Task Execution & Scheduling

Builds on the thread-pool foundation "Threads" covers at the language level -- Spring's TaskExecutor abstraction, background work with @Async and @EnableAsync, why self-invocation breaks @Async (the SAME proxy pitfall as "Transaction Management"), scheduling with @Scheduled's fixed rate/fixed delay/initial delay/cron expressions, and configuring scheduling's own separate thread pool (TaskScheduler). The 3rd lesson in the Advanced Spring series.

Advanced 35 min

Last updated: August 24, 2026
Read online: https://www.learnforgex.com/en/topics/task-execution-and-scheduling

---

"Threads," in the Java course, covered creating threads and managing a pool of them with `ExecutorService` at the language level. This lesson is about what Spring builds on top of that foundation for a typical Spring Boot application: a declarative way to say "run this work in the background" (`@Async`) and a declarative way to say "run this work on a schedule" (`@Scheduled`) — two related but genuinely different tools, covered together because they share the same underlying thread-pool machinery.

Why Background and Scheduled Work Matters

An HTTP request handler that does everything synchronously — including slow work like sending an email or calling a third-party API — makes the caller wait for all of it, even the parts that don't need an immediate answer. And plenty of real work isn't triggered by a request at all: purging old data, generating a daily report, syncing with an external system on a timer. Task execution (`@Async`) solves the first problem; scheduling (`@Scheduled`) solves the second. Keep the distinction sharp: `@Async` means "run this asynchronously, right now, in response to something"; `@Scheduled` means "run this at a specific time or interval, on its own."

Spring's TaskExecutor Abstraction

`TaskExecutor` is Spring's own interface for running work asynchronously — under the hood, an implementation like `ThreadPoolTaskExecutor` manages a real `java.util.concurrent` thread pool, the exact kind covered in "Threads," just exposed as a Spring bean instead of built directly with `Executors` factory methods.

Configuring a Thread Pool with ThreadPoolTaskExecutor

A thread pool matters because creating a new thread per task is expensive and unbounded — a pool reuses a fixed set of threads and queues work when they're all busy, exactly the trade-off "Threads" already covered at the `ExecutorService` level.

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor;

// ThreadPoolTaskExecutor is Spring's own wrapper around the same
// java.util.concurrent thread pool machinery covered in "Threads" --
// under the hood it manages a real ExecutorService, but exposes it as a
// Spring bean with property-style configuration instead of the builder
// methods used there.
@Configuration
class TaskExecutorConfig {

    @Bean
    ThreadPoolTaskExecutor taskExecutor() {
        ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
        executor.setCorePoolSize(4);      // threads kept alive even when idle
        executor.setMaxPoolSize(8);       // ceiling the pool can grow to under load
        executor.setQueueCapacity(50);    // tasks queued once corePoolSize is busy
        executor.setThreadNamePrefix("app-task-");
        executor.initialize();
        return executor;
    }
}
```

`corePoolSize` is how many threads stay alive even when idle; `maxPoolSize` is the ceiling the pool can grow to under load; `queueCapacity` is how many tasks wait once `corePoolSize` threads are all busy. Defining this as a `@Bean` is what makes it a real, injectable Spring component instead of a plain object your code constructs by hand.

Running Work in the Background: @Async and @EnableAsync

`@EnableAsync`, on a `@Configuration` class, turns on Spring's async proxying for the whole application — without it, `@Async` is silently ignored, and every annotated method just runs synchronously, exactly as if the annotation weren't there. `@Async`, on a method, is what actually says "dispatch a call to this method onto a separate thread (from the configured `TaskExecutor`), instead of running it on the caller's own thread."

