# Using Other Shared Starters

This guide explains how to integrate the remaining *shared* Spring Boot starters in the same manner as the starter‑audit module. Each starter provides ready‑made auto‑configuration classes, sensible defaults and extension points. By following the patterns below you can integrate these modules into a service in under five minutes.

## General Integration Pattern

All starters in the shared library follow a similar pattern:

1. **Add the dependency** – declare the module in your pom.xml alongside the version managed by your BOM. Example:

<!-- Add the desired starter -->  
<dependency>  
 <groupId>com.yourcompany.shared</groupId>  
 <artifactId>starter-xyz</artifactId>  
</dependency>

1. **Let auto‑configuration do its work** – the starters are annotated with @AutoConfiguration and guarded by conditions such as @ConditionalOnClass or @ConditionalOnProperty. Spring Boot will eagerly register beans when the corresponding classes are on the classpath and the starter is enabled.
2. **Adjust properties** – each starter exposes a configuration properties class under the shared.\* prefix. The defaults usually enable the feature with sane values; adjust these in your service’s application.yml as needed.
3. **(Optional) Override beans** – if you need custom behavior (e.g. publishing events to a database rather than logging), define a bean of the same type in your application. Spring Boot will back off and use your bean instead of the default.

Below you will find module‑specific details.

## starter‑actuator

### Purpose & overview

The *actuator* starter bundles [Spring Boot Actuator](https://docs.spring.io/spring-boot/reference/actuator/endpoints.html) with sensible defaults for Kubernetes‑hosted services. It exposes common endpoints (/actuator/health, /actuator/info, /actuator/metrics, /actuator/prometheus, /actuator/loggers, /actuator/threaddump) and adds:

* **Common metric tags** – automatically tags every Micrometer meter with application name, environment, region and zone, enabling easier drill‑down across environments. Spring Boot’s documentation notes that “common tags are generally used for dimensional drill‑down on the operating environment, such as host, instance, region, stack, and others”[[1]](https://docs.spring.io/spring-boot/reference/actuator/metrics.html#:~:text=Common%20Tags).
* **HTTP exchange recording** – optionally stores recent HTTP requests so that /actuator/http-exchanges can be queried.
* **Security chain** – optionally adds a dedicated SecurityFilterChain for /actuator/\*\* endpoints.
* **Custom /actuator/whoami endpoint** – returns process, host and environment information.

### Key dependencies & auto‑configuration

The starter brings in spring-boot-starter-actuator and configures itself via SharedActuatorAutoConfiguration and SharedActuatorSecurityAutoConfiguration. Beans are registered only when the required classes are on the classpath. Properties live under shared.actuator with sensible defaults (see [actuator-defaults.properties](shared-lib/shared-lib/shared-starters/starter-actuator/src/main/resources/com/shared/actuator/starter/actuator-defaults.properties)).

### Integration steps

1. **Add the dependency**:

<dependency>  
 <groupId>com.yourcompany.shared</groupId>  
 <artifactId>starter-actuator</artifactId>  
</dependency>

1. **Adjust exposure** – by default, Spring Boot only exposes /actuator/health over HTTP[[2]](https://docs.spring.io/spring-boot/reference/actuator/endpoints.html#:~:text=Exposing%20Endpoints). The starter’s defaults expose additional endpoints via management.endpoints.web.exposure.include property. You can override it in application.yml to include or exclude endpoints:

management:  
 endpoints:  
 web:  
 exposure:  
 include: "health,info,metrics,prometheus,loggers,threaddump"

1. **Enable HTTP exchange recording** (optional):

shared:  
 actuator:  
 http-exchanges:  
 enabled: true  
 capacity: 1000 # number of requests to keep

1. **Enable actuator security** (optional) – to protect actuator endpoints with basic auth, set shared.actuator.security.enabled=true. You can control whether /actuator/prometheus requires authentication via shared.actuator.security.permit-prometheus.

### Usage

Once enabled, you can navigate to /actuator to view the available endpoints. The /actuator/whoami endpoint returns the process ID, uptime and deployment metadata (region, zone, pod, node). To add custom info, expose BuildProperties or GitProperties beans – the starter’s SharedInfoContributor will include them.

### Advanced customisation

You can define your own MeterRegistryCustomizer bean to add additional tags or filter metrics. To secure actuator endpoints differently, define your own SecurityFilterChain named actuatorSecurityFilterChain; the starter will back off.

