# Using the Starter‑Audit Module

The **starter‑audit** module is part of our shared library ecosystem. It provides a simple, consistent and non‑intrusive way to record what your application is doing without scattering logging statements through your code. It leverages Spring Boot and Aspect‑Oriented Programming (AOP) so that auditing becomes a cross‑cutting concern—AOP allows us to separate concerns like logging and transaction management from business logic[[1]](https://www.baeldung.com/spring-aspect-oriented-programming-logging#:~:text=Additionally%2C%20Aspect,without%20cluttering%20the%20business%20logic).

This guide explains why you need auditing, how the starter works and how you can add it to your service in a few minutes.

## Purpose & Overview

### Why manual auditing is painful

In most projects developers write log statements at the beginning and end of a method to record who did what and when. Besides cluttering the code, it is easy to forget to add log statements or to format them inconsistently. When services are distributed across micro‑services, correlating events becomes even harder because the same request can flow through multiple services and threads. AOP addresses this problem by letting us “insert” behaviour (such as logging) at predefined points of the application without modifying our core logic[[1]](https://www.baeldung.com/spring-aspect-oriented-programming-logging#:~:text=Additionally%2C%20Aspect,without%20cluttering%20the%20business%20logic).

### What the starter‑audit module does

The starter encapsulates a complete audit framework. Once on the classpath it automatically registers an AuditService, several **enrichers** (to add metadata) and **sinks** (to emit events). You can annotate any Spring component or method with @Audited, or rely on the built‑in HTTP filter, and the aspect will intercept the call, build an AuditEvent and send it to the configured sinks. Each AuditEvent is a structured JSON object containing information about the action (e.g. CREATE, UPDATE, ACCESS), the target entity, the timestamp, the tenant and actor, the outcome, and any contextual metadata. The starter integrates with the **ContextCarrier** from shared‑common and with Micrometer tracing to populate correlation identifiers—the correlation ID is built from the traceId and spanId values in the logging MDC[[2]](https://docs.spring.io/spring-boot/reference/actuator/tracing.html#:~:text=Correlation%20IDs%20provide%20a%20helpful,in%20your%20logs%20by%20default).

### Key benefits

* **Declarative auditing:** annotate methods with @Audited and the aspect will record the call automatically. The HTTP filter records every incoming request as an ACCESS event.
* **Structured JSON logs:** all audit events are emitted in a consistent JSON schema. This makes it easy to ingest them into Elasticsearch, Loki or similar observability platforms.
* **Automatic context propagation:** enrichers add the current trace/correlation ID, tenant ID, user and host details to each event. When Micrometer tracing is enabled, Spring Boot automatically includes correlation IDs built from traceId and spanId in the logging output[[2]](https://docs.spring.io/spring-boot/reference/actuator/tracing.html#:~:text=Correlation%20IDs%20provide%20a%20helpful,in%20your%20logs%20by%20default).
* **Pluggable sinks:** audit events can be sent to a database, Kafka, an outbox table or exported via OTLP. You can also register your own sink.
* **Masking support:** configurable masking hides sensitive fields (for example personally identifiable information) before an event is persisted or sent downstream.
* **Customisation points:** you can control which paths are audited, batch sizes and retry policies, and you can override the TenantProvider to determine the tenant ID.

## Core Dependencies & Auto‑Configuration

### Transitive dependencies

The starter is published as com.lms:starter-audit. It depends transitively on:

| Library | Purpose |
| --- | --- |
| spring-boot-starter-aop | Enables Aspect‑Oriented Programming for method interception. Spring AOP is designed for intercepting method executions[[3]](https://www.baeldung.com/spring-aspect-oriented-programming-logging#:~:text=As%20the%20name%20suggests%2C%20Aspect,without%20modifying%20their%20current%20implementations). |
| shared-common | Provides the ContextCarrier and utilities for working with correlation IDs, headers and metadata. |
| spring-boot-starter-actuator | (Optional) Needed if you want correlation IDs and tracing support via Micrometer; correlation IDs are constructed from traceId and spanId in the MDC[[2]](https://docs.spring.io/spring-boot/reference/actuator/tracing.html#:~:text=Correlation%20IDs%20provide%20a%20helpful,in%20your%20logs%20by%20default). |
| spring-kafka | (Optional) Required when enabling the Kafka sink. |
| spring-boot-starter-jdbc | (Optional) Required when persisting events to a relational database or an outbox table. |