```
import org.springframework.context.annotation.Configuration;
import org.springframework.scheduling.annotation.Async;
import org.springframework.scheduling.annotation.EnableAsync;
import org.springframework.stereotype.Service;

import java.util.concurrent.CompletableFuture;
```

```
// @EnableAsync turns on Spring's async proxying for the whole application
// -- without it, @Async is silently ignored and every method runs
// synchronously, on the calling thread, exactly as if the annotation
// weren't there at all.
@Configuration
@EnableAsync
class AsyncConfig {
}

@Service
class ReportService {

    // @Async makes THIS method run on a separate thread (from the
    // configured TaskExecutor) whenever it's called through Spring, and
    // dispatches the WHOLE method body -- including the sleep below --
    // there. Thread.sleep(...) here is only a stand-in for genuinely slow
    // work (a large query, rendering a PDF, a slow external call) -- never
    // use Thread.sleep in real application code; it's used here purely to
    // make the asynchronous timing visible.
    @Async
    public CompletableFuture<String> generateReport(String reportId) {
        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }

        // By this line, the "slow work" is already done -- on the
        // separate thread @Async already dispatched this method to.
        // completedFuture(...) does NOT make anything asynchronous by
        // itself; it just packages a value Java already has into the
        // CompletableFuture shape this method's signature promises to
        // return.
        return CompletableFuture.completedFuture("Report " + reportId + " ready");
    }
}

// The caller's side of the same interaction -- what actually happens with
// the CompletableFuture generateReport(...) hands back.
class ReportRequestHandler {

    private final ReportService reportService;

    ReportRequestHandler(ReportService reportService) {
        this.reportService = reportService;
    }

    public String handle(String reportId) {
        // generateReport(...) returns IMMEDIATELY -- before the sleep,
        // before the report is actually ready -- because @Async already
        // sent the real work to another thread. "future" is NOT that
        // other thread; it's a placeholder object this thread can hold
        // onto right now, representing a result that will exist later.
        CompletableFuture<String> future = reportService.generateReport(reportId);

        System.out.println("Report generation started, request thread continues...");
```

```
        // thenAccept(...) registers a callback -- "when a result
        // eventually lands in this future, run this code with it" --
        // WITHOUT blocking this thread to wait for that moment.
        future.thenAccept(result -> System.out.println("Async result: " + result));

        return "Report " + reportId + " is being generated";
    }
}
```

`Thread.sleep(3000)` inside `generateReport(...)` is only a stand-in for genuinely slow work — a large database query, rendering a PDF, a slow call to another service. (Never write `Thread.sleep(...)` in real application code; it's used here purely to make the asynchronous timing visible.) Because the method is `@Async`, Spring dispatches the ENTIRE method body — including that three-second sleep — onto a separate thread the moment it's called; the calling thread never sits there waiting for it.

What the caller actually gets back. An `@Async` method that needs to hand back a result can't simply `return` it the normal way — its caller has already moved on before the work even finishes. This is exactly why `generateReport(...)` returns `CompletableFuture<String>` instead of a plain `String`: a `CompletableFuture` is an object representing a result that doesn't exist YET, but will exist LATER.

`ReportRequestHandler.handle(...)`, in the same example, shows what a caller actually does with one:

```
CompletableFuture<String> future = reportService.generateReport(reportId);

System.out.println("Report generation started, request thread continues...");

future.thenAccept(result -> System.out.println("Async result: " + result));
```

`generateReport(reportId)` returns IMMEDIATELY — before the sleep, before the report is actually ready — because `@Async` already sent the real work off to another thread. `future` is NOT that other thread; it's a placeholder object the calling thread can hold onto, right now, while the actual work is still running somewhere else. `thenAccept(...)` registers a callback — "when a result eventually lands in this future, run this code with it" — without blocking the calling thread to wait for that moment to arrive.