### Troubleshooting

* **Endpoints not exposed** – ensure management.endpoints.web.exposure.include lists the endpoints you need and that your service isn’t behind an external gateway blocking /actuator.
* **Duplicate tags** – if you register your own MeterRegistryCustomizer, avoid re‑adding tags already set by CommonTagsCustomizer.

## starter‑core

### Purpose & overview

The *core* starter provides foundational infrastructure used by other modules. It solves common cross‑cutting concerns:

* **Context propagation** – ContextFilter and TraceContextHolder propagate correlation IDs and custom headers across threads and frameworks. The filter reads headers (correlation, tenant, user) and populates the MDC and shared context.
* **Correlation ID logging** – LoggingAutoConfiguration sets up a CorrelationIdFilter to ensure every log message contains a traceId and optional tenant/user IDs. Spring Boot documentation explains that correlation IDs are built from the tracing traceId and spanId and let you link log entries to distributed traces[[3]](https://docs.spring.io/spring-boot/reference/actuator/tracing.html#:~:text=Correlation%20IDs%20provide%20a%20helpful,in%20your%20logs%20by%20default).
* **Tenant support** – TenantContextHolder and related classes support multi‑tenant applications and provide annotations (@RequireTenant, @OptionalTenant) to enforce presence of a tenant ID.
* **Global exception handling** – GlobalExceptionHandler produces structured error responses.

### Key dependencies & auto‑configuration

The starter brings in spring-boot-starter-web, spring-boot-starter-aop and common modules. CoreAutoConfiguration registers filters and beans if servlet classes are present. Properties live under shared.core:

| Property (prefix shared.core) | Description |
| --- | --- |
| context.correlation-header | header name for correlation IDs |
| context.generate-if-missing | whether to generate a new ID if missing |
| tenant.enabled | enable tenant checks |
| tenant.default-include-global | include global records by default |

### Integration steps

1. Add the dependency:

<dependency>  
 <groupId>com.yourcompany.shared</groupId>  
 <artifactId>starter-core</artifactId>  
</dependency>

1. *(Optional)* customise correlation and tenant headers in application.yml:

shared:  
 core:  
 context:  
 correlation-header: X-Correlation-Id  
 generate-if-missing: true  
 tenant:  
 enabled: true  
 default-include-global: false

### Usage

The core starter works transparently: each incoming HTTP request is assigned a traceId if one isn’t supplied. You can retrieve the current ID via TraceContextHolder.getTraceId() or the tenant via TenantContextHolder.getTenantId(). Annotate REST controllers or services with @RequireTenant to reject calls lacking a tenant ID, or with @OptionalTenant to allow anonymous access.

### Advanced customisation

You can implement your own TenantResolver and register it as a bean to determine tenants from JWTs or other sources. To customise the log format, set logging.pattern.correlation to include application name and trace IDs[[4]](https://docs.spring.io/spring-boot/reference/actuator/tracing.html#:~:text=If%20you%20prefer%20to%20use,used%20by%20Spring%20Cloud%20Sleuth). For asynchronous correlation across threads, use the provided TraceContextHolder.wrap(Runnable) helper.

### Troubleshooting

If correlation IDs are missing from logs, ensure spring-boot-starter-aop is on the classpath and that CorrelationIdFilter has not been overridden. For tenant issues, verify that your gateway or upstream service is passing X-Tenant-Id (or the configured header) and that shared.core.tenant.enabled=true.

## starter‑crypto

### Purpose & overview

This starter offers a unified symmetric encryption/HMAC service built on top of the shared CryptoService. It tackles the challenge of securely storing secrets, signing payloads and rotating keys. Features include:

* **Algorithm selection** – uses AES‑GCM by default; future algorithms can be plugged in via shared.crypto.algorithm.
* **In‑memory key provider** – loads base64‑encoded keys from configuration for local development. Active key rotation is supported by changing shared.crypto.in-memory.activeKid.
* **Automatic metrics and health** – exposes gauges (active key status) and a crypto health indicator via Micrometer.
* **Correlation filter** – optionally propagates correlation, tenant and user IDs via CryptoMdcFilter so that encryption logs include tracing information.

### Key dependencies & auto‑configuration

CryptoAutoConfiguration runs when CryptoService classes are present and shared.crypto.enabled is true (default). It registers a CryptoService bean using a CryptoAlgorithm and KeyProvider. InMemoryKeyProviderAutoConfiguration decodes base64 keys from shared.crypto.in-memory.keys and provides a KeyProvider bean. You can define your own key provider (e.g. AWS KMS or Vault) and disable the default provider by excluding the bean or setting shared.crypto.key-provider.

