Service Cron Execution

Table of Contents

Status	DRAFT
Version	1.0.0
Author(s)	@Daniel Steinhauf
Date of creation	Feb 17, 2026
Last updated	Feb 17, 2026
Approved on	
<name of the person(s) in charge to approve the document >	Date of approval: yyyy-mm-dd

Overview

This guideline defines the unified architecture for **distributed-safe cron execution** across backend services implemented in:

- **Spring Boot** (Java, Hibernate, Liquibase)
- **Node.js**, including **self-hosted Next.js backend runtimes**

Cron jobs must run **exactly once cluster-wide**, even when multiple instances of the same service are deployed (e.g., in Kubernetes).

To guarantee this, services use **PostgreSQL advisory locks**, ensuring that only **one** instance executes a scheduled job at any point in time.

Concept

Each application instance (Spring Boot / Node.js / Next.js server) locally triggers cron jobs.
Before executing a job, the instance attempts to acquire a distributed lock:

```
SELECT pg_try_advisory_lock(<lock_key>);
```

Execution behavior

Condition	Behavior
Lock acquired	Execute job
Lock NOT acquired	Skip execution

After execution:

```
SELECT pg_advisory_unlock(<lock_key>);
```

Benefits of this architecture

- No external infrastructure required (e.g., Redis, Zookeeper)
- Very fast (PostgreSQL in-memory locking)
- Fully distributed and language-agnostic
- Safe during rolling deployments and autoscaling
- No schema required

Database Requirements for Distributed Cron Execution

PostgreSQL advisory locks:

- **Do NOT require tables**
- **Do NOT write to disk**
- Are **session-scoped**
- Are automatically released if:
  - the process crashes
  - the DB connection drops

PostgreSQL functions used:

Function	Purpose
`pg_try_advisory_lock(key)`	Attempts to acquire lock without waiting
`pg_advisory_unlock(key)`	Releases the lock
Session close	Auto-releases all locks

Lock Key Rules

- Each cron job must have a **globally unique lock key**
- Keys must be consistent across environments (dev/stage/prod)
- Services must **not** reuse lock keys

Environment Variable Schema for Cron Jobs

To define cron jobs dynamically from environment configuration, each service declares:

```
CRON_JOBS=<JSON object defining per-job schedule + lock key>
```

JSON schema

Each job is defined as:

```
"<jobName>": {
  "schedule": "<cron expression>",
  "lockKey": <integer number>
}
```

Example

```
CRON_JOBS={
  "tokenCleanup": { "schedule": "0 */1 * * * *", "lockKey": 5101 },
  "sessionPrune": { "schedule": "0 */5 * * * *", "lockKey": 5102 }
}
```

Why this schema?

- One env var per service → easy to audit
- Job configuration is declarative
- Job schedule + key stored together
- Fully compatible with Node.js, Next.js, and Spring Boot
- Easy to validate at startup

Further Structuring for Large Organizations

Large systems with many microservices require **lock-key ranges** to avoid collisions.

Lock-key allocation strategy

Each service is assigned a **start key**.

Jobs increment keys sequentially:

```
<service_start> + <job_index>
```

This guarantees global uniqueness.

Example:

```
lt-credit-service → 1301–1399
lt-invoice-service → 1551–1599
```

Naming convention

To avoid ambiguity:

```
<service>.<jobName>
```

Example:

```
"lt-credit-service.dailyLimitRefresh": { "schedule": "...", "lockKey": 1301 }
```

Spring Boot — Distributed Cron Execution (AOP-Based)

Spring Boot integrates distributed cron execution *transparently* via:

- A custom annotation `@DistributedScheduled`
- A lock aspect that uses PostgreSQL advisory locks
- A scheduler registrar that discovers annotated methods

This avoids developers having to manually implement locking.

Annotation

```java
import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface DistributedScheduled {
    String cron();
    long lockKey();
}
```

Distributed Lock Aspect

```java
@Slf4j
@Component
public class DistributedLockAspect {

    private final JdbcTemplate jdbc;

    public DistributedLockAspect(JdbcTemplate jdbc) {
        this.jdbc = jdbc;
    }

    public void execute(Object bean, Method method, long lockKey) {
        boolean acquired = false;

        try {
            acquired = Boolean.TRUE.equals(
                jdbc.queryForObject("SELECT pg_try_advisory_lock(?)", Boolean.class, lockKey)
            );

            if (!acquired) {
                log.info("Skipping {} - lock {} held elsewhere", method.getName(), lockKey);
                return;
            }

            log.info("Executing {} with lock {}", method.getName(), lockKey);
            method.invoke(bean);

        } catch (Exception e) {
            log.error("Error executing job {}", method.getName(), e);

        } finally {
            if (acquired) {
                try {
                    jdbc.queryForObject("SELECT pg_advisory_unlock(?)", Boolean.class, lockKey);
                } catch (Exception unlockError) {
                    log.error("Failed to unlock key {}", lockKey, unlockError);
                }
            }
        }
    }
}
```

