Toggly Java

Core, Spring, Servlet and caching

Install

MAVEN DEPENDENCY

```
<dependency>
  <groupId>io.toggly</groupId>
  <artifactId>toggly-core</artifactId>
  <version>2.3.1</version>
</dependency>
```

GRADLE

```
implementation("io.toggly:toggly-core:2.3.1")
```

All add-on artifacts use version 2.3.1. Spring Boot starters: `toggly-spring-boot-starter` (Boot 3.5) and `toggly-spring-boot4-starter` (Boot 4.1).

```
toggly-spring-boot-starter   toggly-spring-boot4-start
toggly-spring-mvc            toggly-servlet
```

Initialize once

```
String key = System.getenv("TOGGLY_APP_KEY");
if (key == null || key.isBlank()) {
  throw new IllegalStateException(
      "Set TOGGLY_APP_KEY before starting");
}
TogglyConfig cfg = TogglyConfig.builder()
    .appKey(key).environment("Production")
    .refreshIntervalSeconds(30)
    .onError((message, error) →
        System.err.println(message))
    .build();

try (TogglyClient client = new TogglyClient(cfg)) {
  client.refresh(); // Attempt initial download.
  System.out.println(
      client.isEnabled("new-feature"));
```

Request and entity context

```
EvaluationContext ctx = EvaluationContext.builder()
    .identity("user-123")
    .addGroup("beta-testers")
    .claim("role", "admin") // Authenticated claim.
    .trait("plan", "premium")
    .request(RequestContext.builder()
        .country("US").build())
    .build();

EvaluationContext orderCtx = ctx.withEntity(
    new TogglyEntityContext("Order", "order-42",
        Map.of("Vip", true, "Total", 120)));
boolean express = client.isEnabled(
    "ExpressCheckout", orderCtx);
```

Build context before the first check. Use trusted auth claims

Evaluation and defaults

```
boolean on = client.isEnabled("new-ui", ctx);
String view = client.getValue("new-ui", "v2", "v1");
// Config builder defaults:
// .defaultFeatureState(false)
// .featureDefault("new-ui", false)
```

Missing definitions use configured defaults. Failed downloads retain the last good snapshot. getValue is a two-way on/off switch, not variant assignment — see the Feature Variants section for getVariant.

Feature variants (catalog-local)

```
TogglyConfig cfg = TogglyConfig.builder()
    .appKey(key).environment("Production")
    .identity("user-42") // targeting userId
    .build();

VariantResult v = client.getVariant(
    "checkout-experiment");
if (v != null) {
  String name = v.getName();
  Object cfgValue = v.getConfigurationValue();
}
Object value = client.getVariantValue(
    "checkout-experiment");
```

Variants assign locally from definitions (`variants` + `allocation`). There is no `enableVariants` toggle and no

Spring Boot host setup

Use Boot 4.1.1 and Spring 7.0.9. Import the Spring Framework BOM before the Boot BOM as shown in the Spring guide. Add `toggly-spring-boot4-starter:2.3.1` and Boot's `spring-boot-starter-aspectj`. On Boot 3.5 use `toggly-spring-boot-starter:2.3.1` instead.

APPLICATION.YML

```
toggly:
  app-key: ${TOGGLY_APP_KEY}
  environment: Production
  refresh-interval-seconds: 30
```

Place these beans in a scanned @Configuration class with @EnableAspectJAutoProxy. The starter creates the core client.

```
@Bean(destroyMethod = "close")
HttpSnapshotProvider snapshotProvider(
    TogglyConfig config) {
  return new HttpSnapshotProvider(config);
}

@Bean
FeatureAspect featureAspect(TogglyClient client) {
  return new FeatureAspect(client);
}
```

The explicit provider avoids the starter's null default-provider bean. Spring closes this provider; the client does not own a supplied provider.

Native AOP service

```
@Service
public class CheckoutService {
  @FeatureEnabled(value = "new-checkout",
      fallbackMethod = "legacy")
  public String checkout(String orderId) {
    return "new:" + orderId;
  }
  public String legacy(String orderId) {
    return "legacy:" + orderId;
  }
}
```

Call the injected service through its Spring proxy. ON calls checkout; OFF calls the public fallback with matching parameters. Self-invocation bypasses the proxy. With no fallback, disabled object-returning methods return null.

@ConditionalOnFeature only selects beans at startup; use runtime checks for refreshed flags.

Boot 4 Actuator

Since 2.2.2, `toggly-spring-boot4-starter` wires Boot 4–compatible `TogglyHealthIndicator` and `/actuator/toggly` (health APIs live in `spring-boot-health`). Add `spring-boot-starter-actuator` to expose endpoints. Disable the health contributor with `management.health.toggly.enabled=false` if needed. Use `toggly-spring-boot-starter` on Boot 3.5; do not mix starters.

