Componente Java y .NET para la gestión de auditoría

Dirección General de Tecnologías de la Información y Comunicaciones

Áreas de Gobierno Tecnológico de SSII y del Dato

Servicio Andaluz de Salud
Consejería de Sanidad, Presidencia y Emergencias

Contenido

Resumen

- **Versión:** v01r13
- **Fecha publicación:** 8 jun 2026
- **Entrada en vigor desde:** 8 jun 2026

blocked URL

Cumplimiento normativo

Las normas expuestas son de obligado cumplimiento. La STIC podrá estudiar los casos excepcionales los cuales serán gestionados a través de los responsables del proyecto correspondiente y autorizados por el Área de Gobernanza de la STIC. Asimismo cualquier aspecto no recogido en estas normas deberá regirse en primera instancia por las guías técnicas correspondientes al esquema nacional de seguridad y esquema nacional de interoperabilidad según correspondencia y en su defecto a los marcos normativos y de desarrollo software establecidos por la Junta de Andalucía, debiendo ser puesto de manifiesto ante la STIC.

La STIC se reserva el derecho a la modificación de la norma sin previo aviso, tras lo cual, notificará del cambio a los actores implicados para su adopción inmediata según la planificación de cada proyecto.

En el caso de que algún actor considere conveniente y/o necesario el incumplimiento de alguna de las normas y /o recomendaciones, deberá aportar previamente la correspondiente justificación fehaciente documentada de la solución alternativa propuesta, así como toda aquella documentación que le sea requerida por la STIC para proceder a su validación técnica.

Contacto Arquitectura: l-arquitectura.stic@juntadeandalucia.es

Histórico de cambios

Los cambios en la normativa vendrán acompañados de un registro de las modificaciones. De este modo se podrá realizar un seguimiento y consultar su evolución. Ordenándose de mas recientes a menos recientes, prestando especial cuidado a las cabezeras de la tablas dónde se indican las fechas de entrada en vigor y versión.

Versión	v01r12	**Fecha publicación**	15 ene 2019	**Fecha entrada en vigor**	15 ene 2019
Alcance					
• Versión inicial sobre el componente para facilitar la aplicación de la norma por parte de terceros proveedores					

Versión	v01r13	**Fecha publicación**	8 jun 2026	**Fecha entrada en vigor**	8 jun 2026
Alcance					
• Se incorpora Audita como solución transversal para la auditoría de negocio en los sistemas de información del SAS. • Se depreca el uso del componente Audit4J-SAS en nuevos desarrollos. No obstante, su uso sigue siendo válido en aquellos sistemas que ya lo tuvieran implementado a la fecha de publicación de esta normativa.					

Selección de solución de auditoría

Introducción

Con el objetivo de facilitar el cumplimiento de la normativa del SAS en cuanto auditoría se refiere, desde el área de gobernanza de la STIC se proporcionan dos posibles soluciones basadas en componentes que permitan a los desarrolladores incorporar todo el sistema de auditorías en sus aplicaciones de forma rápida y estandarizada, minimizando el tiempo necesario que se requiere para realizar este proceso.

En el SAS coexisten actualmente dos soluciones para integrar auditoría en aplicaciones. Esta página documenta ambas alternativas y establece los criterios de elección conforme a la normativa técnica vigente y a los principios de la arquitectura de referencia del SAS.

- Audita
- Audit4J-SAS

Audita (Solución corporativa vigente)

Solución corporativa de auditoría (Audita)

Introducción

El objetivo principal de Audita es disponer de un sistema de información independiente, único y centralizado para la consolidación y registro de todos los logs de aplicación auditoría de negocio de todos los sistemas de información del SAS independientemente de si son asistenciales, económico financieros, de recursos humanos o cualquier otro ámbito. Con esto se pretende consolidar en un único entorno y base de datos centralizada toda esta información, de forma que se consigan mejoras posteriores heredadas de esta arquitectura, como la mejora en la consulta de información para las auditorías de acceso a datos que suelen requerirse por parte de organismos judiciales o policiales.

Con el objetivo de facilitar el cumplimiento de la normativa del SAS en cuanto auditoría se refiere, desde el área de gobernanza de la STIC se ha realizado un conjunto de componentes reutilizables que permiten a los desarrolladores incorporar todo el sistema de auditorías en sus aplicaciones de forma rápida y estandarizada, minimizando el tiempo necesario que se requiere para realizar este proceso.

Propósito y Visión

El proyecto AUDITA surge de la necesidad estratégica del Servicio Andaluz de Salud (SAS) de establecer un repositorio centralizado, único e independiente para la consolidación de registros de auditoría de negocio.

Actualmente, la dispersión de logs en sistemas asistenciales, económicos y de recursos humanos dificulta la trazabilidad y la respuesta ante requerimientos de organismos reguladores o judiciales. AUDITA resuelve este problema proporcionando una capa de persistencia unificada que garantiza la integridad, la disponibilidad y la facilidad de consulta de la actividad crítica de todos los sistemas de información del SAS.

Alcance Funcional

El sistema actúa como un hub de auditoría transversal, permitiendo la captura de eventos desde entornos heterogéneos (desde arquitecturas modernas de microservicios en contenedores hasta sistemas *legacy* en entornos *baremetal*), asegurando que cada acción de negocio quede debidamente documentada con un contexto completo (identidad, objeto, acción y resultado).

Resumen de Componentes

La solución se articula mediante una arquitectura basada en eventos y microservicios, compuesta por los siguientes pilares:

- **Capa de Ingesta:** Mediante librerías específicas, Sidecars y microservicios productores.
- **Capa de Mensajería (Buffer):** Basada en **Apache Kafka** para garantizar el desacoplamiento y la alta disponibilidad de la entrada de datos.
- **Capa de Procesamiento:** Microservicios en **Java (MicroProfile)** que consumen y transforman los eventos.
- ***Capa de Persistencia:*** *Almacenamiento escalable en* ***MongoDB****, optimizado para grandes volúmenes de datos mediante estrategias de* sharding.
- **Capa de Consumo y Observabilidad:** Interfaz de usuario en **Lit** para consultas analíticas y stack de monitorización basado en **Prometheus/Grafana**.

Estrategia de Consumo

Para garantizar un consumo ágil y minimizar el acoplamiento técnico entre los sistemas del SAS y la plataforma AUDITA, se han definido dos mecanismos fundamentales:

- **Consumo a través Audita Connector + Audita Sidecar** : Opción recomendada, ya que permite una integración sencilla, limpia, y desacoplada.
    - **`audita-connector` (Capa de Aplicación)** : Es la componente de tipo librería diseñada para ser integrada directamente en el proyecto. Su objetivo es facilitar la producción de eventos de negocio de forma declarativa.
        - **Mecanismo:** Proporciona la anotación Java @Auditable o el atributo [Auditable] en el caso de .NET, permitiendo marcar los procesos de negocio que requieren trazabilidad sin necesidad de escribir lógica de intercepción manual, delegando en el componente la captura del contexto de ejecución. Simplemente decorando los métodos de negocio que requieran auditoría, audita-connector se encarga de comunicar a audita-sidecar las trazas para que puedan ser persistidas.
        - **Compatibilidad de Stacks:** Para asegurar la cobertura en gran parte del parque tecnológico del SAS, se distribuye en múltiples versiones:
            - **Java:** Versiones compatibles con **JDK 8+ (Java EE)**, y **JDK 17+ (Jakarta EE: `javax.*` `jakarta.*`)**.
            - **.NET:** Versión específica para aplicaciones en el ecosistema Microsoft.

    - **`audita-sidecar` (Capa de Transporte)**: Actúa como un intermediario (proxy) entre la aplicación y el núcleo de AUDITA. Este componente es totalmente autónomo, exponiendo una fachada REST que es consumida por audita-connector, y encargándose de transformar y gestionar la persistencia de la información según se configure.
        - ***Propósito:*** Funciona como puente de comunicación, permitiendo el **desacoplo técnico** completo. La aplicación solo se comunica con el sidecar (preferiblemente de forma local para menor latencia), y es este el encargado de gestionar la complejidad del envío hacia la plataforma Audita.
        - **Reutilización:** Este componente es agnóstico al lenguaje, por lo que **se comparte** tanto para los proyectos Java como para los de .NET, simplificando el mantenimiento de la infraestructura.
- **Consumo directa con Audita** : Opción reservada para escenarios donde no sea posible hacer uso de Audita Connector por causas justificadas.