The properties are extensive; the most used ones include:

| Key | Purpose |
| --- | --- |
| shared.crypto.enabled | master switch (default true) |
| shared.crypto.algorithm | AES\_GCM (default) or future algorithms |
| shared.crypto.in-memory.activeKid | active key identifier |
| shared.crypto.in-memory.keys | map of key IDs to base64 secrets |
| shared.crypto.log-active-kid | log active KID on startup |

### Integration steps

1. Add the dependency:

<dependency>  
 <groupId>com.yourcompany.shared</groupId>  
 <artifactId>starter-crypto</artifactId>  
</dependency>

1. Configure keys in application.yml (example with two 256‑bit keys):

shared:  
 crypto:  
 algorithm: AES\_GCM  
 in-memory:  
 activeKid: k1  
 keys:  
 k1: BvlzsNKBmJNWQvQxeSu9Hde3gsq3VJbmFz1g7w==  
 k2: e8cxYkKNSu9dKFhl3B+NhOQmvWq2XwUO9nrAbg==

1. Inject and use the CryptoService in your code:

@Autowired  
CryptoService crypto;  
  
byte[] encrypted = crypto.encrypt("mySecret".getBytes(StandardCharsets.UTF\_8));  
byte[] decrypted = crypto.decrypt(encrypted);

1. (Optional) Sign and verify payloads via the provided HmacSigner:

String signature = hmacSigner.signBase64("ignored", dataString);  
boolean ok = hmacSigner.verify("ignored", dataString.getBytes(), Base64.getDecoder().decode(signature));

### Advanced customisation

To use an external key management system, disable the in‑memory provider and define your own KeyProvider and CryptoAlgorithm beans. For example, implement a provider backed by AWS KMS and inject it in an @Configuration class. To send metrics to your observability backend, ensure io.micrometer:micrometer-core is on the classpath; the starter will automatically publish gauges.

### Troubleshooting

* **Invalid KID** – ensure shared.crypto.in-memory.activeKid matches one of the keys defined under keys.
* **Unsupported algorithm** – currently only AES\_GCM is supported; choosing another value will throw an exception.

## starter‑data

### Purpose & overview

The *data* starter supplies JPA base classes, pagination helpers and multi‑tenant filtering. It simplifies building data access layers by centralising best practices:

* **Base entities** – BaseEntity defines id, createdAt, updatedAt and optimistic locking via @Version. SoftDeleteEntity adds a deleted boolean and transparently filters out soft‑deleted rows.
* **Tenant support** – TenantBaseEntity adds a tenantId column and integrates with Hibernate filters. Aspect‑oriented code applies a filter around all @Transactional methods to automatically restrict queries to the current tenant. Annotations @TenantOnly and @IncludeGlobal let you override the default behaviour per method or type.
* **Pagination DTOs** – PageResponse and PageResponses convert Page<T> results into API‑friendly responses.

### Key dependencies & auto‑configuration

StarterDataTenantAutoConfiguration runs when jakarta.persistence.EntityManagerFactory and Aspect are present. It registers a TenantHibernateFilterAspect which:

* Looks up the current tenant from the ContextManager (populated by starter-core).
* Enables a Hibernate filter (tenantFilter) with parameters tenantId and allowGlobal before each transactional method.
* Disables the filter afterwards.

Configuration properties under shared.data.tenant-filter control the filter name and default behaviour.

### Integration steps

1. Add the dependency:

<dependency>  
 <groupId>com.yourcompany.shared</groupId>  
 <artifactId>starter-data</artifactId>  
</dependency>

1. For multi‑tenant support, annotate your JPA entities:

@FilterDef(name = "tenantFilter",  
 parameters = { @ParamDef(name = "tenantId", type = StringType.class),  
 @ParamDef(name = "allowGlobal", type = BooleanType.class) })  
@Filter(name = "tenantFilter",  
 condition = "tenant\_id = :tenantId OR (:allowGlobal = true AND tenant\_id IS NULL)")  
public abstract class TenantBaseEntity { /\* ... \*/ }

In your service methods, annotate with @TenantOnly to force tenant filtering, or @IncludeGlobal to allow global rows.

