ns-secure

Developer usage guide — eight schema-driven engines behind one facade

v0.2.0-beta.2 | Java 8+ | Apache-2.0 | io.github.nsprism | 8 engines

1 · Installing

One dependency brings the facade plus all eight engines. Published to Maven Central — no repository config needed.

Gradle

```
dependencies {
    // umbrella: ns-secure-core + log + id + crypto engines
    implementation 'io.github.nsprism:ns-secure:0.2.0-beta.2'
}
```

Maven

```
<dependency>
    <groupId>io.github.nsprism</groupId>
    <artifactId>ns-secure</artifactId>
    <version>0.2.0-beta.2</version>
</dependency>
```

Or just one engine

Artifact	Gives you	Pulls in
`io.github.nsprism:ns-secure`	Everything	all modules below
`io.github.nsprism:ns-secure-log`	`NS.use(LOG,…)`, `NS.log(…)`	core + logback-core
`io.github.nsprism:ns-secure-id`	`NS.use(GENERATE,…)`	core only

`io.github.nsprism:ns-secure-crypto`	`NS.use(CRYPTO / VERIFY,…)`	core + keys
`io.github.nsprism:ns-secure-validation`	`NS.use(VALIDATE,…)`	core only
`io.github.nsprism:ns-secure-sanitize`	`NS.use(SANITIZE,…)`	core only
`io.github.nsprism:ns-secure-flag`	`NS.use(FLAG,…)`	core only
`io.github.nsprism:ns-secure-keys`	key store by key id (secrets, RSA/EC pairs, rotation)	core only
`io.github.nsprism:ns-secure-token`	`NS.use(TOKEN,…)` HS/RS/ES	core + keys
`io.github.nsprism:ns-secure-common`	`NS.use(COMMON,…)` — 189 utilities	core only
`io.github.nsprism:ns-secure-core`	Facade + SPIs only — **no engines**; calls fail with `NSServiceNotFoundException` until an engine jar arrives	nothing

> Engines are discovered by `java.util.ServiceLoader` — the jar on the classpath *is* the configuration. No SLF4J, no logback-classic; logback-core is the only third-party dependency.

---

2 · The 30-second mental model



- **Schemas are typed JSON** — `"type"` is a log level (`INFO` …), `GENERATE` , a crypto direction (`ENCRYPT` / `DECRYPT` / `SEAL` / `OPEN`), or `VALIDATE` / `SANITIZE` / `FLAG` / `TOKEN` / `COMMON` .
- **Load once, keep in a static final** — `NSSchema.fromResource("/x.json")` or `fromJson(str)` ; validation happens at load, not per call.
- **Calls never mutate your map** — every op returns a new map.
- `NS.use` **returns** `<T> T` — assign to a typed variable; to chain, write `NS.<Map<String,Object>>use(…).get("x")` .
- **Failures fail fast and generically** — config errors at startup, crypto errors as one generic `NSCryptoException` .

---

3 · Log engine — `ns-secure-log`

Deny-by-default masking. A field with no rule is **not logged** — it simply never appears.

Minimal use

```
import io.github.nsprism.nssecure.core.*;
import java.util.Map;

private static final NSSchema LOGIN = NSSchema.fromResource("/schemas/auth-login.json");

Map<String,Object> safe = NS.use(NSOp.LOG, LOGIN, Map.of(
    "userId", "u-88412",
    "email",  "jane@acme.com",
    "password", "hunter2"));          // no rule ? suppressed entirely
// ? {"userId":"#a9f2…","email":"j•••e@acme.com"}  ? one JSON line to the sink

NS.log("payment completed for order %s", orderId);           // plain message
NSLogContext.with("traceId", traceId);                       // merged into every line
```

Schema — `auth-login.json`

```
{
  "name": "auth.login", "type": "INFO",
  "message": "User signed in",
  "rules": {
    "userId":         "PSEUDONYM_RULE",
    "email":          "EMAIL_RULE",
    "user.ip":        "IP_RULE",
    "payment.card":   "CARD_RULE",
    "sessionToken":   "CREDENTIAL_RULE",
    "method":         "ALLOW_RULE",
    "requestBody":    "ALLOW_RULE",     // passes, but content floor still scrubs cards/SSNs/tok
```

```
    "notes":          "ALLOW_RAW",      // bypasses the content floor — opt-in, flagged in prod
    "items[*].id":    "ALLOW_RULE"      // list paths work too
  }
}
```