> **DevKit**
>
> **Los detalles técnicos de Audita, así como las instrucciones para integrar y configurar sus componentes en los proyectos se puede consultar en su Development Kit**

Histórico de cambios

Audita Connector

Java 8+ (Java EE)

Implementación base del componente.

API (`api`)

- Anotación `@Auditable` con atributos: `transformer`, `user` (UserResolver), `actionCode` (enum `ActionCode`), `auditableOnError`, `failIfError`.
- Interfaz `AuditInvocation` con acceso a `className`, `methodName`, `target` (objeto del método) y `auditableAnnotation`.
- Interfaz `AuditTransformer` y `UserResolver` para extensión.
- Interfaz `Client` con método `sendAsync(AuditDataDTO, String)`.

Interceptor (`interceptor`)

- `AuditableInterceptor`: intercepta métodos anotados con `@Auditable`. Captura **sólo el resultado** (`result`) del método; no captura `target` ni parámetros de entrada. La anotación se resuelve únicamente desde el método (`method.getAnnotation`).
- Estrategias de error iniciales: `FailIfErrorStrategy`, `SilentStrategy`, `AuditAndThrownExceptionStrategy`, `DontAuditAndThrowStrategy`.

Modelo (`model`)

- `AuditDataDTO`: campos `codigoProcesoEvento`, `codigoAccionRealizada` (tipo `ActionCode`), `error`, `usuarioDispositivo`, `nombreCentro`, `nombrePrograma`, `correlationId`, `payload`, `objetoPadre`, `objetoHijo`.
- Enum `ActionCode`: valores `CONSULTA`, `LOGIN`, `EDICION`, `CREACION`, `ELIMINACION`.
- Sin validación interna del DTO (sin paquete `validator`).

Cliente REST (`rest`)

- `AuditaSidecarRestClient` basado en `java.net.http.HttpClient`. Constructor único con `AuditSidecarRestClientConfig`.
- Envío asíncrono puro con `CompletableFuture`; sin bloqueo síncrono, sin validación del DTO antes del envío.
- Serialización JSON con `javax.json.bind` (JSON-B).

Sin cambios funcionales en la API pública ni en la lógica de negocio.

- Corrección de anotación `@Test` ausente en tres casos de prueba de `AuditaSidecarRestClientTest` que impedía su ejecución.
- Corrección de las aserciones de nombre de campo JSON serializado: `"code"` `"codigoProcesoEvento"` y `"actionCode"` `"codigoAccionRealizada"` para reflejar el contrato real del DTO.

Cambios de implementación relevantes.

API (`api`)

- `AuditInvocation`: se añaden `getParameters()` (`Object[]`) y `getResult()` al contrato de la interfaz. El interceptor ahora expone tanto los parámetros de entrada como el resultado al `AuditTransformer`.
- `@Auditable`: `actionCode` renombrado a `action`; el tipo cambia de `ActionCode` a `Action`. Ambos atributos (`user` y `action`) marcados como `@Deprecated(forRemoval=true)`, promoviendo que la resolución se haga en el `AuditTransformer`.
- Eliminación de `Result.java`.

Interceptor (`interceptor`)

- `AuditableInterceptor`: ahora captura `target` (instancia sobre la que se invoca) y `parameters` (argumentos del método) además del resultado, pasándolos a `AuditInvocationDefault`.
- Uso de `java.util.logging.Logger` en lugar de SLF4J.
- Estrategias de error refactorizadas: `FailIfErrorStrategy` renombrada a `ClientFailIfErrorStrategy`; nueva estrategia `ValidationFailIfErrorStrategy` para tratar fallos de validación del DTO de forma diferenciada.
- La firma de `onFailure` en todas las estrategias se extiende para recibir `target`, `parameters` y `result`.

Modelo (`model`)

- Enum `ActionCode` renombrado a `Action`; valores renombrados a español técnico: `CREAR`, `CONSULTAR`, `ACTUALIZAR`, `ELIMINAR` (se elimina `LOGIN`).
- Nuevo paquete `validator` con cadena de responsabilidad Chain of Responsibility: `AuditDataDTOValidator`, `AgentValidator`, `EntityParentObjectValidator`, `EntityChildObjectValidator`, `NullValidator`, `ValidationHandler`.
- Nueva excepción `AuditaNotValidException` para señalizar fallos de validación del DTO.
- Modelo de error: `IssueError`, `MessageError` para representar errores de validación estructurados.
- `AuditDataDTO` añade `fechaAccion` y `fechaRegistroAuditoria` (`LocalDateTime`).

Cliente REST (`rest`)

- `AuditaSidecarRestClient` migrado de `java.net.http.HttpClient` a **cliente JAX-RS (`javax.ws.rs`)** con `ClientBuilder` e `Invocation.Builder`. Implementa `AutoCloseable` y gestiona su ciclo de vida con `ExecutorService` + `ScheduledExecutorService` para timeout.
- Se añade validación previa del DTO (`dto.validate()`) antes del envío HTTP; un DTO inválido lanza `AuditaNotValidException` sin realizar llamada de red.
- Múltiples constructores para inyección en tests.

Correcciones de compatibilidad sin cambios funcionales.

Interceptor (`interceptor`)

- `AuditableInterceptor`: prioridad fijada a valor literal `2000` en lugar de `Interceptor.Priority.APPLICATION` para compatibilidad con CDI 1.1 (Java EE 7).
- Nuevo método privado `resolveAuditable(Method, Object)`: resuelve la anotación `@Auditable` contemplando proxies CDI (clase proxy, superclase, `declaringClass`) evitando `NullPointerException` en contenedores que generan proxies para beans CDI.

Cliente REST (`rest`)

- Dependencia CDI reducida de 2.0 1.2 **1.1** para asegurar compatibilidad con WSL 12 (Java EE 7 / WebSphere Liberty 12.2.1.4).
- `maven-enforcer-plugin` reducido a **3.5.0** para compatibilidad con Maven 3.6.1 del entorno CI.
- Eliminación de `@Deprecated(forRemoval=true)` en los atributos de `@Auditable` ya que `forRemoval` no está soportado en JDK 8.
- Ampliación de la suite de tests del cliente REST y documentación Javadoc.

Corrección funcional mayor: el interceptor ahora audita correctamente métodos que devuelven `CompletionStage` o `Future`.

Interceptor (`interceptor`)

- `AuditableInterceptor`: el método `intercept` detecta el tipo de retorno del método auditado y bifurca:
  - **Síncrono**: comportamiento idéntico a versiones anteriores.
  - **`CompletionStage`** : envuelve el stage con `wrapCompletionStageWithAudit`; la auditoría se emite al completarse la operación de negocio (éxito o error), no antes.
  - **`Future`** : convertido a `CompletableFuture` con timeout mediante `wrapFutureWithAudit` y tratado como `CompletionStage`.
- Nuevo `ScheduledExecutorService` estático (`TIMEOUT_SCHEDULER`, daemon) para aplicar timeout a `CompletionStage` en JDK 8 sin `CompletableFuture.orTimeout` (método no disponible hasta JDK 9). Timeout configurable mediante `ASYNC_AUDIT_TIMEOUT_MINUTES` (5 minutos por defecto) para evitar stages que nunca se completan.
- Nuevo método `withTimeoutJava8(CompletableFuture, timeout, unit)`: implementación portable de timeout para JDK 8.
- Cachés estáticas con límite (`InstanceCache`, `MAX_CACHE_SIZE = 10000`) para `Auditable`, `AuditTransformer` y `UserResolver` por `Method`, evitando instanciación repetida y previniendo memory leak en entornos con proxies dinámicos.
- `AuditFailureStrategyFactory` extraída como campo de instancia en lugar de crearse en cada invocación.
- Nuevo paquete `interceptor/util/InstanceCache.java`.
- `strategyFactory` inicializado en el constructor (una sola vez).

Java 17+ (Jakarta EE: javax.* jakarta.*)

Ruptura de namespace `javax` `jakarta`. Equivalente funcional a 1.1.0.1.

API (`api`)

- Todos los imports migrados de `javax.*` → `jakarta.*` (`jakarta.interceptor`, `jakarta.enterprise`, `jakarta.inject`).
- `@Auditable`: atributos `user` y `action` marcados `@Deprecated(forRemoval=true)`.
- `AuditInvocation`: mismos contratos que 1.1.0.1 (`getParameters()`, `getResult()`).