1. Adjust tenant filter properties if needed:

shared:  
 data:  
 tenant-filter:  
 enabled: true  
 filter-name: tenantFilter  
 default-include-global: false

1. Use the PageResponses.of(Page<T>, Function<T,R>) helper to convert Page results to response DTOs.

### Troubleshooting

* **Filter not applied** – ensure your service methods are annotated with @Transactional; the aspect only wraps transactional methods. Also verify that the entity defines the Hibernate filter with the names specified in properties.
* **Tenant missing** – make sure starter-core is on the classpath and that the current tenant is set in the request context.

## starter‑headers

### Purpose & overview

This module centralises HTTP header propagation and security best practices. It addresses common issues when building microservices:

* **Correlation and request IDs** – ensures every incoming request has a correlation ID; automatically generates one when missing and adds it to the MDC and outgoing calls.
* **Header propagation** – interceptors for RestTemplate, Spring Cloud OpenFeign and WebClient propagate configured headers (correlation ID, request ID, tenant ID, user ID) to downstream services. Without such interceptors, distributed tracing headers can easily be lost.
* **Security headers** – adds HTTP security headers (HSTS, X‑Frame‑Options, X‑Content‑Type‑Options, Referrer‑Policy, Permissions‑Policy, Content‑Security‑Policy, COOP/COEP, XSS‑Protection) to every response. This reduces the risk of clickjacking, MIME sniffing and cross‑site scripting.
* **Forwarded header support** – when behind a proxy (e.g. ingress), enables ForwardedHeaderFilter to correctly populate X‑Forwarded-\* information.

### Key dependencies & auto‑configuration

SharedHeadersAutoConfiguration is active when servlet filters and RestTemplate classes are available and shared.headers.enabled is true (default). It registers:

* CorrelationHeaderFilter – generates/propagates correlation and request IDs.
* SecurityHeadersFilter – adds security headers; can be disabled via shared.headers.security.enabled=false.
* ForwardedHeaderFilter – enabled via shared.headers.forwarded.enabled=true.
* RestTemplateCustomizer, RequestInterceptor (Feign) and WebClientHeaderCustomizer – propagate headers on outgoing calls.

Properties include:

| Prefix shared.headers | Purpose |
| --- | --- |
| enabled | master switch |
| generate-if-missing | create IDs when absent |
| correlation.header | name of correlation header |
| propagation.enabled | propagate headers |
| propagation.include | list of headers to propagate |
| security.enabled | enable security headers |
| security.exclude-paths | paths where headers are not added |

### Integration steps

1. Add the dependency:

<dependency>  
 <groupId>com.yourcompany.shared</groupId>  
 <artifactId>starter-headers</artifactId>  
</dependency>

1. Tune the properties:

shared:  
 headers:  
 generate-if-missing: true  
 propagation:  
 include: [X-Correlation-Id, X-Request-Id, X-Tenant-Id, X-User-Id]  
 security:  
 enabled: true  
 exclude-paths: ["/public/\*\*"]

1. To propagate headers in Feign clients, simply use Spring Cloud OpenFeign (no extra code). For RestTemplate or WebClient, the starter customizers will automatically add interceptors.

### Troubleshooting

* **Headers not propagated** – ensure shared.headers.propagation.enabled is true. If using custom RestTemplateCustomizer or WebClient.Builder, call customizers.customize() or register the provided WebClientHeaderCustomizer bean.
* **Security headers missing on some paths** – verify that the path isn’t excluded via security.exclude-paths.

## starter‑kafka

### Purpose & overview

This starter packages Kafka producer/consumer configuration with idempotency support and observability. Key features include:

* **JSON serialization** – configures ObjectMapper with Java Time module and registers JsonSerializer/JsonDeserializer for values.
* **Exactly‑once publishing** – when shared.kafka.exactly-once=true, producers are configured with idempotence and transactions; a transactional ID prefix is derived from the application name.
* **Concurrent consumers** – provides a ConcurrentKafkaListenerContainerFactory with exponential backoff retry and dead‑letter topic (DLT) publishing.
* **Topic management** – KafkaTopicConfig creates topics defined in properties on application startup.
* **Idempotent listeners** – annotation @IdempotentKafkaListener ensures a Kafka listener processes each message only once by storing message IDs in Redis.