### Audit Auto‑Configuration

The module’s AuditAutoConfiguration class is loaded automatically when shared.audit.enabled is true (default is true). It performs the following tasks:

* **Registers a MaskingStrategy:** uses configuration under shared.audit.masking to mask sensitive fields and JSON paths.
* **Creates an AuditDispatcher:** collects all beans implementing the Sink interface and dispatches events to each sink. Dispatcher behaviour (asynchronous mode, queue size, retry policy) is configured under shared.audit.dispatcher.
* **Provides the AuditService and ReactiveAuditService:** used by aspects, filters and application code to emit audit events.
* **Adds enrichers:** MdcTraceEnricher adds traceId and spanId from the MDC for correlation[[2]](https://docs.spring.io/spring-boot/reference/actuator/tracing.html#:~:text=Correlation%20IDs%20provide%20a%20helpful,in%20your%20logs%20by%20default); HostEnricher adds host/pod/node/region information; TenantEnricher resolves the tenant ID via a TenantProvider; SecurityEnricher extracts the current user from Spring Security; and RequestEnricher adds the request path, HTTP method, client IP and user agent.
* **Configures a dedicated transaction template** for audit persistence when JDBC sinks are enabled.
* **Registers sinks when enabled:**
* **DatabaseSink** – persists events to a table (default public.audit\_events). Fields such as event ID, timestamp, tenant ID, actor, action, resource and the entire event payload (as JSONB) are inserted.
* **KafkaSink** – publishes events to the configured Kafka topic.
* **OutboxSink** – writes events into an outbox table for eventual dispatch via a separate process.
* **OtlpSink** – sends events to an OpenTelemetry collector (stub implementation in this version).
* **Exposes a servlet filter (AuditWebMvcFilter):** if shared.audit.web.enabled is true (default), each HTTP request produces an ACCESS audit event. You can include headers and bodies, and restrict paths using shared.audit.web.includePaths and shared.audit.web.excludePaths.
* **Publishes health and metrics beans:** an AuditHealthIndicator is added to the Spring Boot Actuator and an AuditMetrics bean is registered when a MeterRegistry is available.

## Integration: How to Add the Starter to Your Project

### Step 1 – Add the dependency

Add the starter to your service’s pom.xml (or Gradle build). Use the version defined in your BOM or the latest published version:

<!-- In your service's pom.xml -->  
<dependency>  
 <groupId>com.lms</groupId>  
 <artifactId>starter-audit</artifactId>  
 <version>1.0.0</version>  
</dependency>

The starter brings in spring-boot-starter-aop, so you do not need to declare it explicitly. If you want to intercept calls to non‑public methods (e.g. private or package‑private methods), enable class‑based proxies by adding to your application.yml:

spring:  
 aop:  
 proxy-target-class: true

### Step 2 – Configure the audit module

Auditing is enabled by default. To customise it, add the following properties to your application.yml and adjust as needed:

shared:  
 audit:  
 enabled: true  
 # Include request/response headers and bodies in ACCESS events  
 web:  
 include-headers: false  
 track-bodies: false  
 include-paths:  
 - "/api/\*\*" # only audit API endpoints  
 exclude-paths:  
 - "/actuator/\*\*" # exclude actuator endpoints  
 # Configure how events are dispatched  
 dispatcher:  
 async: true  
 queue-capacity: 50000  
 max-drain-batch: 1000  
 max-retries: 5  
 retry-backoff-ms: 200  
 # Masking sensitive fields by key or JSON path  
 masking:  
 json-paths:  
 - "$.password"  
 - "$.creditCard.number"  
 fields-by-key:  
 - "token"  
 - "ssn"  
 # Enable one or more sinks  
 sinks:  
 db:  
 enabled: true  
 table: audit\_events  
 schema: public  
 batch-size: 200  
 kafka:  
 enabled: false  
 topic: audit.events.v1  
 bootstrap-servers: localhost:9092  
 outbox:  
 enabled: false  
 table: audit\_outbox  
 otlp:  
 enabled: false  
 endpoint: http://otel-collector:4317  
 tenant:  
 header: X-Tenant-Id # header used to resolve tenant id  
 required: true