Interceptor (`interceptor`)

- `AuditableInterceptor`: prioridad restaurada a `Interceptor.Priority.APPLICATION` (disponible en CDI 2.0+/Jakarta). **No incluye** el método `resolveAuditable` de la rama 1.1.1.1; resolución de `@Auditable` sólo desde el método, sin soporte de proxies CDI.
- Mismas estrategias de error que 1.1.0.1: `ClientFailIfErrorStrategy`, `ValidationFailIfErrorStrategy`, `SilentStrategy`, `AuditAndThrownExceptionStrategy`, `DontAuditAndThrowStrategy`.

Modelo (`model`)

- Enum `Action`: mismos valores que 1.1.0.1 (`CREAR`, `CONSULTAR`, `ACTUALIZAR`, `ELIMINAR`).
- `AuditDataDTO` incluye `validate()` con `AuditDataDTOValidator` y `fechaAccion`/`fechaRegistroAuditoria`.
- Paquete `validator` idéntico al de 1.1.0.1.

Cliente REST (`rest`)

- `AuditaSidecarRestClient` basado en java.net.`http.HttpClient` (JDK nativo), **no en JAX-RS**. Constructor principal con `(config)` y constructor de test con `(config, HttpClient, Jsonb)`.
- Serialización con `jakarta.json.bind` (JSON-B).
- Envío semi-síncrono: lanza la petición async y bloquea con `.get(timeout)` para propagar `ClientException` síncronamente al interceptor.
- Validación del DTO previa al envío (`dto.validate()`).

Corrección funcional en `AuditInvocationDefault`.

- `AuditableInterceptor`: se añade captura de `context.getTarget()` además de `context.getParameters()`. `createAuditInvocation` ahora recibe y propaga `target` separado de `parameters`, corrigiendo que `setTarget()` recibía los parámetros en lugar del objeto invocado.
- Corrección de tests unitarios afectados por el cambio.
- Ajuste menor de versiones de plugins Maven para pipeline CI/CD.

Equivalente funcional a 1.1.1.1.

Interceptor (`interceptor`)

- `AuditableInterceptor`: incorporación del método `resolveAuditable(Method, Object)` para resolución segura de `@Auditable` en proxies CDI, igual que en 1.1.1.1.

Cliente REST (`rest`)

- `AuditaSidecarRestClient` migrado de `java.net.http.HttpClient` a **cliente JAX-RS (`jakarta.ws.rs`)** con `ClientBuilder` e `Invocation.Builder`, alineado con Gobierno Tecnológico SAS (WebServices REST JAX-RS). Implementa `AutoCloseable` con gestión de `ExecutorService` y `ScheduledExecutorService`.
- Múltiples constructores para inyección de ejecutores en tests.
- Ampliación masiva de la suite de tests del cliente REST (`AuditaSidecarRestClientTest`, nueva clase `RestClientTest`).

Equivalente funcional a 1.1.2.1. Incorpora además mejoras de rendimiento y robustez en el cliente REST.

Interceptor (`interceptor`)

- `AuditableInterceptor`: mismas capacidades asíncronas que `1.1.2.1`:
  - Detección del tipo de retorno (`CompletionStage`, `Future`, síncrono) con auditoría diferida hasta completar la operación de negocio.
  - Timeout de 5 minutos para stages que no se completan (`ASYNC_AUDIT_TIMEOUT_MINUTES`).
  - Cachés estáticas con límite (`InstanceCache`, `MAX_CACHE_SIZE = 10000`) para `Auditable`, `AuditTransformer` y `UserResolver`.
  - `AuditFailureStrategyFactory` como campo de instancia.
  - Nuevo paquete `interceptor/util/InstanceCache.java`.
- A diferencia de 1.1.2.1, **no necesita** `withTimeoutJava8` ni `TIMEOUT_SCHEDULER` estático: usa `CompletableFuture.orTimeout` (disponible nativo en JDK 9+) cuando el stage es `CompletableFuture`.

Cliente REST (`rest`)

- `AuditaSidecarRestClient`: el cliente JAX-RS (`jaxrsClient`) se construye **una sola vez en el constructor** en lugar de por cada llamada `sendAsync`, eliminando la creación y destrucción repetida de clientes HTTP.
- `jaxrsClient` se cierra únicamente en `close()`, no en el bloque `finally` de cada petición.
- `ThreadPoolExecutor` con **cola acotada** (`ArrayBlockingQueue`, capacidad 1000) y política `CallerRunsPolicy` como back-pressure, en sustitución del `Executors.newCachedThreadPool()` sin límite que podía provocar `OutOfMemoryError` bajo carga extrema. Parámetros: `corePoolSize=5`, `maxPoolSize=50`, `keepAlive=60s`.

- Nuevo constructor `AuditaSidecarRestClient(config, jsonb, requestExecutor, timeoutScheduler, clientBuilder)` con `ClientBuilder` inyectable para facilitar tests sin sobrescritura de método virtual en constructor.

.Net 8

Implementación base del componente. Equivalente funcional a Java `2.0.1.1`.

API (`SAS.Audita.Connector.Api`)

- `AuditableAttribute`: atributo `[Auditable]` aplicable a método o clase (`AttributeTargets.Method | AttributeTargets.Class`, `Inherited = true`). Atributos: `Transformer`, `User` (marcado `[Obsolete]`), `Action` (marcado `[Obsolete]`), `AuditableOnError`, `FailIfError`.
- `IAuditInvocation`: interfaz con `ClassName`, `MethodName`, `Target`, `Parameters`, `Result`, `Auditable Attribute`, `IsError`. Contrato completo desde el inicio (equivalente a Java `1.1.0.1+`).
- `IAuditTransformer` / `IUserResolver`: contratos de extensión para transformación de datos y resolución de usuario.
- `IClient`: contrato del cliente de envío con firma `SendAsync(AuditDataDTO, string, CancellationToken)`.
- `AuditInvocationDefault`: implementación por defecto de `IAuditInvocation`.
- `DefaultAuditTransformer` / `DefaultUserResolver`: implementaciones por defecto.

Interceptor (`SAS.Audita.Connector.Interceptor`)

- `AuditableInterceptor`: implementado como `DispatchProxy` (patrón de proxy dinámico de .NET). Creación mediante método estático `Create<T>(target, client, logger)`.
- Método `Invoke`: detecta el tipo de retorno y bifurca:
  - **Síncrono**: audita tras la ejecución y propaga `ClientException` o silencia según estrategia.
  - **`Task` (void async)**: `WrapTask` — audita al completarse (`await`), con soporte de `auditableOnError`.
  - **`Task<T>` (async con resultado)**: `WrapTaskOfT<T>` — resuelve el resultado real y audita al completarse. Usa reflexión con caché para construir el método genérico en tiempo de ejecución.
- `ResolveAuditable`: busca `[Auditable]` en el método, tipo del target, superclase o `DeclaringType`. Cachea resultado por `(MethodInfo, TargetType)` — incluye `null` para métodos no anotados (ejecución sin auditoría).
- `InstanceCache<TKey, TValue>` en `Util/`: caché thread-safe con `ConcurrentDictionary` y límite configurable (`MaxCacheSize = 10000`) para `TransformerCache`, `ResolverCache` y `WrapTaskOfTCache`.

- `AuditFailureStrategyFactory` como campo de instancia, inicializada en `Create`.
- `ILogger<AuditableInterceptor>` inyectado (`Microsoft.Extensions.Logging`); soporta `NullLogger` si no se proporciona.
- Estrategias de error: `ClientFailIfErrorStrategy`, `ValidationFailIfErrorStrategy`, `SilentStrategy`, `AuditAndThrownExceptionStrategy`, `DontAuditAndThrowStrategy`.

Modelo (`SAS.Audita.Connector.Model`)

- `Action` (enum): valores `Crear`, `Consultar`, `Actualizar`, `Eliminar`, `Ejecutar` (este último no presente en Java). Incluye `ActionExtensions.GetCode()` y `EnumExtensions.ToDescription()`.
- `AuditDataDTO`: campos `Id`, `FechaAccion`, `FechaRegistroAuditoria`, `CodigoProcesoEvento`, `CodigoAccionRealizada` (`string`, no enum), `Error`, `UsuarioDispositivo`, `NombreCentro`, `NombrePrograma`, `CorrelationId`, `Payload`, `ObjetoPadre`, `ObjetoHijo`. Constructor vacío público y constructor interno por builder. Método `Validate()`.
- `AuditDataDTOBuilder`: builder fluido con métodos `With*`.
- `Agent`, `Entity`: value objects del modelo de auditoría.
- `AuditaNotValidException`, `AuditaServiceException`: excepciones del dominio.
- `IssueError`, `MessageError`: representación estructurada de errores de validación.
- Paquete `Validator/` con Chain of Responsibility: `AuditDataDTOValidator`, `AgentValidator`, `EntityParentObjectValidator`, `EntityChildObjectValidator`, `NullValidator`, `ValidationHandler`.

