). Aquí van las entidades JPA, los producers Kafka, etc.

4. `config/`

Configuraciones Spring: seguridad, base de datos, eventos, CORS.

Uso de @ConfigurationProperties si hay propiedades complejas.

5. `shared/`

Utilidades, constantes, enums que pueden ser usados en todas las capas. No contienen lógica de negocio ni acceso a frameworks.

6. `mappers/`

Uso de MapStruct o conversores manuales entre:

core.model  infra.db.entity

core.model  api.dto

7. `resources/`

Configs YAML, logs, migraciones, mensajes i18n. Aquí podrías incluir los archivos para Flyway/Liquibase también.

8. `test/`

Carpeta donde se reunen todas las pruebas para componentes individuales, integraciones y generales.

### 1.2.3. Estructura N°3

Es similar a la Estructura N°2, pero esto encapsula todo dentro del mismo dominio

```
src/
  main/
      java/
          com/empresa/app/
              Application.java                # Punto de entrada (@SpringBootApplication)

              api/                            # Interfaces de entrada (REST, eventos)
                  controller/                 # Controladores REST (HTTP)
                  eventlistener/              # Listeners de eventos (Kafka, RabbitMQ)
                  dto/                        # DTOs de entrada/salida

              core/                           # Lógica del dominio (pura, sin Spring)
                  model/                      # Entidades del dominio
                  service/                    # Casos de uso / reglas de negocio
                  exception/                  # Excepciones del dominio
                  repository/                 # Interfaces de persistencia (puertos)

              infra/                          # Implementaciones técnicas (DB, broker, etc.)
                  config/                     # Configuración de Spring, beans, properties
                  persistence/                # Implementaciones JPA, Mongo, etc.
                      entity/                 # Entidades JPA (ORM)
                      adapter/                # Implementaciones de repositorios
                  messaging/                  # Publicadores de eventos, consumidores, mapper
                  mapper/                     # Conversores entre entidades y dominio

              shared/                         # Código común reutilizable
                  constants/                  # Constantes globales
                  enums/                      # Enums compartidos
                  utils/                      # Utilidades generales
                  events/                     # Definición de eventos (clases de dominio)

              config/                         # Carga de configuración externa (env, profile
                  AppProperties.java
```

```
    resources/
        application.yml                    # Configuración principal
        application-dev.yml                # Config para entorno de desarrollo
        application-prod.yml               # Config para producción
    test/
        unit/               # Test de servicios, lógica interna (sin contexto Spring)
        integration/        # Test con contexto Spring completo
        contract/           # Test de contratos con otros servicios (si usas Pact)
        resources/          # Configs de test, data mock, testcontainers, etc.
```

#### 1.2.3.1. Descripcion de Componentes para la Estructura N°3

- *Application.java*
  - Punto de entrada principal.
  - Contiene @SpringBootApplication, habilita @ComponentScan (escanea todo bajo com.empresa.app).
- *api/*: Capa de entradaContiene todos los puntos de entrada al microservicio:
  - controller/: endpoints REST (@RestController).
  - eventlistener/: escucha de eventos Kafka, RabbitMQ, etc.
  - dto/: objetos de entrada/salida (no entidades del dominio, solo transporte).
- *core/*: Lógica del dominio (núcleo): Aquí vive la regla de negocio pura, sin dependencias externas:
  - model/: entidades de negocio.
  - service/: casos de uso (reglas, procesos).
  - exception/: errores específicos del dominio.
  - repository/: interfaces de persistencia (abstractas, no concretas).
- *infra/*: Adaptadores / Implementación técnica: Aquí está todo lo que depende de tecnologías externas:
  - **config/**: configuración de Spring, beans, seguridad, CORS, mappers, Swagger, etc.
  - **persistence/**:
    - ***entity/***: entidades ORM (JPA/Hibernate, Mongo, etc.).
    - ***adapter/***: implementación de core.repository.
  - **messaging/**:
    - Productores y consumidores de eventos.
    - Serialización de eventos, publicación, etc.

  - **mapper/**: convierte entre entidades de dominio y entidades JPA o DTOs.

- *shared/*: Código reutilizable

  - **constants/**: mensajes, códigos, nombres fijos.
  - **enums/**: enumeraciones compartidas (roles, estados, tipos, etc.).
  - **utils/**: funciones auxiliares genéricas.
  - **events/**: clases que representan eventos del dominio.

- *config/ (opcional)*: Si quieres separar las clases que cargan @ConfigurationProperties o settings globales.

- *test/*: Carpeta donde se reunen las pruebas de componentes, integracion y generales.

#### 1.2.4. Justificacion de propuesta para Estructura N°2 y Estructura N°3:

1. Cada microservicio necesita:

   - Un dominio (`core/`) con lógica y reglas de negocio.
   - Una forma de exponer funciones: HTTP (`api/`) y/o eventos (`events/`).
   - Adaptadores técnicos: base de datos, brokers, integraciones externas (`infra/`).
   - Utilidades compartidas (`shared/`).

2. La estructura permite los eventos (Aquitectura Orientada a Eventos):

| Necesidad | ¿Dónde lo haces? |
|---|---|
| Recibir eventos | `infra/events/listener` |
| Publicar eventos | `infra/events/publisher` |
| Procesar eventos como casos de uso | `core/service`, accediendo vía `port.in` |
| Comunicarte con base de datos | `core.port.out` + implementación en `infra/db/` |
| Lógica de negocio central | `core/service` y `core/model` |
| Constantes, enums reutilizables | `shared/` |
| Configuración de Kafka, Rabbit, etc | `infra/config/` |
| Reutilizar validaciones o utilidades | `shared/utils/` |

3. El Flujo de eventos es claro:

i. `Llega un evento desde Kafka (Listener en infra.events.listener)`

ii. `Se parsea y valida el mensaje (infra o shared)`

iii. Se llama a una interfaz de dominio (core.port.in)

iv. `El servicio del dominio (core.service) ejecuta la lógica de negocio`

v. `Se puede guardar algo en base de datos (a través de core.port.out → infra.db)`

vi. `Se emite un nuevo evento si es necesario (core.port.out.EventPublisher → infra.events.pu`

3. Permite la publicacion y escucha de eventos:

- Publicar eventos (emisión): Desde el dominio (core.service) decides que un evento debe emitirse.
  - Se llama a un port.out.EventPublisherPort → se implementa en infra.events.publisher.
- Escuchar eventos (consumo): infra.events.listener tiene listeners de Kafka, Rabbit, etc. (con @KafkaListener por ejemplo).
  - Este listener convierte el mensaje a un DTO y lo envía al dominio usando un port.in.

4. El Manejo de BD se realiza de la siguiente manera:

| Elemento | Ubicación | Descripción |
|---|---|---|
| Modelo JPA | `infra/db/entity/` | Entidad anotada con `@Entity` |
| Repositorio Spring Data | `infra/db/repository/` | Interfaces `JpaRepository` |
| Adaptador del repositorio | `infra/db/repository/DocumentRepositoryAdapter` | Implementa `DocumentRepositoryPort` |
| Declaración de interfaz del dominio | `core/port/out/` | Abstracción `DocumentRepositoryPort` |
| Uso en servicios | `core/service/` | El caso de uso depende de la interfaz |

### 1.2.2. Comunicacion entre Microservicios

- *Eventos (asíncronos)*: Usar Kafka, RabbitMQ o NATS. Ej: UserCreatedEvent, DocumentUploadedEvent o Serializados como JSON o Avro.
- *REST (síncronos)*: Sólo si es estrictamente necesario.
- *API Gateway*: Unifica y enruta peticiones al microservicio correspondiente.