```
ReportRequestHandler.handle("123")
        |
        | reportService.generateReport("123")
        v
  CompletableFuture<String> returned immediately
        |                                  \
        v                                   \ (meanwhile, on another thread)
"request thread continues..."                \
        |                                     v
        v                                  Thread.sleep(3000) ...
   future.thenAccept(...) registered             |
        |                                        v
        |                                "Report 123 ready"
        |                                        |
        +------------------<-------------------+
        |
        v
"Async result: Report 123 ready"   (printed once the future completes)
```

@Async vs. CompletableFuture: two different jobs. It's worth being precise about what each piece is actually responsible for, since they're easy to blur together:

- `@Async` controls WHERE and HOW the method call is executed — on a separate thread, dispatched through Spring's proxy, instead of on the caller's own thread.
- `CompletableFuture<T>` represents the EVENTUAL RESULT of that execution — a value that isn't ready yet, but that the caller can hold onto and react to once it is.

`@Async` can, in principle, be used on a method returning `void` (as `NotificationService.sendPushNotification(...)` does further below) — there's simply nothing for the caller to react to later. `CompletableFuture` shows up specifically when the caller genuinely needs the eventual result — it isn't `@Async`'s only valid return type, just the one that matters here.

Why `completedFuture(...)` doesn't make anything asynchronous. `CompletableFuture.completedFuture("Report " + reportId + " ready")` creates a `CompletableFuture` that is ALREADY completed, right there, with the given value — it does nothing more than wrap a value Java already has in hand.

```
@Async                  → "Run this method asynchronously."
CompletableFuture<T>    → "Represent the result of that asynchronous work."
thenAccept(...)         → "Do something when that result becomes available."
```

By the time `generateReport(...)` reaches its `return` line, the slow work has ALREADY happened — on the separate thread `@Async` dispatched it to. `completedFuture(...)` isn't what made any of that asynchronous; `@Async` already did that, before this method's body ever started running on that thread. `completedFuture(...)`'s only job is to package a value Java already has into the `CompletableFuture` shape the method's signature promises to return.

> This is a narrow, deliberately minimal introduction to `CompletableFuture` — just enough to use it as `@Async`'s return type. Composing multiple futures together, blocking with `join()` / `get()`, or combining results with `allOf()` / `anyOf()` belong to a dedicated concurrency lesson, not this one.

Why Self-Invocation Breaks @Async

`@Async` works through the exact same proxy mechanism as `@Transactional`, covered in "Transaction Management" — Spring wraps a proxy around the real bean, and it's the PROXY that actually intercepts a call and dispatches it to a separate thread.

```
import org.springframework.scheduling.annotation.Async;
import org.springframework.stereotype.Service;

// @Async works through the exact same PROXY mechanism as @Transactional
// (see "Transaction Management") -- Spring wraps a proxy around the real
// bean, and the proxy is what actually intercepts a call and dispatches
// it to a separate thread. Calling an @Async method through that proxy
// (from OUTSIDE the class) works; calling it directly on "this" (from
// INSIDE the class) bypasses the proxy entirely, and @Async is silently
// ignored -- the method just runs like any ordinary method call, on the
// current thread. This is the identical self-invocation pitfall already
// covered there for @Transactional, now showing up for @Async instead.
@Service
class NotificationService {

    @Async
    public void sendPushNotification(String userId) {
        System.out.println("Sending push notification to " + userId
                + " on thread " + Thread.currentThread().getName());
    }

    // BROKEN: calling sendPushNotification(...) from another method in
    // the SAME class goes through "this", not through Spring's proxy --
    // @Async has no effect here, and this runs synchronously.
    public void processOrder_broken(String userId) {
        sendPushNotification(userId); // runs on the CALLING thread, not async
    }
}

@Service
class OrderService {

    private final NotificationService notificationService;

    OrderService(NotificationService notificationService) {
        this.notificationService = notificationService;
    }