Cliente REST (`SAS.Audita.Connector.Rest`)

- `AuditaSidecarRestClient` (`sealed`): basado en `HttpClient` inyectado (compatible con `IHttpClientFactory`). El `HttpClient` se construye externamente y se pasa por constructor — ciclo de vida gestionado por el contenedor DI del consumidor.
- Serialización con `System.Text.Json` con `JsonNamingPolicy.CamelCase` y `DateTimeConverter` personalizado.
- Validación del DTO previa al envío (`dto.Validate()`).
- Manejo de errores: `ClientException` por timeout (`TaskCanceledException`), errores de red (`HttpRequestException`) y respuestas con código de error HTTP.
- Timeout configurado sobre el `HttpClient` inyectado si `config.Timeout > TimeSpan.Zero`; respeta timeout externo si fue configurado vía `IHttpClientFactory`.
- `AuditSidecarRestClientConfig` con patrón builder interno (`AuditSidecarRestClientConfig.Create().WithBaseUrl(...).WithTimeout(...).Build()`)

Audita Sidecar

Implementación base del sidecar. Único modo de operación: HTTP directo a la API Audita.

Arquitectura

- Estructura multi-módulo Maven: `application`, `boot`, `domain`, `http`, `jpa` (vacío), `rest`.
- Patrón hexagonal: dominio independiente, adaptadores `http` y `rest`.

Dominio (`domain`)

- `AuditDataDTO`: modelo central de evento de auditoría con campos `codigoProcesoEvento`, `codigoAccionRealizada` (`ActionCode`), `error`, `usuarioDispositivo`, `nombreCentro`, `nombrePrograma`, `correlationId`, `payload`, `objetoPadre`, `objetoHijo`.
- `ActionCode`: enum de acciones de auditoría.
- `AuditaRepository`: interfaz única de repositorio para creación de eventos.
- `AuditaCreateService`: caso de uso de creación de auditoría.

Lógica de negocio (`application`)

- `AuditaCreateServiceImpl`: implementación mínima — recibe `AuditDataDTO` y delega en `AuditaRepository` sin resiliencia ni métricas.

HTTP (`http`)

- `AuditDataHttpRepository`: envío del evento a la API Audita mediante cliente JAX-RS (javax.ws.rs).
- `PersistentHttpRepository`: persistencia de reintentos en modo HTTP.
- `AuditKeycloakAuthorization`, `AuditKeycloakEntity`, `KeycloakClient`: obtención y caché de token OAuth2 desde Keycloak.
- `AuditEventMapper` (en `http/fhirR5/mapper/`): mapeo de `AuditDataDTO` a recurso `AuditEvent` FHIR R5 (HAPI FHIR).
- `AuditEventMessageBodyWriter`: serialización del `AuditEvent` FHIR R5 a JSON para la petición HTTP.
- Mappers HTTP: `AgentHttpEntityMapper`, `AuditDataHttpEntityMapper`, `EntityHttpEntityMapper`, `ResponseAuditHttpEntityMapper`.

REST (`rest`)

- `AuditSidecarRest`: endpoint JAX-RS `POST /notificaciones/auditoria/eventos` — recibe `AuditDataDTOCommand`, mapea a dominio, invoca el caso de uso y devuelve `201 Created` con `Location`.
- `BusinessExceptionMapper`, `RuntimeExceptionMapper`, `ThrowableMapper`: mappers de excepción a respuesta HTTP.

Incremento mayor: el sidecar pasa de un único modo HTTP a una arquitectura dual HTTP/JPA seleccionable en tiempo de ejecución. Se añaden resiliencia, métricas y observabilidad.

Dominio (`domain`)

- `Action` (nuevo): enum que reemplaza a `ActionCode`, con valores `CREAR`, `CONSULTAR`, `ACTUALIZAR`, `ELIMINAR` alineados con los del conector Java.
- `ActionCode` (eliminado): sustituido por `Action`.
- `AuditaTransportType` (nuevo): enum que define los modos de transporte `HTTP` y `JPA`.
- `AuditDataJpaDTO` (nuevo): DTO para persistencia en base de datos, con campos `id`, `createdOn`, `version` y referencia al `AuditDataDTO` de dominio.
- `AuditoriaJpaRepository` (nuevo): interfaz de repositorio para el modo JPA.
- `AuditDataDTO`: adaptado para incluir los campos necesarios en ambos modos de transporte.
- Diccionarios FHIR (`AgentType`, `EntityExtensionCode`, `ExtensionCode`, `OutcomeCode`, `SourceCode`): movidos de `http/fhirR5/dictionary/` a `domain/fhir/r5/dictionary/` (desacoplamiento entre capas).
- `AuditEventMapper` (nuevo en `domain/fhir/r5/mapper/`): mapeo de `AuditDataDTO` a `AuditEvent` FHIR R5, movido al dominio.
- `FhirJsonSerializer` (nuevo en `domain/fhir/r5/serializer/`): serialización FHIR R5 a JSON, disponible en el dominio para uso por JPA.
- `AuditaEnvConfigSource` (nuevo): fuente de configuración MicroProfile desde variables de entorno.
- `ConfigHelper` (nuevo): utilidad para leer propiedades de configuración con valores por defecto tipados.
- `AuditaSidecarParameterException` (nuevo): excepción de dominio para parámetros inválidos.
- `MessageError`: ampliado con nuevos campos de detalle de error.

Lógica de negocio (`application`)

- `AuditaCreateServiceImpl`: refactorizado para soportar los dos modos de operación (`AUDITA_MODE=http` por defecto, `AUDITA_MODE=jpa`). Introducido `Instance<AuditaRepository>` e `Instance<AuditoriaJpaRepository>` para resolución CDI lazy de ambos repositorios. Añadidos `@Bulkhead(value=20, waitingTaskQueue=30)`, `@Timeout(5000)`, `@Counted` y `@Timed` (MicroProfile Fault Tolerance y Metrics). La operación JPA se ejecuta como `@Transactional(rollbackOn=Exception.class)`.

JPA (`jpa`)

- `AuditoriaEntity` (nuevo): entidad JPA mapeada a la tabla de persistencia de auditoría.
- `AuditoriaEntityMapper` (nuevo): mapeo entre `AuditDataJpaDTO` y `AuditoriaEntity`.
- `AuditaJpaRepositoryImpl` (nuevo): implementación del repositorio JPA.
- `AuditaJpaRepositoryProducer` (nuevo): productor CDI del repositorio JPA.

HTTP (`http`)

- `WebServiceClientBuilder`: actualizado para configuración dinámica del cliente JAX-RS.
- `AuditKeycloakAuthorization`: refactorización del flujo de autorización Keycloak.
- `AuditaProducerContants`: actualización de constantes de endpoint.
- `AuditDataDTOHttpEntity`, `EntityHttpEntity`: adaptados al nuevo modelo de dominio.
- `HttpReponseException`: ampliada con detalle de error estructurado.
- `AgentHttpEntityMapper`, `AuditDataHttpEntityMapper`, `EntityHttpEntityMapper`: adaptados al nuevo modelo.
- `AuditDataHttpRepository`, `ClientProducer`, `HttpContext`, `PersistentHttpRepository`: refactorizados para integración con la nueva arquitectura dual.
- `AuditEventMapper` en `http/fhir/r5/mapper/` y `FhirContextProducer` en `http/fhir/r5/handle/`: renombrado de paquete (`fhirR5` `fhir/r5`).

Arranque (`boot`)

- `LivenessCheck` (nuevo): sonda MicroProfile Health `/health/live`.
- `ReadinessCheck` (nuevo): sonda MicroProfile Health `/health/ready` con verificación de conectividad HTTP (modo HTTP) y JNDI/SQL (modo JPA), control de `@Bulkhead` y ventana deslizante de rechazos.

Correcciones en la deserialización del token Keycloak y en el repositorio HTTP.