### Key dependencies & auto‑configuration

KafkaAutoConfiguration registers a KafkaAdmin and an ObjectMapper. Additional configuration classes such as KafkaProducerConfig, KafkaConsumerConfig, KafkaObservabilityConfig and KafkaTopicConfig are imported conditionally. Properties (prefix shared.kafka) include:

| Property | Description |
| --- | --- |
| bootstrap-servers | list of Kafka brokers (required) |
| group-id | default consumer group |
| concurrency | number of consumer threads |
| max-poll-records | maximum records per poll |
| exactly-once | enable idempotent producers |
| auto-offset-reset | latest or earliest |
| topics | map of topic names to partition/replication settings |

### Integration steps

1. Add the dependency:

<dependency>  
 <groupId>com.yourcompany.shared</groupId>  
 <artifactId>starter-kafka</artifactId>  
</dependency>

1. Configure brokers and basic settings:

shared:  
 kafka:  
 bootstrap-servers: broker1:9092,broker2:9092  
 group-id: my-service  
 concurrency: 3  
 topics:  
 order-events:  
 partitions: 3  
 replication-factor: 1  
 order-events.dlt:  
 partitions: 3  
 replication-factor: 1

1. Produce messages using the injected KafkaTemplate<String,Object>:

kafkaTemplate.send("order-events", orderId, new OrderEvent(...));

1. Consume messages with idempotency:

@IdempotentKafkaListener(topics = "order-events", groupId = "my-service")  
public void handleOrder(EventEnvelope<OrderEvent> envelope) {  
 // business logic  
}

The annotation stores processed message IDs (from the envelope header) in the configured idempotency store (default is Redis).

### Advanced customisation

* **Override consumer factory** – define your own ConsumerFactory or ConcurrentKafkaListenerContainerFactory bean to customise error handling or concurrency.
* **Custom idempotency store** – implement IdempotencyStore and mark your bean as @Primary. For example, use a relational database instead of Redis.
* **Observability** – ensure io.micrometer:micrometer-observation is on the classpath; the starter will record Kafka send/receive observations automatically.

### Troubleshooting

* **Missing topics** – topics defined under shared.kafka.topics are created by the starter only if the application has permission to create topics. Alternatively, create them manually.
* **Duplicate message processing** – ensure @IdempotentKafkaListener is used and that the message envelope contains a unique ID; verify that Redis is configured correctly.

## starter‑mapstruct

### Purpose & overview

This starter centralises MapStruct configuration and provides helper mappers. MapStruct generates type‑safe mappers at compile time; however, without consistent settings each mapper may behave differently. The starter defines a @MapperConfig with common options:

* **Spring component model** – generated mappers are @Component and injectable.
* **Constructor injection** – nested mappers are injected via constructors.
* **Adders for collections** – when mapping collections, MapStruct prefers calling addX methods (useful for JPA aggregates).
* **Null‑handling** – always perform null checks, ignore nulls on update methods and map null lists/maps to empty collections.
* **Strictness** – unmapped target properties produce errors while unmapped source properties trigger warnings.
* **Disable Lombok builders** – by default, MapStruct uses setters rather than Lombok builder methods.

### Integration steps

1. Add the dependency:

<dependency>  
 <groupId>com.yourcompany.shared</groupId>  
 <artifactId>starter-mapstruct</artifactId>  
</dependency>

1. Annotate your mappers with @Mapper(config = SharedMapstructConfig.class) to inherit the default configuration:

@Mapper(config = SharedMapstructConfig.class)  
public interface CustomerMapper {  
 CustomerDto toDto(Customer entity);  
 void update(@MappingTarget Customer entity, CustomerDto dto);  
}

1. Use CommonMappers for frequently needed conversions (UUID ↔ String, trimming, date/time conversions). For example:

@Mapping(target="id", source="id", qualifiedByName="uuidToString")  
String id;

1. Map paginated results using PageMapper:

PageDto<OrderDto> pageDto = pageMapper.toDto(orderPage, orderMapper::toDto);

### Troubleshooting

If MapStruct fails to generate a mapper, check the compiler logs for unmapped properties. Since the starter sets unmappedTargetPolicy = ReportingPolicy.ERROR, missing mappings will cause compilation to fail.