Built-in rules

Rule	Does
`REDACT_RULE`	? `[REDACTED]`
`EMAIL_RULE` / `PARTIAL`	`j•••e@acme.com` (pattern-checked)
`CARD_RULE` / `SSN_RULE` / `PHONE_RULE` / `DOB_RULE`	last-4 / redact when the value actually matches (card is Luhn-checked)
`IP_RULE`	format-preserving mask
`CREDENTIAL_RULE`	JWT/API-key-looking strings ? redacted
`PSEUDONYM_RULE`	stable `#hash` — join logs without the real id
`ALLOW_RULE`	log the field; still goes through the content floor
`ALLOW_RAW`	log verbatim — opt-in only; refused in `ns.secure.env=prod` unless explicitly allowed
`REG_EXP:<pattern>`	your own match ? redact

Engine settings — classpath `/ns-secure.json` (all keys optional)

```
{
  "logging": {
    "service": "claims-api",
    "level": "INFO",
    "unlisted": "SUPPRESS",
    "contentRules": ["CREDENTIAL_RULE","CARD_RULE","SSN_RULE"],
    "maskChar": "•",
    "contextKeys": ["traceId","spanId","tenant"],
    "levelPolicy": {
      "DEBUG": { "forceStrategy": "REDACT" },
      "ERROR": { "unlisted": "REDACT" }
    }
  },
  "pseudonym": { "secretRef": "env:NS_SECURE_PEPPER" }
}
```

Outputs — classpath `/ns-secure-logback.xml`

Defaults to one JSON line on stdout. Drop this file on the classpath for real outputs — **logback-core only**, works on 1.2?1.5:

```
<configuration>
  <appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">
    <encoder class="io.github.nsprism.nssecure.log.logback.NSJsonEncoder"/>
  </appender>
  <appender name="FILE" class="ch.qos.logback.core.rolling.RollingFileAppender">
    <file>logs/app.json</file>
    <rollingPolicy class="ch.qos.logback.core.rolling.SizeAndTimeBasedRollingPolicy">
      <fileNamePattern>logs/app.%d.%i.json.gz</fileNamePattern>
      <maxFileSize>10MB</maxFileSize><maxHistory>14</maxHistory>
    </rollingPolicy>
    <encoder class="io.github.nsprism.nssecure.log.logback.NSJsonEncoder"/>
  </appender>
  <appender name="TCP" class="io.github.nsprism.nssecure.log.logback.NSTcpAppender">
    <remoteHost>log-collector</remoteHost><port>5044</port>
  </appender>
  <appender name="SPLUNK" class="io.github.nsprism.nssecure.log.logback.NSSplunkHecAppender">
    <url>${SPLUNK_HEC_URL}</url><token>${SPLUNK_HEC_TOKEN}</token>
  </appender>
  <root level="INFO">
    <appender-ref ref="CONSOLE"/><appender-ref ref="FILE"/>
    <appender-ref ref="TCP"/><appender-ref ref="SPLUNK"/>
  </root>
</configuration>
```

4 · Id engine — `ns-secure-id`

A `"type": "GENERATE"` schema fills **empty** fields (missing / null / `""`). Existing values are never touched — idempotent and safe to re-apply.

```
private static final NSSchema ORDER = NSSchema.fromResource("/schemas/order-create.json");

Map<String,Object> out = NS.use(NSOp.GENERATE, ORDER, order);
// orderId, publicRef, otp… filled; fields already set are kept

NS.use(NSOp.GENERATE, ORDER, order,
       (field, v) -> repo.exists(field, v));   // collision ? retries a fresh value
```