HTTP (`http`)

- `AuditKeycloakEntity`: añadidas anotaciones JSON-B (`@JsonbProperty`, `@JsonbNillable`) para mapear correctamente los campos en `snake_case` de la respuesta de Keycloak (`access_token`, `expires_in`, `refresh_expires_in`, `token_type`). Corregido tipo de setter de `expiresIn y refreshExpiresIn de int a long` (evita truncamiento silencioso en tokens con TTL elevado).
- `KeycloakClient`: ajuste de imports (orden) y corrección de log.
- `PersistentHttpRepository`: corrección en el manejo de respuestas HTTP del repositorio de persistencia.
- `ClientProducer`: ajuste en la producción del cliente JAX-RS.

Primera versión del subsistema de warm-up. Resolución de problemas de configuración de unidad de persistencia externa.

Arranque (`boot`)

- `JndiWarmup` (nuevo): bean `@ApplicationScoped` con `@Priority(100)` que en el startup (sólo modo `AUDITA_MODE=jpa`) resuelve el `DataSource` mediante JNDI y abre una conexión de prueba. Reintenta hasta `AU`

DITA_JNDI_WARMUP_MAX_ATTEMPTS veces (5 por defecto) con espera de AUDITA_JNDI_WARMUP_DELAY_MS (1 s) entre intentos. Configurable con AUDITA_JNDI_WARMUP_ENABLED, AUDITA_JNDI_WARMUP_MAX_ATTEMPTS, AUDITA_JNDI_WARMUP_DELAY_MS, AUDITA_DATASOURCE_JNDI.

- ReadinessCheck: integrado con JndiWarmup. Nuevo handler para verificación de conectividad con base de datos mediante JNDI.

JPA (jpa)

- ExternalPersistenceUnitLoader (nuevo): carga dinámicamente una unidad de persistencia externa, permitiendo que el DataSource se configure mediante JNDI en lugar de estar embebido en persistence.xml.
- AuditaJpaRepositoryImpl, AuditaJpaRepositoryProducer: adaptados para soportar la unidad de persistencia externa.
- JpaSchemaHealthChecker: comprobación del esquema JPA durante el health-check.
- AuditoriaEntityMapper: ajustes en el mapeo para la nueva estructura de persistencia.

REST (rest)

- BusinessExceptionMapper, RuntimeExceptionMapper, ThrowableMapper: revisión de los mappers de excepciones para alinear los códigos de respuesta HTTP con el contrato REST (normalización de mensajes de error y estado HTTP).
- BulkheadRejectionRecorder: correcciones en el registro de rechazos de @Bulkhead para la sonda de readiness.

Lógica de negocio (application)

- AuditaCreateServiceImpl: corrección en createViaHttp para propagación correcta de excepciones de negocio.

Elimina la latencia de la primera petición causada por la inicialización lazy de HAPI FHIR.

Arranque (boot)

- FhirWarmup (nuevo): bean @ApplicationScoped con @Priority(300) que durante el startup ejecuta en orden: (1) inicializa el singleton estático de FhirContext vía FhirContextProducer, (2) crea y ejercita el parser JSON de HAPI FHIR con un Patient dummy, (3) ejecuta una serialización completa con FhirJsonSerializer usando un AuditDataDTO dummy válido, (4) ejecuta el mapeo con AuditoriaEntityMapper. Si el warmup falla el arranque se aborta (@Priority obliga secuencialidad). Configurable con AUDITA_FHIR_WARMUP_ENABLED.
- ReadinessCheck: timeout del health-check aumentado de 5000 ms a **15 000 ms** (por defecto) para acomodar el mayor tiempo de inicio con warm-ups. Métodos isSchemaValid() y canWriteAuditoria() expuestos

como `protected` para facilitar tests. Logging de errores migrado a lambdas (`() -> "..."`) para evitar evaluación eager de mensajes.

Dominio (`domain`)

- `FhirContextProducer` (nuevo en `domain/fhir/r5/serializer/`): centraliza la creación del singleton `FhirContext.forR5()`. El contexto FHIR se mueve de `http` a `domain` para ser reutilizable por warmups y persistencia JPA sin dependencia circular. El `FhirContextProducer` de `http` (en `http/fhir/r5/handle/`) es eliminado.
- `FhirJsonSerializer`: refactorizado para recibir `FhirContext` y `AuditEventMapper` como dependencias inyectadas (constructor) en lugar de instanciarlos internamente, permitiendo la reutilización del mismo contexto FHIR entre escrituras.

JPA (`jpa`)

- `JpaWarmup` (nuevo): bean `@ApplicationScoped` con `@Priority(200)` que verifica conectividad JNDI y realiza una query trivial al esquema durante el startup en modo `AUDITA_MODE=jpa`, precalentando el pool de conexiones JPA antes de la primera petición real.

Elimina la penalización de latencia en la primera petición productiva mediante warm-up proactivo de los proxies CDI y el stack CXF.

Arranque (`boot`)

- `JaxRsWarmup` (nuevo): bean `@ApplicationScoped` con `@Priority(500)` que lanza un hilo en background tras el inicio del contexto CDI. El hilo espera `AUDITA_JAXRS_WARMUP_DELAY_MS` (3 s por defecto) para garantizar que el puerto TCP está abierto y luego ejecuta una petición HTTP dummy al endpoint de negocio, forzando la inicialización completa de Apache CXF (escaneo de clases, registro de proveedores, construcción del modelo de recursos JAX-RS). Expone un flag estático `isWarmupCompleted()` para ser consultado desde `ReadinessCheck`. Configurable con: `AUDITA_JAXRS_WARMUP_ENABLED`, `AUDITA_JAXRS_WARMUP_DELAY_MS`, `AUDITA_JAXRS_WARMUP_TIMEOUT_MS`.
- `FaultToleranceWarmup` (nuevo): bean `@ApplicationScoped` con `@Priority(400)` que ejecuta durante el startup una invocación dummy de `AuditaCreateService.create()` usando `RequestContextController` para activar programáticamente el contexto `@RequestScoped` requerido. Fuerza la inicialización de los interceptores MicroProfile Fault Tolerance (`@Bulkhead`, `@Timeout`, `@Counted`, `@Timed`) que son lazy por naturaleza. En modo `AUDITA_MODE=http` también invoca `AuditDataHttpRepository.create()` directamente. No bloqueante ante errores: un fallo de warmup registra el aviso pero no impide el arranque. Configurable con `AUDITA_FT_WARMUP_ENABLED`.

- `ReadinessCheck`: integrado con `JaxRsWarmup.isWarmupCompleted()` — el endpoint `/health/ready` devuelve `DOWN` con `jaxrsWarmup=pending` mientras CXF no haya completado su inicialización, evitando que el orquestador (Kubernetes) envíe tráfico antes de que el servicio esté listo.

Lógica de negocio (`application`, `http`, `jpa`)

- `AuditaCreateServiceImpl`, `AuditDataHttpRepository`, `AuditaJpaRepositoryImpl`: añadido cortocircuito (`return "warmup-skipped"` / `return -1L`) cuando `codigoProcesoEvento == "WARMUP_AUDITA_SIDECAR"`, evitando la creación de asientos de auditoría reales durante los warm-ups de arranque.

Incorpora correcciones y mejoras enfocadas en el modo HTTP del servicio (Uso directo de la API de Audita).

- Estabilidad de las inyecciones CDI: se han corregido posibles fallos en la gestión de dependencias de proveedores y mappers que podían afectar al funcionamiento del servicio en modo HTTP.
- Serialización FHIR: el envío de eventos de auditoría ahora utiliza explícitamente el formato `application/fhir+json`, unificando la serialización entre los módulos `http` y `domain` para evitar posibles rechazos por parte del servicio Audita.
- Robustez en la entrada REST: las peticiones con body inválido ahora devuelven una respuesta `400 Bad Request` estructurada en lugar de un error no controlado.
- Trazabilidad de errores: el log del sidecar ahora registra el detalle de los errores respondidos por el servicio Audita, facilitando el diagnóstico de incidencias como, por ejemplo, la omisión de datos obligatorios.
- Optimización del health-check: se ha optimizado la renovación del token Keycloak en los probes de readiness y se ha corregido un problema de caché de configuración que podía impedir la lectura de propiedades críticas tras el arranque.
- Consistencia entre modos: se ha eliminado la capa duplicada de entidades HTTP, consolidando el modelo en `domain` para garantizar un comportamiento coherente independientemente del modo de operación (JPA o HTTP).
- Refuerzo de tests: se han añadido nuevos tests para mejorar las garantías de calidad del componente.