MVC: register all three adapters

Add toggly-spring-mvc:2.3.1 and the host MVC starter. Import SDK types from io.toggly.spring.mvc. Add both overrides to the same WebMvcConfigurer; its constructor receives TogglyClient.

```
private final TogglyClient client;
public WebConfig(TogglyClient client) {
  this.client = client;
}
@Override
public void addInterceptors(
    InterceptorRegistry registry) {
  registry.addInterceptor(
      new TogglyContextInterceptor(
          RequestContexts::from)).order(0);
  registry.addInterceptor(
      new FeatureGateInterceptor(client)).order(1);
}
@Override
public void addArgumentResolvers(
    List<HandlerMethodArgumentResolver> resolvers) {
  resolvers.add(
      new FeatureArgumentResolver(client));
}
```

Use the complete RequestContexts helper: authenticated identity/claims, request headers, no shared identity mutation.

MVC: controller methods

INSIDE A @RESTCONTROLLER

```
@GetMapping("/beta")
@FeatureGate("beta") // OFF: 404; ON: response.
public String beta() { return "Beta"; }

@GetMapping("/dashboard")
public String dashboard(
    @FeatureArgumentResolver.FeatureFlag(
        "new-dashboard") boolean useNew) {
  return useNew ? "v2" : "v1";
}
```

For 403, use @FeatureGate(value = "beta", status = 403). Gates do not replace authorization. Native context cleanup is synchronous; pass context explicitly to async work.

Context boundaries

NESTED SYNCHRONOUS WORK

```
ContextHolder.runWithContext(ctx, () → {
  boolean on = client.isEnabled("new-feature");
}); // Restores previous context on success/error.
```

ThreadLocal does not follow executors, Servlet async dispatch or Reactor subscriptions. For async work, pass ctx explicitly or use the native Reactor context integration.

Omit defaultIdentity in multi-user configuration. Merging a configured default into an identity-less context drops claims/request/entity. Always supply the intended request identity.

DeviceType classifies Macintosh as Other, not an Apple device. OS targeting is a separate filter.

WebFlux: wrapper and context

Add toggly-spring-webflux:2.3.1 and the host WebFlux starter. Use the managed core client/provider from the Spring setup. These are beans in a scanned @Configuration class; imports are from io.toggly.spring.webflux.

```
@Bean(destroyMethod = "")
ReactiveTogglyClient reactiveTogglyClient(
    TogglyClient client) {
  return new ReactiveTogglyClient(client);
}
@Bean
@Order(-20)
TogglyContextFilter togglyContextFilter() {
  return new TogglyContextFilter(
      new HeaderReactiveContextResolver(
          "X-User-Id", "X-User-Groups"));
}
```

The core bean owns shutdown. The native wrapper has no automatic bean registration. The header resolver requires a trusted proxy; it supplies only identity/groups.

WebFlux: ordered native gate

```
@Bean
@Order(-10)
WebFilter betaApiFilter(TogglyClient client) {
  PathPattern api = PathPatternParser
      .defaultInstance.parse("/api/v2/**");
  FeatureGateFilter gate =
      FeatureGateFilter.builder(client)
          .features("beta-api")
          .pathMatcher(exchange → api.matches(
              exchange.getRequest().getPath()
                  .pathWithinApplication()))
          .blockedStatus(HttpStatus.NOT_FOUND)
          .build();
  return (exchange, chain) → Mono.defer(
      () → gate.filter(exchange, chain))
      .subscribeOn(Schedulers.boundedElastic());
}
```

Import PathPattern and PathPatternParser from org.springframework.web.util.pattern. Parsed paths match Spring routes for encoded segments and matrix parameters without turning encoded slashes into separators.

Context runs before the gate. OFF returns 404; ON subscribes to the downstream chain. An empty snapshot can perform blocking HTTP, so application scheduling wraps the native gate. Reactor context survives this scheduler boundary.

WebFlux: complete controller

```
@RestController
public class ReactiveController {
  private final ReactiveTogglyClient toggly;
  public ReactiveController(
      ReactiveTogglyClient toggly) {
    this.toggly = toggly;
  }
  @GetMapping("/api/data")
  public Mono<String> getData() {
    return toggly.isEnabled("new-data-api")
        .subscribeOn(Schedulers.boundedElastic())
        .map(on → on ? "new" : "legacy");
  }
}
```

The wrapper calls synchronous core evaluation; Mono does not guarantee nonblocking first load. Native refresh() is offloaded, but evaluation still needs the scheduling shown here when the snapshot can be empty.

Full integration guide

WebFlux configuration includes all imports and the native gate/controller wiring. Use a ReactiveContextResolver for authenticated claims and HTTP segments; emit an empty EvaluationContext for anonymous users, not an empty Mono.