Schema

```
{
  "name": "order.create", "type": "GENERATE",
  "rules": {
    "orderId":       "ULID",
    "publicRef":     "UUID4",
    "correlationId": "UUID7",
    "otp":           "NUMERIC:6",
    "apiKey":        { "type": "RANDOM", "alphabet": "base62", "bits": 256, "prefix": "sk_live_"
    "licenseKey":    { "type": "PATTERN", "regex": "[A-HJ-NP-Z2-9]{4}-[A-HJ-NP-Z2-9]{4}" },
    "lines[*].lineId": "ULID",
    "snow":          { "type": "SNOWFLAKE" }
  }
}
```

Id types

Type	Shape / notes
`ULID`	26-char Crockford-32, time-ordered; monotonic within the same millisecond
`UUID4` / `UUID7`	random / time-ordered UUID
`SNOWFLAKE`	64-bit; needs one unique nodeId per instance: `-Dns.secure.id.nodeId=7` or `NS_SECURE_ID_NODE_ID`
`NUMERIC:<n>`	digits — 6-digit OTPs etc.

`RANDOM:<len>[:<prefix>]` or object	`alphabet` (base58/62/64url/digits/hex/crockford32/`custom:<chars>`), `bits` or `length`, `prefix`, `case`, `"groups": "4-4-4"`
`PATTERN` / `REG_EXP:<re>`	generates a value matching a (bounded) regex
Presets	`API_KEY` · `PUBLIC_ID` · `SORT_KEY` · `HUMAN_CODE` · `OTP`

> At schema compile the engine warns when a key-like field has under 64 bits of keyspace — collision/guessing risk surfaces before production, not after.

5 · Crypto engine — `ns-secure-crypto`

Direction comes from the schema's `"type"`. Keys live in `keyRef` references — never in code or schemas.

```
private static final NSSchema W = NSSchema.fromResource("/schemas/customer-encrypt.json"); // ty
private static final NSSchema R = NSSchema.fromResource("/schemas/customer-decrypt.json"); // ty

Map<String,Object> stored = NS.use(NSOp.CRYPTO, W, row);
// {"ssn":"<base64 envelope>","email":"h1.Xyz…","password":"$ns-pbkdf2-…","orderId":"ORD-1"}

Map<String,Object> back = NS.use(NSOp.CRYPTO, R, stored);
// ssn decrypts; h1. token and pbkdf2 hash stay as-is (irreversible)

boolean ok = NS.use(NSOp.VERIFY, "hunter2", stored.get("password"));   // constant-time
```

Schema — `customer-encrypt.json` (same `name` as its decrypt twin!)

```
{
  "name": "customer", "type": "ENCRYPT",
  "keyRef": "env:NS_SECURE_KEY",            // default key for rules without their own
  "strict": true,                            // unlisted fields ? NSCryptoException (names field
  "rules": {
    "ssn":      { "algo": "AES-GCM" },
    "payload":  { "algo": "RSA-OAEP", "keyRef": "vault:payments-pub" },
    "email":    { "algo": "HMAC-SHA256", "searchable": true, "keyRef": "env:NS_HMAC_KEY" },
    "password": { "algo": "PBKDF2", "iterations": 600000 },
    "consentId":{ "algo": "PLAIN" },
    "notes":    { "algo": "DROP" }
  }
}
```

Algorithms

Algo	Kind	Output / notes
`AES-GCM`	reversible	Base64 envelope *version · alg · keyId · nonce(12B) · ct+tag*; fresh nonce per call; default choice
`ChaCha20-Poly1305`	reversible	AEAD; Java 11+ only (clear error on 8–10)

`RSA-OAEP`	reversible	Hybrid: fresh AES key per message, RSA-wrapped; =2048-bit keys; use `SEAL` / `OPEN` schemas
`AES-CBC`	reversible	Legacy interop — *requires* `"legacy": true`
`HMAC-SHA256`	one-way	`h1.<token>` blind index with `"searchable": true`; normalised input ? equality lookups without decrypting
`PBKDF2`	one-way	`$ns-pbkdf2-sha256$i=<cost>$<salt>$<hash>`; VERIFY re-derives from the stored string
`PLAIN` / `DROP`	—	pass through / remove the field