Audit4j-sas (En desuso)

Módulo corporativo de auditoría (Utilizado por aplicaciones antes de la entrada en funcionamiento de Audita)

En desuso

Esta solución se mantiene documentada por motivos de compatibilidad, dado que actualmente sigue siendo utilizada por diversos aplicativos. Sin embargo, no deberá emplearse en nuevos desarrollos, que deberán integrarse mediante la solución corporativa basada en Audita.

Introducción

Con el objetivo de facilitar el cumplimiento de la normativa del SAS en cuanto auditoría se refiere, desde el área de gobernanza de la STIC se ha realizado un componente reutilizable que permita a los desarrolladores incorporar todo el sistema de auditorías en sus aplicaciones de forma rápida y estandarizada, minimizando el tiempo necesario que se requiere para realizar este proceso.

Como punto de partida, y tras analizar las diferentes alternativas ya existentes, se decide partir de una librería de código abierto ya existente que cubre parcialmente las necesidades existentes en la casa, de forma que desde la STIC se ha realizado una ramificación propia de este software para ser adaptado completamente a los requerimientos de auditorías exigidos por la organización.

La librería desarrollada cuanta con las siguientes características:

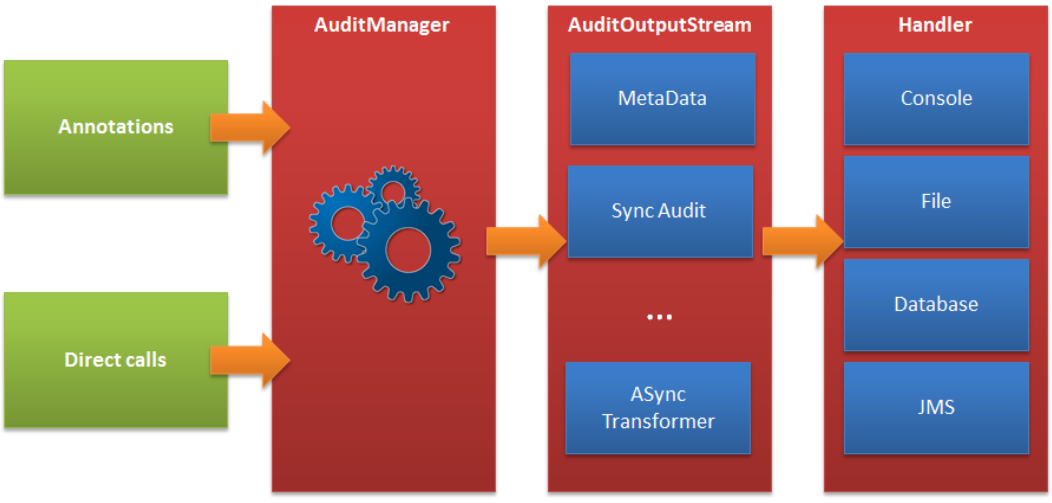
- **Fácilmente usable**: Compatible con cualquier proyecto Java EE 7, siendo suficiente con incluir la librería en el pom.xml e inicializar el motor al arrancar la aplicación.
- **Versatilidad**: Permite varios modos de funcionamiento. Desde llamadas directas al motor de auditoría para registrar cualquier información al uso de anotaciones o ficheros de configuración XML.
- **Síncrono vs asíncrono**: Permite funcionar en modo síncrono y asíncrono, aislando los hilos de ejecución encargados de auditar del resto de la aplicación, no afectando al rendimiento de la aplicación.
- **Alto rendimiento**: El uso de buffers internos permiten recibir gran cantidad de peticiones sin afectar a la estabilidad de la aplicación.
- **Varios modos de salida**: Permite configurar la auditoría para escribir en consola o fichero para trabajos de depuración o base de datos y colas JMS para funcionamientos en producción.
- **Anonimación**: Permite aplicar reglas de anonimación en datos sensibles. Por ejemplo, si se desea almacenar una tarjeta de crédito se podría almacenar la cadena "*********6542"
- **Mínima configuración**: La configuración necesaria para funcionar es mínima.

Histórico de cambios

- **Versión 1.1.1.1** (WIP)
    - **Mejora**: Incorporación del nuevo *handler* `audit4j-sas-audita-handler` , que simplifica la integración del componente Audit4J con Audita sin necesidad de realizar modificaciones en el código existente de la aplicación. Toda la configuración se realiza directamente en el *handler* me diante el fichero `audit4j.conf.yaml` .

    - **Mejora**: Incorporación del nuevo componente `audit4j-sas-audita-jms-consumer` . Este componente permite consumir los mensajes de la cola de mensajería y remitirlos a Audita-Sidecar, posibilitando la integración directa con Audita sin realizar modificaciones sobre el producto.
- **Versión 1.0.2.2**
    - Bug: añadida dependencia que faltaba
- **Versión 1.0.2.1**
    - **Bug**: Corregido error al auditar parámetro de tipo listado/array/colección (Antes sólo auditaba el primer elemento del listado)
    - **Bug**: Corregida consulta a base de datos para comprobar si existe una tabla. Por defecto buscará la tabla auditer en el esquema del usuario. Para buscar otra tabla o otro esquema hay que setear el parametro "table_name" del handle de base de datos y especificar el nombre de la tabla buscada (Indicar [Esquema].[Tabla] si la tabla está en otro esquema).
    - **Mejora**: Añadido java.util.Date como tipo básico, permitiendo auditar tipos Date (Se audita en formato ISO Javascript).
- **Versión 1.0.1.1**
    - **Bug**: Corregido error al auditar entidades con collecciones no iterables.
    - **Mejora**: Optimizado proceso de recorrido de atributos en objetos complejos.
- **Versión 1.0.0.1**: Versión inicial

Flujo de información



- **AuditManager**: Es el punto de entrada al motor de auditoría, es la clase a la que hay que invocar para realizar llamadas directas al motor de auditoría e igualmente es el punto por el que se canalizan todas las auditorías que pueden realizarse con esta librería, como por ejemplo las anotaciones.

- **MetaData**: Esta interfaz debe ser implementada por el la aplicación que utiliza la librería, debiendo implementar sólamente tres funciones las cuales se van a encargar de recuperar la información del usuario que está realizando la acción (actor, perfil y unidad) y una identificación de la aplicación, por ejemplo "MPA". En el caso de aplicaciones donde no hay sesión de usuario (Por ejemplo, REST), en lugar de implementar esta interfaz, se deberá indicar directamente en la configuración de cada evento a auditar de donde debe recuperarse la información de actor, perfil y unidad del operador
- **AuditOutputStream**: Se trata de una interfaz que es implementada por la propia librería varias veces para realizar distintas tareas de construcción/transformación/adaptación sobre los objetos que se están auditando. Por ejemplo, una de esas implementaciones es la que lee de la sesión del usuario el login y el perfil del mismo, y otra de las implementaciones es la que se encarga de procesar las anotaciones y convertirla al formato interno de la librería.
- **Handler**: Los handler son el último paso del proceso, las implementaciones de esta interfaz son las que recoger el evento de auditoría y realiza el almacenamiento. Dependiendo de la implementación el almacenamiento podrá ser consola, fichero, base de datos o colas JMS.

Requerimentos

La librería original ha sido adaptada a la normativa vigente en el SAS, por lo que está diseñada para ser utilizada en proyectos que sean ejecutados bajo el paraguas de **Weblogic 12.1.3**. En el siguiente enlace puede consultarse el detalle de las APIs que ofrece esta versión de Weblogic: http://docs.oracle.com/middleware/1213/wls/NOTES/whatsnew.htm#BGGGHCJD

Instalación

La librería desarrollada se encuentra ubicada en el artifactory corporativo del SAS, estando preparada para ser usada por cualquier usuario que lo desee. Por tanto, para hacer uso de la libería bastará con incorporar librería en pom.xml

```
        <dependency>
        <groupId>es.ja.csalud.sas.componentescomunes.audit4j-extension</groupId>
        <artifactId>audit4j-sas</artifactId>
        <version>1.0.3.1</version>
</dependency>
```

(*) Antes de hacer uso de la librería, se recomienda consultar en artifactory la última versión disponisble (http://calidad.sas.junta-andalucia.es/artifactory/sas-internal/es/ja/csalud/sas/componentescomunes/).