    // WORKING: notificationService here is the Spring-managed PROXY,
    // injected from a separate bean -- calling through it correctly
    // dispatches to a separate thread, exactly as @Async promises.
    public void processOrder_working(String userId) {
        notificationService.sendPushNotification(userId); // genuinely async
    }
}
```

`processOrder_broken(...)` calls `sendPushNotification(...)` through `this` — from inside the same class — which bypasses the proxy entirely, so `@Async` has no effect and the call runs synchronously. `processOrder_working(...)` calls the SAME method through an injected `NotificationService` bean instead, going through the real proxy, and genuinely dispatches to a separate thread. This is the identical self-invocation pitfall already covered for `@Transactional`, showing up here for `@Async` instead.

> Self-invocation is the single most common reason `@Async` (or `@Transactional`) "doesn't seem to work." If an annotated method's caller lives in the same class, the annotation is silently skipped — always call it through an injected bean reference, never through `this`.

Scheduling Work: @Scheduled, Fixed Rate, Fixed Delay, and Initial Delay

`@EnableScheduling` turns on Spring's scheduling infrastructure, the way `@EnableAsync` does for `@Async`. `@Scheduled` then accepts several different timing strategies.

```
import org.springframework.scheduling.annotation.EnableScheduling;
import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.stereotype.Component;

// @EnableScheduling turns on Spring's scheduling infrastructure for the
// whole application, the same way @EnableAsync does for @Async.
@Component
class MetricsReporter {

    // fixedRate: a new run starts every 5 seconds, measured from the
    // START of the previous run -- if a run takes longer than the rate,
    // the next one starts as soon as the current one finishes (it does
    // NOT overlap by default), but runs can end up "catching up" back to
    // back with no gap between them.
    @Scheduled(fixedRate = 5000)
    public void reportMetrics() {
        System.out.println("Reporting metrics at " + System.currentTimeMillis());
    }

    // fixedDelay: a new run starts 10 seconds after the PREVIOUS run
    // actually FINISHED, not from when it started -- this guarantees a
    // real gap between runs regardless of how long each one takes.
    //
    // initialDelay: the scheduler waits 30 seconds after the application
    // starts before the very first run -- useful when a task depends on
    // other startup work (a cache warming up, a connection pool
    // initializing) finishing first.
    @Scheduled(initialDelay = 30000, fixedDelay = 10000)
    public void syncWithExternalSystem() {
        System.out.println("Syncing with external system at " + System.currentTimeMillis());
    }
}
```

`fixedRate` starts a new run every N milliseconds, measured from when the PREVIOUS run STARTED — if a run takes longer than the rate, the next one starts as soon as the current one finishes, with no gap. `fixedDelay` starts a new run N milliseconds after the previous run FINISHED — this guarantees a real gap regardless of how long each run takes. `initialDelay` simply delays the very FIRST run after startup, useful when a task depends on other startup work finishing first.

Cron Expressions

A cron expression describes an actual SCHEDULE — specific times and days — rather than an interval relative to the previous run, for requirements that are genuinely calendar-based.

```
import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.stereotype.Component;

// A cron expression describes a SCHEDULE (specific times/days), unlike
// fixedRate/fixedDelay, which describe an INTERVAL relative to the
// previous run -- reach for cron when the requirement is genuinely
// calendar-based ("every night at 2 AM," "every Monday morning"), not
// just "repeat every N minutes."
@Component
class DailyCleanupJob {

    // Spring's six-field cron format: second minute hour day-of-month month day-of-week.
    // "0 0 2 * * *" -- second 0, minute 0, hour 2, every day-of-month,
    // every month, every day-of-week -- runs once, every day, at 2:00:00 AM.
    @Scheduled(cron = "0 0 2 * * *")
    public void purgeExpiredSessions() {
        System.out.println("Purging expired sessions at 2 AM");
    }