## starter‑money‑time

### Purpose & overview

This module integrates JavaMoney, time utilities and validation into your service. It solves typical internationalisation issues when dealing with monetary amounts and timestamps:

* **Default currency and formatting** – defines a CurrencyUnit bean, a MonetaryAmountFormat and registers a Jackson module so that monetary amounts are serialised/deserialised in ISO format.
* **Time service** – provides a TimeService that knows the configured time zone, working days and office hours; useful for scheduling and business‑day calculations.
* **Formatters** – registers Formatters for MonetaryAmount, Instant and Duration so they can be used as MVC request parameters.
* **Validation annotations** – adds @CurrencyAllowed, @MoneyMin, @MoneyMax and @PhoneE164 for domain validation.
* **JPA converters** – includes a MonetaryAmountStringAttributeConverter and MoneyEmbeddable for persisting monetary amounts as strings.

### Key dependencies & auto‑configuration

SharedMoneyTimeAutoConfiguration registers beans when javax.money.CurrencyUnit or Jackson classes are available. Properties live under shared.money and shared.time:

| Property (prefix shared.money) | Description |
| --- | --- |
| default-currency | ISO 4217 code (default USD) |
| locale | locale tag used for formatting (default en-US) |
| rounding-mode | rounding strategy (e.g. HALF\_EVEN) |
| format | formatting style (AMOUNT\_CURRENCY, etc.) |

| Property (prefix shared.time) | Description |
| --- | --- |
| zone | IANA time zone (default UTC) |
| business-days | comma‑separated list of working days |
| work-start, work-end | working day start/end (HH:mm) |

### Integration steps

1. Add the dependency:

<dependency>  
 <groupId>com.yourcompany.shared</groupId>  
 <artifactId>starter-money-time</artifactId>  
</dependency>

1. Configure currency and time zone if necessary:

shared:  
 money:  
 default-currency: SAR  
 locale: ar-SA  
 time:  
 zone: Asia/Riyadh  
 business-days: MONDAY,TUESDAY,WEDNESDAY,THURSDAY,FRIDAY  
 work-start: "08:00"  
 work-end: "16:00"

1. Inject TimeService to determine if a given Instant is within business hours:

boolean open = timeService.isWithinBusinessHours(Instant.now());

1. Use validation annotations on your DTOs:

public record PaymentRequest(  
 @CurrencyAllowed({"USD","SAR"}) MonetaryAmount amount,  
 @PhoneE164 String phoneNumber) {}

### Troubleshooting

Ensure you have javax.money:money-api and a JSR 354 implementation on your classpath (e.g. org.javamoney:moneta) – the starter only configures beans around these types; it does not provide the implementation itself.

## starter‑openapi

### Purpose & overview

The *openapi* starter configures [SpringDoc OpenAPI](https://springdoc.org/) for documenting your REST API. It automatically registers an OpenAPI bean and, optionally, a GroupedOpenApi group with a custom name and path pattern. You can also enable JWT bearer security in the generated specification.

### Integration steps

1. Add the dependency:

<dependency>  
 <groupId>com.yourcompany.shared</groupId>  
 <artifactId>starter-openapi</artifactId>  
</dependency>

1. Configure your API metadata and grouping:

shared:  
 openapi:  
 title: Order Service API  
 version: v2  
 description: REST API for managing orders  
 servers:  
 - https://api.example.com  
 jwt-security: true  
 group:  
 enabled: true  
 name: order  
 paths: ["/api/orders/\*\*"]  
 packages-to-scan: ["com.example.orders.web"]

1. Start your application and browse to /swagger-ui.html or /swagger-ui to view the interactive documentation. The openAPI bean will include your metadata and, if jwt-security is true, will define a bearer authentication scheme.

### Troubleshooting

* **Missing documentation** – ensure you have the springdoc-openapi-starter-webmvc-ui dependency; the shared starter only configures the OpenAPI beans.
* **Endpoints not included** – verify the path patterns under shared.openapi.group.paths and the packages under packages-to-scan.

## starter‑ratelimit

### Purpose & overview

This module adds a simple token‑bucket rate limiter using Redis. It protects HTTP endpoints by counting requests per key (tenant/IP/user) per minute and returning HTTP 429 when the limit is exceeded.

### Integration steps

1. Add the dependency **and ensure Redis is available**:

<dependency>  
 <groupId>com.yourcompany.shared</groupId>  
 <artifactId>starter-ratelimit</artifactId>  
</dependency>

1. Configure Redis connection via starter-redis (see below).
2. Configure rate limit properties:

shared:  
 ratelimit:  
 capacity: 60 # max requests per minute  
 refill-per-minute: 60  
 key-strategy: tenant # tenant | ip | user

1. The RateLimitFilter is registered for /api/\* paths by default and adds X-RateLimit-Limit and X-RateLimit-Remaining headers. If the request count exceeds capacity, a 429 response is returned.