Configuración

Una vez ha sido incorporada la librería en nuestro proyecto, es necesario realizar una pequeña configuración que permita ajustar el funcionamiento del sistema a nuestras necesidades.

El primer paso para configurar nuestro proyecto será crear un archivo YAML llamado "**audit4j.conf.yaml**" en el CLASSPATH del proyecto, el cual será típicamente la ruta "src/main/resources" para un proyecto tipo maven.

```
            !Configuration # Obligatorio

# Listado de Handlers. Pueden ser uno o más.
handlers:
    #- !org.audit4j.core.handler.ConsoleAuditHandler {}
    #- !org.audit4j.core.handler.file.FileAuditHandler {}
    #- !org.audit4j.handler.db.DatabaseAuditHandler{}
    - !org.audit4j.jms.handler.JmsAuditHandler {}

# Configuración de layouts. Sólo necesario para algunos handlers, por ejemplo Console o File
# Dejar SimpleLayout si no se va a usar los handlers anteriormente indicados
layout: !org.audit4j.core.layout.SimpleLayout {}

# Configuración de la clase de la que se recupera información del usuario logado.
# La implementación por defecto DummyMetaData devuele un valores genéricos y constantes
metaData: !org.audit4j.core.DummyMetaData {}

# Comandos especiales
# -metadata: Sincronía, "sync" or "async" (Se recomienda usar "async")
# -annotationTransformer: Clase encargada de procesar las anotaciones
#     + El transformer por defecto auditará todos los parámetros de entradas del método auditado
#     + StrictAnnotationTransformer: Auditará SOLO los parámetros marcados con la anotación @AuditField
#     + ManualAnnotationTransformer: Auditará en función de los parámetros indicados en la anotación
@Audit
#       o bien en lo indicado en los XML de configuración avanzada
commands:
    -metadata=async
    -annotationTransformer=org.audit4j.annotation.transformer.ManualAnnotationTransformer

# Propiedades adicionales
properties:
    # Paquete en el se va a rastrear las anotaciones avanzadas de auditorías (no es necesario si se
utilizan XMLs)
    audit_fields_package: es.ja.csalud.sas.webexample

    # Fichero XML del que se va a leer la información a auditar (no es necesario si se utilizan
Anotaciones)
    audit_fields_xml: auditConfig.xml

    # Número máximo de hilos concurrentes que podrán procesar eventos de auditorías (Valor por defecto
20,
    # poner un valor mayor
    threads_pool_size: 20

    # Nombres de la factoría y cola JMS a utilizar en caso de usar el handler JmsAuditHandler
    jms_factory: ConnectionFactory-ResponsePool
    jms_queue: Queue-ResponsePool
```

Una vez disponemos del módulo configurado, será necesario inicializar el motor al iniciar nuestro servidor. Existen diferentes formas de conseguir que un proceso se ejecute al inicializar un servidor. A modo de ejemplo, en las pruebas realizadas por el área de arquitectura se ha hecho uso de las las anotaciones @Startup y @Singleton:

```
        @Singleton
    @Startup
    public class AuditConfig {
            @PostConstruct
            protected void init() {
                    // Inicialización del motor de auditoría
                    AuditManager.start();
```

```
                // Carga del mapping configurado mediante anotaciones avanzadas o XML
                AuditFieldMapperManager.start();
        }
}
```

Una vez se inicie nuestro servidor, se mostrará por consola un mensaje similar al siguiente:

```
        dic 01, 2016 11:14:03 AM org.audit4j.core.util.Log info
INFORMACIÓN: Audit4j:INFO Audit4j initialized. Total time: 124ms
dic 01, 2016 11:14:03 AM org.audit4j.core.util.Log info
INFORMACIÓN: Audit4j:INFO Loading Audit field mapping
```

Instrucciones de uso

Llamádas directas

El modo más básico de utilizar el motor de auditoría es creando un objeto de tipo AuditEvent e insertarlo en la auditoría. En el siguiente ejemplo puede verse como crear el objeto e invocar al motor de auditoría. Este modo puede utilizarse en cualquier parte de nuestro código:

```
        // Creación del evento
AuditEvent event = new AuditEvent();
event.setObject("Episodio"); // Objeto auditado
event.setAction("Cancelar"); // Acción a realizar

// Listado de campos a auditar
// Los campos con ID "nuhsa", "episodio" y "unidad" se mapearán automáticamente
// con las columnas de mismo nombre en la tabla de adutoría.
// El resto de campos se almacenarán como información adicional
event.addField(new Field("nuhsa", pIn.getNuhsa()));
event.addField(new Field("episodio", "123456"));
event.addField(new Field("unidad", "1111"));
event.addField(new Field("apellido1", pIn.getApellido1()));

// Resultado: 0 Correcto, 1 Incorrecto
event.setResultCode(new Random().nextInt() % 2);

// Datos opcionales. Si no se indican se recuperarán del MetaData configurado
event.setActor("jparriazap");
event.setActorProfile("medico");
event.setActorService("Medicina Interna");
event.setOrigin("MPA");

// Llamada DIRECTA al motor de auditoría
AuditManager.getInstance().audit(event);
```

Anotaciones

La auditoría por anotaciones permiten configurar puntos de auditoría fácilmente sin necesidad de añadir código adicional a nuestros proyectos. Dentro de la auditoría por anotaciones nos encontramos con varios modos de funcionmiento de la librería, siendo el **modo Manual** el que se propone desde el área de gobernanza como el más indicado para las necesidades del SAS.

Es **imporante** destacar que por el propio funcionamiento de las anotaciones y los insterceptores de Java EE, este mecanismo funciona solamente cuando nos encontremos en un contexto CDI.

hay que añadir tambien en el bean.xml el interceptor de auditoria para que funcionen las anotaciones:

```
           <beans xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:schemalocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee
/beans_1_0.xsd">
    <interceptors>
    <class org.audit4j.integration.cdi.AuditInterceptor </class>
    </interceptors>
</beans>
```

Las anotaciones disponibles son las siguientes:

@**Audit**: Es la anotación principal que indica que un método debe ser auditado. Permite las siguientes propiedades:

- **object**: Necesario para especificar el objeto sobre el que se está auditando.
- **action**: Acción que se está realizando sobre el objeto auditado.
- **fields**: Listado de anotaciones de tipo @AuditField, donde se especifican los campos exactos a auditar.

@**AuditField**: Es aplicable tanto a parámetros de un método como a atributos de una clase. Tiene las siguientes propiedades:

- **field**: Nombre que se le desea dar al parámetro o atributo auditado en la tabla de auditoría.
- **fieldPath**: Ruta hasta el parámetro. Por ejemplo, si el tercer párametro de entrada de un método a auditar es de tipo Episodio y este tiene un atributo de tipo Paciente y este a su vez tiene un atributo con el nuhsa, para auditar el nuhsa pondríamos el siguiente path: "[2].episodio.nuhsa"

@**DeIdentify**: Se utiliza para anonimizar parámetros sensibles que desean ser auditados. Tiene las siguientes propiedades:

- **left**: Anonimizar por la izquierda. Ejemplo: @DeIdentify(left=3) -> ***456789
- **right**: Anonimizar por la derecha. Ejemplo: @DeIdentify(right=3) -> 123456***
- **fromLeft**: Anonimizar todos los carácteres a partir de una posición, comenzando a contar por la izquierda. Ejemplo: @DeIdentify(fromLeft=3) -> 123******
- **fromRight**: Anonimizar todos los carácteres a partir de una posición, comenzando a contar por la derecha. Ejemplo: @DeIdentify(fromRight=3) -> ******789

@**IgnoreAudit**: Indica que un parámetro de un método debe ser ignorado. Sólo tiene sentido usar esta anotación en la auditoría por defecto, donde se auditan todos los parámetros de un método menos los marcados con esta anotación.

Default

Modo de funcionar por defecto, es el que aplica si no se indica el comando **annotationTransformer** en el fichero de configuración.