### Optional dependencies

Depending on your chosen sink you may need to add additional starters:

<!-- Persist events to a relational database -->  
<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-jdbc</artifactId>  
</dependency>  
  
<!-- Send events to Kafka -->  
<dependency>  
 <groupId>org.springframework.kafka</groupId>  
 <artifactId>spring-kafka</artifactId>  
</dependency>  
  
<!-- Enable metrics and tracing; provides correlation IDs by default[2] -->  
<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-actuator</artifactId>  
</dependency>

## Annotating Business Logic with @Audited

The heart of the module is the @Audited annotation. When you annotate a method or class with @Audited, the AuditAspect intercepts the call and emits an AuditEvent. The aspect executes **around** the target method—it records the event *after* the method completes and automatically marks the event as FAILURE if an exception was thrown.

### Basic usage

Annotate the methods you want to audit. The action, entity and message attributes are optional; sensible defaults are used if you omit them.

import com.shared.audit.starter.api.AuditAction;  
import com.shared.audit.starter.api.annotations.Audited;  
import org.springframework.stereotype.Service;  
  
@Service  
public class PaymentService {  
  
 @Audited(action = AuditAction.CREATE, entity = "Payment", entityIdExpr = "#result.id", message = "Process payment")  
 public PaymentResult processPayment(PaymentRequest request) {  
 // Business logic here  
 return paymentProcessor.charge(request);  
 }  
}

When the method returns successfully, an AuditEvent is built with:

* **Action:** set to CREATE (or your chosen AuditAction).
* **Entity type and ID:** Payment and the value returned by the SpEL expression #result.id.
* **Outcome:** SUCCESS for normal completion; FAILURE if the method throws an exception.
* **Message:** the provided message or, in case of failure, the exception message.
* **Timestamp, event ID and tenant:** generated or resolved by enrichers.
* **Metadata:** enriched with host, request, security and trace information.

Here is an example JSON representation of such an event (simplified):

{  
 "schemaVersion": "1.0",  
 "eventId": "c4f9924e-0a2e-4dcf-91a9-0d8e7f38c9f3",  
 "timestamp": "2025-08-22T10:15:30.123Z",  
 "tenantId": "tenant-123",  
 "actor": {"id":"42","username":"jdoe","roles":["USER"],"authType":"JWT"},  
 "action": "CREATE",  
 "entityType": "Payment",  
 "entityId": "PAY-1001",  
 "outcome": "SUCCESS",  
 "sensitivity": "INTERNAL",  
 "dataClass": "NONE",  
 "resource": {"path":"/api/payments","method":"POST","query":""},  
 "metadata": {  
 "traceId": "803B448A0489F84084905D3093480352",  
 "spanId": "3425F23BB2432450",  
 "status": 200,  
 "latencyMs": 45,  
 "clientIp": "192.0.2.10",  
 "host": "payments-pod-1",  
 "region": "eu-west-1"  
 },  
 "message": "Process payment",  
 "payload": {}  
}

### Advanced annotation options

The @Audited annotation exposes several attributes to fine‑tune the generated event:

| Attribute | Type | Description |
| --- | --- | --- |
| action | AuditAction enum | The type of operation (e.g. CREATE, UPDATE, DELETE, ACCESS, OTHER). |
| entity | String | Logical name of the domain entity being affected. |
| entityIdExpr | String (SpEL) | Expression used to extract the entity ID from the method arguments or result. The expression is evaluated against the method arguments and return value. |
| sensitivity | Sensitivity enum | Indicates how sensitive the data is (e.g. PUBLIC, INTERNAL, CONFIDENTIAL, RESTRICTED, PII). Use this to drive downstream retention rules. |
| dataClass | DataClass enum | Classifies the type of data involved (NONE, PII, FINANCIAL, HEALTH, etc.). |
| outcome | AuditOutcome enum | Default outcome to record (defaults to SUCCESS). In case of an exception the aspect will override this with FAILURE. |
| message | String | A human‑readable message describing the operation. |
| captureDiff | boolean | When true, the module will attempt to capture a diff between the old and new state of the entity (not implemented in this version). |

You can also annotate a class—every public method of that class will then be audited unless annotated with @AuditIgnore.