Servlet: lifecycle

Use Jakarta Servlet 6.1 / Tomcat 11. Register a subclass of TogglyServletContextListener once; its createConfig override reads System.getenv("TOGGLY_APP_KEY") and rejects blank keys. The native listener owns initialization and shutdown.

The default listener maps toggly.appKey to TOGGLY_APPKEY. It does not expand environment placeholders in web.xml. See the complete listener and descriptor.

NATIVE GATE REGISTRATION IN WEB.XML

```
<filter>
  <filter-name>betaGate</filter-name>
  <filter-class>
    io.toggly.servlet.FeatureGateFilter
  </filter-class>
  <init-param>
    <param-name>features</param-name>
    <param-value>beta-feature</param-value>
  </init-param>
</filter>
<filter-mapping>
  <filter-name>betaGate</filter-name>
  <url-pattern>/beta/*</url-pattern>
</filter-mapping>
```

Map the context filter first. The native gate passes through without an initialized client/facade. Use strict startup or an application 503 guard. The runnable Servlet sample demonstrates the latter.

Caffeine: supplied provider ownership

```
SnapshotProvider cached =
    new CaffeineCachingSnapshotProvider(
        new HttpSnapshotProvider(cfg),
        CaffeineCacheConfig.builder()
            .expireAfterWrite(Duration.ofMinutes(5))
            .refreshAfterWrite(
                Duration.ofMinutes(1))
            .build());
try (TogglyClient client =
         new TogglyClient(cfg, cached)) {
  boolean on = client.isEnabled("new-feature");
} finally {
  cached.close();
}
```

For servers, keep these resources open until shutdown. Close the client, then the outer provider; that wrapper closes its HTTP delegate. Spring can own the outer provider via @Bean(destroyMethod = "close").

Polling updates the HTTP delegate but does not evict the outer snapshot cache. Its refresh/TTL can delay visibility; refresh through the outer provider when needed.

Custom evaluator: real accessor

```
client.getRegistry().register(
    "PaidPlanFilter", (filter, key, ctx) -> {
      String plan = ctx.getTraitAsString("plan");
      if (plan == null) { plan = "free"; }
      return List.of("pro", "enterprise")
          .contains(plan);
    });
```

getTrait returns Object, not Optional. Filter parameters offer string/int/double accessors; for a boolean, parse getStringParameter explicitly with a defined missing-value policy.

Caching and operational limits

Prefer definition caching. Evaluation-cache keys include context but no definition revision/time; refresh and time boundaries can leave answers stale until expiration or explicit invalidation. onRefresh does not invoke listeners.

RedisCachingSnapshotProvider closes its delegate and its own pool; a caller-supplied pool remains caller-owned. Choose the Caffeine or Redis configuration that fits the application's freshness needs.

Close resources to release sockets/schedulers and flush telemetry. The HTTP refresh thread is a daemon. Native Servlet JSON/refresh endpoints have no access control; use the application's policy before exposing them.

Offline test: real definitions

```
FeatureDefinition on = FeatureDefinition.builder()
    .featureKey("new-feature")
    .filters(List.of(FeatureFilter.alwaysOn()))
    .build();
TogglyConfig cfg = TogglyConfig.builder()
    .appKey("local-fixture")
    .enableUsageTracking(false)
    .enableMetrics(false)
    .registerContextsOnStartup(false)
    .enableLiveUpdates(false)
    .refreshIntervalSeconds(0).build();
InMemorySnapshotProvider provider =
    new InMemorySnapshotProvider(
        Map.of("new-feature", on));
try (TogglyClient client =
         new TogglyClient(cfg, provider)) {
  if (!client.isEnabled("new-feature")
      || client.isEnabled("missing")) {
    throw new AssertionError("Flag state");
  }
} finally {
  provider.close();
}
```

A definition with no filters is OFF. The provider accepts FeatureDefinition values, not booleans. A supplied provider remains caller-owned.

Live updates and refresh

TOGGLYCONFIG BUILDER CONTROLS

```
.refreshIntervalSeconds(30)
.enableLiveUpdates(true)
.onError((message, error) →
    System.err.println(message))
```

A successful refresh starts the WebSocket. Positive intervals poll already; zero disables polling only. For manual-only operation set live updates false too. An initial failure uses defaults; later failures keep last-good definitions.

Boot properties do not bind useSignedDefinitions or enableLiveUpdates; supply a custom TogglyConfig bean for these controls.

Guides and runnable examples

Snippets use the SDK imports shown in the Java guide. Its FirstFlag.java example includes compilation and run commands.

Java Servlet sample: Tomcat 11, JDK 26.0.2.1, dashboard recipe, Order context and HTTP gates.

Java Spring MVC sample: native interceptors, controller gates, boolean argument injection and request-aware FreeMarker views.

Java Spring WebFlux sample: Reactor context, native gates, parsed routes and a complete Reactor Netty host.