### Troubleshooting

If the filter appears to have no effect, ensure that a StringRedisTemplate bean is available (provided by starter-redis) and that the application has network access to Redis. Also verify that your endpoints are under /api/; adjust FilterRegistrationBean if needed.

## starter‑redis

### Purpose & overview

This starter configures Redis connectivity, caching and convenience services. It solves the challenges of consistent serialization, key prefixing and distributed synchronization.

Key features:

* **Connection factory & templates** – configures a LettuceConnectionFactory (supporting SSL, timeouts and client names) and provides StringRedisTemplate, RedisTemplate<String,Object> and optional ReactiveRedisTemplate.
* **Customizable key prefixing** – KeyPrefixStrategy prepends a namespace to all keys, preventing collisions across services.
* **Cache manager** – registers a RedisCacheManager that uses JSON serialization for values and allows per‑cache TTLs.
* **Distributed primitives** – includes services for distributed locks (RedisLockService), idempotency (RedisIdempotencyService), rate limiting (RedisRateLimiter) and pub/sub (RedisPubSubPublisher, RedisPubSubListener).

### Key dependencies & properties

Properties under shared.redis include:

| Property | Description |
| --- | --- |
| url | complete Redis URI (overrides host/port/password) |
| host/port/password | connection details when url is not set |
| database | DB index (default 0) |
| ssl | whether to use SSL |
| client-name | optional client name |
| timeout | command timeout |
| key-prefix | prefix applied to all keys (default shared) |
| default-ttl | default cache TTL |
| caches | map of cache names to TTL/key prefix overrides |

### Integration steps

1. Add the dependency:

<dependency>  
 <groupId>com.yourcompany.shared</groupId>  
 <artifactId>starter-redis</artifactId>  
</dependency>

1. Configure connection properties:

shared:  
 redis:  
 url: redis://:password@redis.example.com:6379/0  
 key-prefix: myservice  
 default-ttl: 30m  
 caches:  
 users:  
 ttl: 10m  
 products:  
 ttl: 60m

1. Inject and use Redis primitives:

@Autowired RedisLockService lockService;  
boolean locked = lockService.tryLock("order-123", Duration.ofSeconds(5));

The RedisIdempotencyService can be used to enforce idempotent request handling; RedisRateLimiter offers an alternative to the servlet filter used in starter-ratelimit.

### Troubleshooting

* **Connection errors** – verify the URI and network connectivity. When using SSL, ensure ssl=true or the URI scheme rediss:// is used.
* **Cache not working** – check that spring.cache.type=redis (if you use Spring’s cache abstraction) and that your beans are annotated with @Cacheable.

## starter‑resilience

### Purpose & overview

This lightweight starter configures Reactor Netty timeouts for Spring WebFlux WebClient. It prevents indefinite hangs by setting connection and response timeouts.

### Integration steps

1. Add the dependency:

<dependency>  
 <groupId>com.yourcompany.shared</groupId>  
 <artifactId>starter-resilience</artifactId>  
</dependency>

1. Configure timeouts:

shared:  
 resilience:  
 http-timeout-ms: 5000  
 connect-timeout-ms: 2000

1. Inject WebClient.Builder – the starter registers a bean only if none exists. The builder will use Reactor Netty with the configured timeouts:

@Autowired WebClient.Builder webClientBuilder;  
WebClient client = webClientBuilder.baseUrl("https://api.example.com").build();

### Troubleshooting

If you define your own WebClient.Builder bean, the starter backs off. Ensure you either import starter-resilience alone or merge your custom builder with the configured timeouts.

## starter‑security

### Purpose & overview

The security starter turns your application into a Resource Server with JWT authentication out of the box. It simplifies configuration by providing sensible defaults and helpers:

* **JWT decoder** – supports HS256, jwks or issuer modes. In HS256 mode, you provide a shared secret. In jwks or issuer mode, the decoder fetches keys from a JWKS URI or OIDC issuer.
* **Authority mapping** – maps roles and scopes from configurable claims to Spring Security GrantedAuthoritys with configurable prefixes.
* **JSON error responses** – JsonAuthEntryPoint and JsonAccessDeniedHandler return structured 401/403 responses rather than HTML.
* **Tenant propagation** – JwtTenantFilter extracts a tenant ID from a claim and places it into the ContextManager for multi‑tenant awareness.