## Advanced Customisation

### Implementing a custom sink

Sometimes you need to send audit events to an external system (e.g. message queue, audit micro‑service or third‑party API). You can implement the Sink interface and register your bean in the Spring context. The AuditDispatcher automatically detects all available sinks:

import com.shared.audit.starter.api.AuditEvent;  
import com.shared.audit.starter.core.dispatch.sinks.Sink;  
import org.springframework.stereotype.Component;  
  
@Component  
public class CustomAuditSink implements Sink {  
 @Override  
 public void send(AuditEvent event) {  
 // Convert the event to your preferred format  
 // and send it to an external service, database or queue  
 externalClient.publish(event);  
 }  
}

### Overriding the tenant provider

To determine the current tenant, the starter uses a TenantProvider. The default implementation returns null. You can provide your own bean that implements TenantProvider to extract the tenant ID from a request header, a security token or any other source:

import com.shared.audit.starter.api.TenantProvider;  
import jakarta.servlet.http.HttpServletRequest;  
import org.springframework.stereotype.Component;  
  
@Component  
public class HeaderTenantProvider implements TenantProvider {  
 private final HttpServletRequest request;  
  
 public HeaderTenantProvider(HttpServletRequest request) {  
 this.request = request;  
 }  
 @Override  
 public String getTenantId() {  
 return request.getHeader("X-Tenant-Id");  
 }  
}

### Customising masking

Sensitive information such as tokens, passwords or credit‑card numbers should not appear in audit logs. Use the shared.audit.masking configuration to specify keys or JSON paths that should be replaced with asterisks. For example:

shared:  
 audit:  
 masking:  
 fields-by-key:  
 - "ssn"  
 - "creditCardNumber"  
 json-paths:  
 - "$.data.card.number"

If you need more control you can implement the MaskingStrategy interface and register your bean. The AuditAutoConfiguration will pick it up instead of the default DefaultMaskingStrategy.

### Configuring correlation IDs

Spring Boot 3 includes **Micrometer Tracing**, which adds traceId and spanId entries to the logging MDC. The audit starter’s MdcTraceEnricher uses these values to populate the correlation ID of every event. The Spring Boot documentation explains that correlation IDs link log lines to traces and are built from traceId and spanId[[2]](https://docs.spring.io/spring-boot/reference/actuator/tracing.html#:~:text=Correlation%20IDs%20provide%20a%20helpful,in%20your%20logs%20by%20default). To customise the format of the correlation ID in your logs, set logging.pattern.correlation in your configuration[[4]](https://docs.spring.io/spring-boot/reference/actuator/tracing.html#:~:text=If%20you%20prefer%20to%20use,used%20by%20Spring%20Cloud%20Sleuth), for example:

logging:  
 pattern:  
 correlation: "[${spring.application.name:},%X{traceId:-},%X{spanId:-}] "  
 include-application-name: false

Ensure that you use the auto‑configured HTTP and WebClient builders (from RestTemplateBuilder or WebClient.Builder) to propagate tracing context across service calls; otherwise the correlation IDs will not flow across services[[5]](https://docs.spring.io/spring-boot/reference/actuator/tracing.html#:~:text=To%20automatically%20propagate%20traces%20over,Builder%20to%20construct%20the%20client).

## Practical Examples

The following examples illustrate how to use the starter-audit in a typical service. You can add them to your **demo‑project** to verify the setup.

### Use case 1 – Auditing a REST controller

import com.shared.audit.starter.api.annotations.Audited;  
import com.shared.audit.starter.api.AuditAction;  
import org.springframework.http.ResponseEntity;  
import org.springframework.web.bind.annotation.\*;  
  
@RestController  
@RequestMapping("/api/orders")  
public class OrderController {  
  
 @PostMapping  
 @Audited(action = AuditAction.CREATE, entity = "Order", message = "Create new order")  
 public ResponseEntity<Order> createOrder(@RequestBody Order order) {  
 // persist order ...  
 return ResponseEntity.ok(order);  
 }  
}

When you send a POST /api/orders request, the AuditWebMvcFilter will produce an ACCESS event for the HTTP call and the @Audited annotation will emit an additional CREATE event for the business operation. The two events share the same trace/correlation ID.