Keys

```
keyRef:  env:NS_SECURE_KEY          // Base64 / PEM in env var
         file:/secure/keys/card.aes // raw key file
         file:/ks.p12#alias         // PKCS12 alias
         ns-keys:card-key           // shared key store — same store as tokens, cached
         vault:… / kms:…            // your scheme ? NSKeys.register(scheme, resolver)

NSKeys.generate(NSKeyType.AES_256);        // new keys
NSKeys.generateRsaPem(3072);               // RSA pair ? PEM strings
```

Context-bound: ciphertext is AEAD-bound to `<schema name>/<field>` — moving it to another field/schema fails to decrypt. So the encrypt/decrypt **schema names must match**. HMAC is path-based, so an index token survives across schemas. **Rotation:** keys are cached (default 60 s TTL, refresh-ahead, stale-if-error) — a replaced `file:` / `vault:` key goes live within the TTL; `NSKeyStore.invalidate(keyId)` makes it immediate, and an outage keeps serving the last good key. Use `keyRef: "ns-keys:<keyId>"` for the built-in keyring: encrypt always uses the current key, decrypt picks the right current/previous key from the envelope's own key fingerprint, so old ciphertext stays readable across rotation. Plain `env:` / `file:` keyRefs have no ring — keep versioned keyRefs (`card-key-v1` read / `card-key-v2` write). `env:` keys can't change inside a running JVM.

6 · Validation — `ns-secure-validation`

Six stages: shape & limits ? types & formats ? required/defaults ? conditions ? cross-field & tenant overrides ? external checks. Fails closed; error messages never contain the rejected value.

```
private static final NSSchema CREATE = NSSchema.fromResource("/schemas/customer-create.json");
Map<String,Object> r = NS.use(NSOp.VALIDATE, CREATE, body);
if (Boolean.FALSE.equals(r.get("valid"))) { /* r.get("errors") = [{field,code,id?,message,params
// on success r.get("data") holds coerced values (types applied)

NSValidators.register("unique-username",
        (value, ctx) -> !users.exists((String) value),
        Duration.ofMillis(300));            // per-check timeout — a hang fails closed
NSRequestContext.put("tenant", tenantId);   // read by "$ctx.*" conditions
executor.execute(NSRequestContext.wrap(task));  // carries context into workers, restores after
```

Schema — `customer-create.json`

```
{
  "name": "customer.create", "type": "VALIDATE", "strict": true,
  "rules": {
    "email":   { "type": "EMAIL", "required": true, "message": "Enter a valid email address" },
    "age":     { "type": "INT", "min": 18, "message": { "MIN": "You must be at least {min}" } },
    "zip":     { "type": "PATTERN", "regex": "\\d{5}", "when": "country == 'US'" },
    "card":    "CARD",            // Luhn-checked
    "country": "COUNTRY",
    "lines[*].qty": { "type": "INT", "min": 1 }
  },
  "overrides": [ { "when": "$ctx.tenant == 'acme'", "rules": { "age": { "min": 21 } } } ]
}
```

Notes

- **23 types**: STRING, TEXT, INT, LONG, DECIMAL, BOOL, DATE, INSTANT, EMAIL, URL, UUID, ULID, PHONE, CARD, IBAN, COUNTRY, CURRENCY, LOCALE, IP, HOSTNAME, PATTERN, ENUM, LIST/MAP shapes.
- **Messages** support placeholders (`{min}` , `{max}` …) — `{value}` is refused at load.
- **Regexes** go through a safe-regex check (nested quantifiers / unbounded repeats refused) and run under a time budget (`"limits": {"regexBudgetMs": …}`); a timeout fails the field and is counted — opt into a distinct `PATTERN_TIMEOUT` code with `"onRegexTimeout"` , default stays `PATTERN` .

- **External checks** run on a bounded worker pool with per-check timeouts (default 1 s) and a total budget (default 3 s); timeout, error or a saturated pool all fail closed as `CHECK_ERROR`. A check that needs the request thread registers with `Duration.ZERO`. See `NSValidators.totalBudget/executor/stats`.
- **Clocks are injectable** (`new NSDefaultValidationEngine(clock)`) for relative-date rules like `MIN_DATE:now+30d`.
- Run **VALIDATE before CRYPTO**: reject bad input at the edge, then encrypt what you keep.