### Key properties (prefix shared.security)

| Property | Description |
| --- | --- |
| mode | hs256 (default), jwks, issuer |
| hs256.secret | base64 or plain secret for HMAC |
| jwks.uri | JWKS endpoint when mode=jwks |
| issuer | issuer URL when mode=issuer; also used to validate issuer |
| audience | optional expected audience |
| roles-claim | claim containing role names (default roles) |
| scope-claim | claim containing scopes (default scope) |
| tenant-claim | claim containing tenant ID |
| authority-prefix | prefix for scopes (default SCOPE\_) |
| role-prefix | prefix for roles (default ROLE\_) |
| resource-server.permit-all | additional public endpoints besides /actuator/health and Swagger |
| resource-server.disable-csrf | disable CSRF (default true) |
| resource-server.stateless | stateless sessions (default true) |

### Integration steps

1. Add the dependency:

<dependency>  
 <groupId>com.yourcompany.shared</groupId>  
 <artifactId>starter-security</artifactId>  
</dependency>

1. Configure your JWT decoding strategy:

shared:  
 security:  
 mode: hs256  
 hs256:  
 secret: super-secret-key  
 roles-claim: realm\_access.roles  
 scope-claim: scope  
 tenant-claim: tenant  
 resource-server:  
 permit-all:  
 - /api/public/\*\*

1. Secure your endpoints by default; only the paths listed in permit-all are anonymous. For user roles, annotate controller methods with @PreAuthorize("hasRole('ADMIN')") or use method-level security.

### Advanced customisation

* **Custom JWT decoder** – define a JwtDecoder bean to override all decoding logic (for example, to use RSA keys). The starter will back off.
* **Multiple security chains** – define additional SecurityFilterChain beans if you need separate access rules for different URL patterns (e.g. /actuator/\*\* via starter-actuator).
* **Custom error responses** – override JsonAuthEntryPoint or JsonAccessDeniedHandler with your own beans.

### Troubleshooting

* **401 errors** – ensure that your JWT is signed with the configured key and that the audience and issuer claims match. Check that the Authorization: Bearer <token> header is present.
* **Roles not applied** – verify that roles-claim points to the correct claim in your JWT and that roles are not nested in an unexpected structure.

## starter‑validation

### Purpose & overview

The validation starter collects a set of useful Hibernate Validator constraints and registers them automatically. It includes:

* @CurrencyCode – validates that a string is a valid ISO 4217 currency code.
* @PhoneE164 – validates that a phone number follows the E.164 international format.
* @Trimmed – trims leading and trailing whitespace during validation.
* @MoneyMin / @MoneyMax – validate monetary amounts against minimum/maximum values.

### Integration steps

1. Add the dependency:

<dependency>  
 <groupId>com.yourcompany.shared</groupId>  
 <artifactId>starter-validation</artifactId>  
</dependency>

1. Use the annotations on your DTOs. For example:

public record PaymentDto(  
 @Trimmed  
 @NotBlank  
 String customerName,  
  
 @PhoneE164  
 String phone,  
  
 @MoneyMin(value = "10.00", currency = "USD")  
 @MoneyMax(value = "1000.00", currency = "USD")  
 MonetaryAmount amount  
) {}

### Troubleshooting

If constraints seem to have no effect, ensure that the starter is on the classpath **and** that Spring’s validation infrastructure is active (e.g. annotate your controller methods with @Valid). Also verify that you are using the same javax.money.MonetaryAmount implementation as the validation classes (e.g. JSR 354 with Moneta).

## Conclusion

The shared starters encapsulate best practices and infrastructure concerns, allowing you to focus on business logic. By adding the appropriate starter dependencies, leveraging auto‑configuration and adjusting a handful of properties, you can enable features such as metrics, encryption, multi‑tenancy, header propagation, Kafka messaging, MapStruct mapping, money/time handling, OpenAPI docs, rate limiting, Redis primitives, resilience, security and validation in minutes. Consult the starter sources for more details or extend the provided beans when necessary.

[[1]](https://docs.spring.io/spring-boot/reference/actuator/metrics.html#:~:text=Common%20Tags) Metrics :: Spring Boot

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

[[2]](https://docs.spring.io/spring-boot/reference/actuator/endpoints.html#:~:text=Exposing%20Endpoints) Endpoints :: Spring Boot

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

[[3]](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) Tracing :: Spring Boot

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