### Use case 2 – Auditing a service method with potential failure

import com.shared.audit.starter.api.annotations.Audited;  
import org.springframework.stereotype.Service;  
  
@Service  
public class UserService {  
  
 @Audited(action = AuditAction.UPDATE, entity = "User", entityIdExpr = "#userId")  
 public User disableUser(String userId) {  
 User user = userRepository.findById(userId)  
 .orElseThrow(() -> new UserNotFoundException(userId));  
 user.setActive(false);  
 return userRepository.save(user);  
 }  
}

If userRepository.findById throws a UserNotFoundException, the aspect sets the outcome to FAILURE and records the exception message. Successful execution produces a SUCCESS outcome.

## Troubleshooting & Common Pitfalls

**No audit events are emitted:**

* Verify that shared.audit.enabled is true. The auto‑configuration uses this property to decide whether to register beans.
* Only **public methods** of Spring beans are proxied by default. Calls inside the same class are considered self‑invocation and bypass the proxy. To work around this, extract the logic into another bean or enable proxy exposure and call the method via AopContext.currentProxy().
* Ensure that your packages are scanned by Spring (e.g. your services live under a package seen by @SpringBootApplication).

**Correlation ID is missing or null:**

* Enable Micrometer tracing by adding spring-boot-starter-actuator and any tracer implementation you need. Spring Boot includes correlation IDs in logs based on traceId and spanId[[2]](https://docs.spring.io/spring-boot/reference/actuator/tracing.html#:~:text=Correlation%20IDs%20provide%20a%20helpful,in%20your%20logs%20by%20default).
* Use the auto‑configured HTTP clients (RestTemplateBuilder, RestClient.Builder or WebClient.Builder) so that trace context is propagated across service calls[[5]](https://docs.spring.io/spring-boot/reference/actuator/tracing.html#:~:text=To%20automatically%20propagate%20traces%20over,Builder%20to%20construct%20the%20client).
* If you have your own filter that clears or overwrites the MDC, ensure it preserves traceId and spanId.

**Events are not persisted to the database:**

* Make sure you have added spring-boot-starter-jdbc and configured a datasource.
* Set shared.audit.sinks.db.enabled: true and adjust the table and schema names if necessary.
* The default schema file (db/migration/V1\_\_create\_audit\_tables.sql) in the starter can be used to create the necessary tables.

**High throughput causes back‑pressure:**

* Increase shared.audit.dispatcher.queue-capacity and max-drain-batch.
* Consider enabling the asynchronous dispatcher to avoid blocking the calling thread; tune max-retries and retry-backoff-ms to suit your environment.

## Summary

The **starter‑audit** module decouples audit logging from your business logic by using Spring’s AOP facilities, emitting structured JSON events and enriching them with contextual metadata. It supports multiple sinks out of the box and can be customised to meet your organisation’s security and observability requirements. With a few lines of configuration and a couple of annotations, you can start capturing comprehensive audit trails across all your services.

[[1]](https://www.baeldung.com/spring-aspect-oriented-programming-logging#:~:text=Additionally%2C%20Aspect,without%20cluttering%20the%20business%20logic) [[3]](https://www.baeldung.com/spring-aspect-oriented-programming-logging#:~:text=As%20the%20name%20suggests%2C%20Aspect,without%20modifying%20their%20current%20implementations) Logging With AOP in Spring | Baeldung

<https://www.baeldung.com/spring-aspect-oriented-programming-logging>

[[2]](https://docs.spring.io/spring-boot/reference/actuator/tracing.html#:~:text=Correlation%20IDs%20provide%20a%20helpful,in%20your%20logs%20by%20default) [[4]](https://docs.spring.io/spring-boot/reference/actuator/tracing.html#:~:text=If%20you%20prefer%20to%20use,used%20by%20Spring%20Cloud%20Sleuth) [[5]](https://docs.spring.io/spring-boot/reference/actuator/tracing.html#:~:text=To%20automatically%20propagate%20traces%20over,Builder%20to%20construct%20the%20client) Tracing :: Spring Boot

<https://docs.spring.io/spring-boot/reference/actuator/tracing.html>