---

7 · Sanitize — `ns-secure-sanitize`

Encode once, at the output edge. 11 destination contexts plus pattern rules; the report names fields, never values.

```
{ "name": "customer.export", "type": "SANITIZE",
  "rules": {
    "name":  { "context": "CSV_CELL", "maxLength": 120 },
    "bio":   "TEXT_ONLY",
    "phone": { "pattern": { "keep": "[0-9+]" } },
    "html":  { "context": "HTML" }
  } }
Map<String,Object> r = NS.use(NSOp.SANITIZE, EXPORT, row);
// {data, changed, dropped, unsanitized, rejected} — counts + field names only
```

Context	For
`HTML` / `HTML_ATTRIBUTE`	page text / attribute values
`JAVASCRIPT` / `JSON`	inside script and JSON literals
`URL` / `CSS`	link values / style values
`SQL`	prefer bind params; escapes when you can't
`CSV_CELL`	neutralises `= + - @` formula injection, quotes as needed
`FILENAME` / `SHELL` / `LOG_LINE` / `TEXT_ONLY`	safe names / no shell metachars / no control chars / strip all markup

---

8 · Feature flags — `ns-secure-flag`

Kill switch, deterministic sticky rollout, fail-closed default — and guards for whole methods.

```
{ "key": "new-checkout", "on": true, "default": false, "rollout": 25,
  "messages": { "ROLLOUT": "New checkout is rolling out gradually." } }
```

```
NSDecision d = NS.use(NSOp.FLAG, NEW_CHECKOUT, userId);   // value() + message()
Report r = NS.use(NSOp.FLAG, NEW_CHECKOUT, userId,
                  () -> service.run(req));               // blocked ? NSFeatureDisabledExceptior
NS.use(NSOp.FLAG, NEW_CHECKOUT, userId, NSFlag.THROW);    // guard line — code below only runs v
NSFlagBindings.verify(flags, controllers);                // @NSFlag annotations checked at star
```

- Rollout bucketing is deterministic: the same unit id lands on the same side of the percentage every call, on every node.
- Flags are read per call — flip `"on"` or the rollout % and the next call sees it (kill switch). The swap is per instance — distribute the schema as config and restart, or push it to every pod.

9 · Keys + tokens — `ns-secure-keys` · `ns-secure-token`

The token type pins algorithm, key id, issuer, audience and TTL — the token's own header is never trusted, so `alg:none` and RS256?HS256 confusion can't happen.

Key store — `/ns-keys.json`

```
{ "keys": {
    "access-signing-key": { "secret": "env:NS_ACCESS_KEY",
                            "previous": ["env:NS_ACCESS_KEY_OLD"] },   // rotate: verify old, si
    "api-rsa":            { "private": "file:/keys/api.pem", "public": "derived" } },
  "cache": { "ttl": "60s", "refreshAhead": "15s", "maxStale": "1h" } }
// resolved keys are cached — refreshed in the background before expiry,
// and an outage keeps serving the last good key up to maxStale
// NSKeyStore.invalidate(keyId) / invalidateAll() / cacheStats() for ops
```

Token type — `access-token.json`

```
{ "key": "access-token", "type": "TOKEN",
  "alg": "HS256", "keyId": "access-signing-key",
  "issuer": "ns-secure", "audience": "orders-api",
  "ttl": "15m", "maxTtl": "1h", "clockSkew": "30s",
  "claims": { "sub": { "required": true }, "tenant": { "required": true }, "roles": {} } }
```

```
String token = NS.use(NSOp.TOKEN, ACCESS, claims);    // issue
NSDecision d = NS.use(NSOp.TOKEN, ACCESS, bearer);    // verify — never throws
if (!d.allowed()) { /* d.code(): BAD_SIGNATURE | MALFORMED | EXPIRED | ISSUER | AUDIENCE
                                | MISSING_CLAIM | KEY_UNAVAILABLE | NOT_YET_VALID */ }
String sub = d.claims().get("sub");
```

- Algorithms: HS256/384/512, RS256/384/512, ES256/384 — keys resolve through the cached key store (TTL + refresh-ahead; a source outage serves the last good key).
- Signature is checked on the raw segments **before** any JSON is parsed (nesting depth capped at 64) — a malformed unsigned token is a `BAD_SIGNATURE`, never a parser error; claims are whitelisted by the type.
- A misconfigured type (unknown alg, missing keyId, ttl > maxTtl) fails at startup with every problem listed.
- Rotation: add the new secret, move the old ref to `previous` — old tokens keep verifying until you retire it.