En este modo, se auditan todos los parámetros de entrada de un método a excepción de los marcados como @IgnoreAudit.

```
               public class Paciente {
                private long id;
                private String nuhsa;
                private String nombre;
                @AuditIgnore
                private String apellido1;
                private String apellido2;

                ...
        }

        @Stateless
        public class Pacientes {

                @Audit (
                        object = "Paciente",
                        action = "Buscar"
                )
                public Paciente buscaPacienteDefault(Paciente pacienteIn, @IgnoreAudit int edad, boolean
activo) {
                        ...
                }
        }
```

En este ejemplo, todos los objetos que inyecten el servicio "**Pacientes**" e invoque al método "**buscaPacienteStrict**" generará un registro en la auditoría indicando que se ha actuado sobre el objeto "**Paciente**", realizando la acción "**Buscar** " y se auditarán todos los parámetros de entrada recursivamente (auditando también todos los atributos de los objetos complejos no marcados con @IgnoreAudit) salvo el parametro "**edad**" por estar marcado con la anotación @IgnoreAudit

Strict

El modo estricto audita sólo aquellos parámetros y atributos marcados a conciencia con la anotación @AuditField. En el caso de que uno de los parámetros de entradas no sea de tipo primitivo Java, el motor de auditoría recorrerá recursivamente los atributos internos y auditará sólo aquellos que estén marcados como @AuditField.

Para hacer uso de este módo es necesario indicar el comando **annotationTransformer** con el siguiente valor: **org.audit4j.annotation.transformer.StrictAnnotationTransformer**

```
                public class Paciente {
                private long id;
                @AuditField(field="nuhsa")
                private String nuhsa;
                @AuditField(field="nombre")
                private String nombre;
                private String apellido1;
                private String apellido2;

                ...
        }

        @Stateless
        public class Pacientes {

                @Audit (
                object = "Paciente",
                action = "Buscar"
                )
                public Paciente buscaPacienteStrict(@AuditField(field = "paciente") Paciente pacienteIn,
@AuditField(field = "edad") int edad, boolean activo) {
                        ...
```

```
        }
}
```

En este ejemplo, todos los objetos que inyecten el servicio "**Pacientes**" e invoque al método "**buscaPacienteStrict**" generará un registro en la auditoría indicando que se ha actuado sobre el objeto "**Paciente**", realizando la acción "**Buscar** " y se auditarán los atributos "**paciente.nuhsa**", "**paciente.nombre**" y el segundo parámetro (**edad**).

Manual

El modo manual permite realizar toda la configuración a auditar directamente sobre la anotación padre @Audit o en ficheros de configuración XML. Al igual que en el modo estricto, auditará sólo aquellos parámetros y atributos marcados como auditables.

Para hacer uso de este módo es necesario indicar el comando **annotationTransformer** con el siguiente valor: **org.audit4j.annotation.transformer.ManualAnnotationTransformer**

Desde el área de gobernanza **se aconseja usar este método** para conseguir así un código resultante más limpio.

Dentro de este método nos encontramos dos formas utilizarlo.

Anotación

Permite indicar toda la configuración de la auditoría en la misma anotación @Audit:

```
        public class Paciente {
        private long id;
        private String nuhsa;
        private String nombre;
        private String apellido1;
        private String apellido2;

        ...
}

@Stateless
public class Pacientes {

        @Audit (
                object = "Paciente",
                action = "Buscar",
                fields = {
                        @AuditField(field="nuhsa", fieldPath="[0].nuhsa"),
                        @AuditField(field="id", fieldPath="[0].id"),
                        @AuditField(field="apellido1", fieldPath="[0].apellido1"),
                        @AuditField(field="edad", fieldPath="[1]"),
                        @AuditField(field="apellido2", fieldPath="result.apellido2")
                }
        )
        public Paciente buscaPacienteAnotacion(Paciente pacienteIn, int edad, boolean activo) {
                ...
        }
}
```

En este ejemplo, todos los objetos que inyecten el servicio "**Pacientes**" e invoque al método "**buscaPacienteAnotacion**" generará un registro en la auditoría indicando que se ha actuado sobre el objeto "**Paciente**", realizando la acción "**Buscar** " y se auditarán los atributos "**nuhsa**", "**id**" y "**apellido1**" del primer parámetro de entrada (**pacienteIn**), el segundo parámetro (**edad**) y del objeto devuelto como resultado, el atributo "**apellido2**"

XML

Permite indicar toda la configuración de la auditoría en un fichero XML, centralizando toda la configuración de auditoría de todo el proyecto completo en un sólo fichero de configuración. En este caso, a nivel de método sólo será necesario marcarlo como @Audit y no será necesario especificar más propiedades:

```
                public class Paciente {
                private long id;
                private String nuhsa;
                private String nombre;
                private String apellido1;
                private String apellido2;

                ...
        }

        @Stateless
        public class Pacientes {

                @Audit
                public Paciente buscaPacienteXML(Paciente pacienteIn, int edad, boolean activo) {
                        ...
                }
        }
```

```
        <auditConfiguration>
                <method>...</method>
                <method path="es.ja.csalud.sas.webexample.service.boundary.Pacientes.buscaPacienteXML"
object="Paciente" action="Buscar">
                        <auditField field="nuhsa" fieldPath="[0].nuhsa"/>
                        <auditField field="id" fieldPath="[0].id"/>
                        <auditField field="apellido1" fieldPath="[0].apellido1"/>
                        <auditField field="edad" fieldPath="[1]"/>
                        <auditField field="apellido2" fieldPath="result.apellido2"/>
                </method>
                <method>...</method>
                <method>...</method>
        </auditConfiguration>
```

En este ejemplo, todos los objetos que inyecten el servicio "**Pacientes**" e invoque al método "**buscaPacienteXML**" generará un registro en la auditoría indicando que se ha actuado sobre el objeto "**Paciente**", realizando la acción "**Buscar** " y se auditarán los atributos "**nuhsa**", "**id**" y "**apellido1**" del primer parámetro de entrada (**pacienteIn**), el segundo parámetro (**edad**) y del objeto devuelto como resultado, el atributo "**apellido2**"

Campos especiales

Como puede verse en el apartado **"¿Qué se debe auditar en el SAS?"** de la normativa de auditoría, existen una serie de campos básicos que deben ser auditados, igualmente se reserva un espacio adicional para otros campos no básicos que también deban ser auditados.

Para poder especificar qué campos corresponden con cada uno de estos campos básicos se han definido un patrón de etiquetado que debe ser seguido para el buen funcionamiento del sistema. De esta forma, se definen los siguientes elmentos (No se diferencia entre mayúsculas y minúsculas):

- **OPERADOR**: Login del operador que realiza la acción
- **OPERADOR_PERFIL**: Perfil del operador que realiza la acción
- **OPERADOR_UNIDAD**: Unidad funcional del operador que realiza la acción
- **NUHSA**: Identificación del usuario o paciente
- **EPISODIO**: Episodio sobre el que se está realizando la acción
- **UNIDAD_PACIENTE**: Unidad funcional del episodio sobre el que se realiza la acción.

Mención especial requiere los campos Operador, Operador_Perfil y Operador_Unidad, ya que estos campos son normalmente recuperados de la implementación de la interfaz MetaData propia de cada producto. Pero pueden existir casos en los que la aplicación no tenga sesión (Aplicaciones REST) en donde no sea posible realizar una implementación de MetaData para recuperar datos del contexto. En estos casos esta información deberá ser configurada manualmente en cada evento a auditar.
Por orden de prioridades, el motor de auditoría mirará en primer lugar si se ha configurado un valor específico para un evento, en caso de no existir lo buscará en la implementación de la clase MetaData y en caso de no existir insertará un valor por defecto.

Ejemplo

```
		@Stateless
	public class Pacientes {

		@Audit (
			object = "Paciente",
			action = "Buscar",
			fields = {
				@AuditField(field="operador", fieldPath="[0].login"),
				@AuditField(field="operador_perfil", fieldPath="[0].perfil"),
				@AuditField(field="nuhsa", fieldPath="[1].nuhsa"),
				@AuditField(field="id", fieldPath="[1].id"),
				@AuditField(field="apellido1", fieldPath="[1].apellido1"),
				@AuditField(field="apellido2", fieldPath="[1].apellido2"),
				@AuditField(field="edad", fieldPath="[2]")
			}
		)
		public Paciente buscaPaciente(Usuario user, Paciente pacienteIn, int edad, boolean
activo) {
			...
		}
	}
```

En este ejemplo vemos como se ha configurado para que los campos operador y operador_perfil sean cargados directamente del primer parámetro de la función auditada. En el caso de la unidad del operador, al no estar configurado en el evento, se recuperará de la implementación de MetaData que se haya realizado y en caso de no existir el registro de auditoría se creará con el valor por defecto ("Default User Service")