Scheduler Registrar

```java
@Component
public class DistributedSchedulerRegistrar implements SchedulingConfigurer {
```

```
    private final ApplicationContext ctx;
    private final TaskScheduler scheduler;
    private final DistributedLockAspect aspect;

    public DistributedSchedulerRegistrar(ApplicationContext ctx,
                                         TaskScheduler scheduler,
                                         DistributedLockAspect aspect)
{
        this.ctx = ctx;
        this.scheduler = scheduler;
        this.aspect = aspect;
    }

    @Override
    public void configureTasks(ScheduledTaskRegistrar registrar) {
        registrar.setTaskScheduler(scheduler);

        ctx.getBeansWithAnnotation(Component.class).forEach((name,
bean) -> {
            Class<?> targetClass = AopUtils.getTargetClass(bean);

            for (Method m : targetClass.getMethods()) {
                DistributedScheduled ann =
m.getAnnotation(DistributedScheduled.class);

                if (ann != null) {
                    registrar.addCronTask(
                        () -> aspect.execute(bean, m, ann.lockKey()),
                        ann.cron()
                    );
                }
            }
        });
    }
}
```

Example Distributed Cron Job

```
@Component
public class CleanupJob {

    @DistributedScheduled(cron = "0 */1 * * * *", lockKey = 5001)
    public void cleanup() {
        System.out.println("Distributed cleanup executed");
    }
}
```

Node.js / Next.js Cron Execution

Next.js (self-hosted) supports cron execution only in the **Node.js server runtime**:

- NOT in API routes
- NOT in Edge runtime
- NOT in browser React

Use the same distributed locking helper for all Node-based runtimes.

Distributed Cron Helper

```
const cron = require('node-cron');

class DistributedCron {
  constructor(pgClient) {
    this.pg = pgClient;
  }

  schedule(schedule, lockKey, job) {
    cron.schedule(schedule, async () => {
      let acquired = false;

      try {
        const res = await this.pg.query(
          "SELECT pg_try_advisory_lock($1) AS acquired",
```

```
          [lockKey]
        );
        acquired = !!res.rows?.[0]?.acquired;

        if (!acquired) {
          console.log(`[DistributedCron] skipping job ${lockKey}`);
          return;
        }

        console.log(`[DistributedCron] executing job ${lockKey}`);
        await job();

      } finally {
        if (acquired) {
          await this.pg.query("SELECT pg_advisory_unlock($1)",
[lockKey]);
        }
      }
    });
  }
}

module.exports = DistributedCron;
```

Using CRON_JOBS

```
const CRON_JOBS = JSON.parse(process.env.CRON_JOBS);

const dcron = new DistributedCron(pg);

Object.entries(CRON_JOBS).forEach(([jobName, cfg]) => {
  dcron.schedule(cfg.schedule, cfg.lockKey, async () => {
    console.log(`Executing job: ${jobName}`);
    // job implementation
  });
});
```

Lock-Key Registry

Each service is assigned a starting key. The range/number of cron jobs is 50 as of now:

Service Name	Start Key
lt-carbone-connector	1
lt-compass-connector	51
lt-crif-connector	101
lt-hyundai-connector	151
lt-idnow-connector	201
lt-intrum-connector	251
lt-ksv-connector	301
lt-mesoneer-connector	351
lt-odoo-connector	401
lt-pdf4Me-connector	451
lt-postmark-connector	501
lt-schufa-connector	551
lt-tesla-connector	601

lt-zek-connector	651
lt-zefix-connector	701
bawag-proxy	751
calculator	801
dealer	851
dealer-service-v1	901
ebicsbatch	951
insurance-service-v1	1001
keycloak-theme	1051
leasing-calculator-service-v1	1101
lt-accounting-service	1151
lt-aml-service	1201
lt-company-registry-check-service	1251
lt-credit-service	1301
lt-customer-sync-service	1351
lt-flowapp-v2-service	1401
lt-invoice-delivery-service	1451
lt-invoice-payment-sync-service	1501
lt-invoice-service	1551
lt-invoice-updater-service	1601
lt-kyc-service	1651
lt-partner-service	1701
lt-payment-plan-service	1751
lt-proxy-service	1801
partner	1851
partner-service-v1	1901
proxy	1951
proxy-service-v1	2001
lt-crm-be	2051
lt-crm-fe	2101
lt-flowapp	2151

next-react	2201

Summary

This guideline provides:

- A robust model for **distributed cron execution**
- Architecture compatible with Spring Boot, Node.js, and Next.js
- A standardized **CRON_JOBS JSON schema** defining job schedule + lock key
- A global lock-key strategy and registry
- Ready-to-use implementation for both runtimes