10 · Common toolbox — `ns-secure-common`

189 bounded, null-safe utilities behind one door — the Core + Standard tiers of the common catalogue.

```
private static final NSSchema CUT =
    NSSchema.fromJson("{\"op\":\"strings.truncate\",\"max\":200}");
String shortText = NS.use(NSOp.COMMON, CUT, comment);          // settings in schema ? value in

long days = NS.use(NSOp.COMMON,
    NSSchema.fromJson("{\"op\":\"dates.businessDaysBetween\"}"),
    Map.of("a", start, "b", end));                             // several params ? Map by name
```

Category	Examples
`strings.*` (23)	truncate/abbreviate (code-point safe), case conversion, splitWords, mask, maskEmail, slugify, constant-time equals
`collections.*` (10)	safeGet, slice (clamped), distinctBy, mostCommon, topN, zip
`dates.*` (14)	parseIsoDate, ageOn, business days, duration parse/format, ago, expiresAt/isExpired
`numbers.*` (16)	strict parse*, saturating arithmetic, half-up/even rounding, bytes, median/percentile, minor units, splitAmount
`maps.*` (15)	getPath a.b[0].c, typed getters, deepMerge, flatten/unflatten, diff, redactKeys, query strings
`objects.*` (7)	isEmptyValue, isTruthy, canonical/canonicalJson, redactedToString, describe (type+size, never content)
`encoding.*` (17)	strict Base64(+url)/hex/percent, HTML/XML/JSON/CSV escaping, CSV parse, strict UTF-8, sha256Hex
`checks.*` (22)	require* preconditions — nonBlank, inRange, maxLength, matches (safe regex), email, uuid, httpsUrl, keys/onlyKeys
`io.*` (22)	bounded reads/copies/walks, resolveInside (path escapes refused), atomic writes, owner-only temp files, safeFileName, sha256File, zip-slip/zip-bomb guards, detectMime
`concurrency.*` (18)	retry + backoff (full jitter), withTimeout, rateLimiter, circuitBreaker, bulkhead, bounded pools, parallelMapBounded, lazy/expiring/boundedCache, deadline, swallowing
`reflection.*` (9)	guarded class loading, annotatedFields, shortStack/exceptionSummary (class names only — never messages); private access off unless `-Dns.secure.common.reflection=true`
`random.*` (10)	SecureRandom everywhere — secureInt/Long/bytes/hex/token (=16B), randomString/digits, shuffled, generatePassword
`env.*` (6)	env/envOr/envInt (range)/envBool/envDuration/envSecret (`Secret[***]`)

- **Binding:** schema keys set parameters by name (checked at load, "did you mean" on typos); whatever's open comes from data — one open param takes data itself, several take a Map keyed by name.
- A `Clock` parameter is injected — construct `new NSDefaultCommonEngine(clock)` for deterministic tests.
- Errors name the parameter and the rule — never the value. Exceptions from helpers surface as themselves.

11 · Changing things at runtime

Swap a schema or re-register an engine — the **next** call uses it; a broken replacement is rejected while the old one keeps working.

```
// schemas behind an AtomicReference — hot-swap rules
static final AtomicReference<NSSchema> ORDER =
    new AtomicReference<>(NSSchema.fromResource("/schemas/order.json"));
ORDER.set(NSSchema.fromJson(newJson));                    // throws on bad edit ? old stays

// engine settings — level floor, maskChar, outputs
NSServiceRegistry.register(NSLogService.class,
    new NSDefaultLogService(NSLogSettings.fromJson(newCfg),
                            NSLogbackSink.fromXml(newXml)));

// existing NS.log loggers and schemas pick the new service up immediately
```

> The other engines swap the same way: replace the VALIDATE/SANITIZE/FLAG/COMMON schema behind your `AtomicReference` ; flip a flag's `"on"` or rollout at the next call; rotate token keys by moving the old ref into `previous` in the key store — verify keeps working while issue switches.

12 · Metrics & request context

Install one SPI and every engine reports — engine, schema, outcome, timing — with low-cardinality, value-free tags (never a value, user id, tenant or key). Default is a no-op, and a throwing backend can never break an engine call.

```
NSMetrics.install(new NSCountingMetrics());     // or your Micrometer/OTel adapter
// every NS.use        → operation(engine, schema, outcome, nanos)
// background signals  → event: keys.stale / keys.refresh-failed / validate.check.timeout /
//                       validate.regex.timeout / sanitize.regex.timeout / log.dropped
```

```
// request context across worker threads — the caller's context is installed for the
// task and the worker thread is restored/cleared afterwards
executor.execute(NSRequestContext.wrap(task));            // Runnable / Callable / Executor
Supplier<T> s = NSRequestContext.wrapSupplier(task);
try (NSRequestContext.Applied a = NSRequestContext.capture().apply()) { … }
// Spring: taskDecorator(NSRequestContext::wrap)
```

> Counters to alert on: `keys.stale` (key source failing — last good key in use), `validate.check.timeout` (slow dependency), `*.regex.timeout` (pattern budget exceeded), `log.dropped` (appender queue full).

---

13 · Compatibility & performance

Checked	Result
Java	bytecode is Java 8; tested every release 8 ? 27
Build tools	Maven, Gradle 8 & 9, plain javac, shaded fat jars (ServiceLoader entries merge correctly), JPMS module path, Spring Boot 3.x
logback-core	1.2 / 1.3 / 1.4 / 1.5 — falls back to stdout JSON if absent
Masked log event	~ 5–8 µs; suppressed DEBUG call < 11 ns
ULID	~ 50–80 ns; AES-GCM encrypt/decrypt ~ 4–7 µs
PBKDF2 @600k	~ 0.2 s by design (~ 1 s on Java 8); RSA-OAEP open ~ 6–8 ms
Under load	800k concurrent ULIDs all unique; 1.5M-op soak, flat heap; 200 live schema swaps @ 8 threads, zero errors

---

14 · Gotchas

- **Empty schema list ? log everything** — unlisted fields are suppressed; list what you *want* to keep.
- **Encrypt/decrypt schema names must match** — ciphertext is bound to `<name>/<field>`.
- `env:` **keys don't rotate in-process** — use `file:` / `vault:` where live rotation matters.
- **ChaCha20-Poly1305 needs Java 11+** — clear failure on 8–10.
- **Missing engine jar** ? `NSServiceNotFoundException` at the call, never silent.
- **A replaced** `file:` / `vault:` **key is not instant** — the cache picks it up within the TTL (60 s default); call `NSKeyStore.invalidate(keyId)` for an immediate switch, or `"cache": {"ttl": "0s"}` to read the source on every use.
- **Unsigned token + broken JSON =** `BAD_SIGNATURE`, not `MALFORMED` — the signature is verified over the raw segments before any JSON is parsed.
- **External checks run on a worker pool**, not the request thread — if a check needs the request thread's own state, register it with `Duration.ZERO`.
- `ns-keys.json` **is strict** — only `keys` and `cache` top-level keys are accepted.
- `NS.use` **is generic now** — assign the result; to chain a call use `NS.<Map<String,Object>>use(…)`. Code compiled against 0.1.x must be recompiled.
- **Token algorithm lives in the type, not the token** — you can't "ask for" a different alg; add a new token type instead.

- **Common op settings are load-checked** — a typo'd setting or wrong type fails when the schema loads, listing every problem.
- **Published versions are permanent** — fixes ship as the next version, not a re-upload.

---

io.github.nsprism:ns-secure:0.2.0-beta.2 · Apache License 2.0 · Java 8+ · modules: ns-secure-core / -log / -id / -crypto / -validation / -sanitize / -flag / -keys / -token / -common