    // "0 0 9 * * MON-FRI" -- 9:00:00 AM, but only Monday through Friday --
    // cron expressions can restrict to specific days the way a fixed
    // interval never could.
    @Scheduled(cron = "0 0 9 * * MON-FRI")
    public void sendWeekdayDigest() {
        System.out.println("Sending weekday digest at 9 AM");
    }
}
```

Spring's cron format has six fields: second, minute, hour, day-of-month, month, day-of-week. `"0 0 2 * * *"` means "every day at 2:00:00 AM"; `"0 0 9 * * MON-FRI"` restricts to specific days of the week — something no fixed interval could express on its own.

Which Thread Pool Actually Runs a @Scheduled Task?

A detail that's easy to miss: by DEFAULT, Spring runs every single `@Scheduled` method on ONE shared thread — a slow scheduled task can delay every other scheduled task behind it, even ones whose own trigger time has already arrived. This has nothing to do with the `@Async` `TaskExecutor` configured earlier — `@Scheduled` uses a completely separate `TaskScheduler`.

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.scheduling.annotation.EnableScheduling;
import org.springframework.scheduling.annotation.SchedulingConfigurer;
import org.springframework.scheduling.concurrent.ThreadPoolTaskScheduler;
import org.springframework.scheduling.config.ScheduledTaskRegistrar;

// A critical, easy-to-miss detail: by DEFAULT, Spring runs every single
// @Scheduled method on ONE shared thread -- if syncWithExternalSystem()
// from ScheduledFixedRateDelayExample takes 20 seconds, purgeExpiredSessions()
// from ScheduledCronExample simply waits, even though its own cron time
// already arrived. This has nothing to do with the @Async TaskExecutor
// configured elsewhere -- @Scheduled uses a completely separate
// TaskScheduler, which needs its OWN pool configured if multiple
// scheduled methods genuinely need to run at once.
@Configuration
@EnableScheduling
class SchedulingConfig implements SchedulingConfigurer {

    @Bean
    ThreadPoolTaskScheduler taskScheduler() {
        ThreadPoolTaskScheduler scheduler = new ThreadPoolTaskScheduler();
        scheduler.setPoolSize(5); // now up to 5 @Scheduled methods can run concurrently
        scheduler.setThreadNamePrefix("scheduled-task-");
        scheduler.initialize();
        return scheduler;
    }

    @Override
    public void configureTasks(ScheduledTaskRegistrar taskRegistrar) {
        taskRegistrar.setTaskScheduler(taskScheduler());
    }
}
```

Configuring a `ThreadPoolTaskScheduler` bean with a real `poolSize`, and registering it through `SchedulingConfigurer`, is what lets multiple `@Scheduled` methods genuinely run concurrently instead of queuing behind each other on Spring's single default thread.

A Practical Example

`@Async` and `@Scheduled` often show up together in the same small feature, each doing the job only it can do.

```
import org.springframework.scheduling.annotation.Async;
import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.stereotype.Service;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

import java.time.LocalDateTime;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.CompletableFuture;
import java.util.Map;

// A realistic pairing of both mechanisms in one small feature: a REST
// endpoint that must respond IMMEDIATELY (task execution -- @Async), and
// a nightly job that cleans up stale data left behind by it (scheduling
// -- @Scheduled). Neither annotation could do the other's job: the
// controller can't afford to wait for a slow email send, and the cleanup
// isn't triggered by any request at all -- it just needs to run at 3 AM.
@Service
class SignupConfirmationService {

    private final Map<String, LocalDateTime> pendingConfirmations = new
ConcurrentHashMap<>();

    @Async
    public CompletableFuture<Void> sendConfirmationEmail(String email) {
        pendingConfirmations.put(email, LocalDateTime.now());
        System.out.println("Sending confirmation email to " + email
                + " on thread " + Thread.currentThread().getName());
        return CompletableFuture.completedFuture(null);
    }

    @Scheduled(cron = "0 0 3 * * *")
    public void purgeStaleUnconfirmedSignups() {
        LocalDateTime cutoff = LocalDateTime.now().minusDays(7);
        pendingConfirmations.entrySet().removeIf(entry -> entry.getValue().isBefore
(cutoff));
        System.out.println("Purged stale unconfirmed signups older than 7 days");
    }
}

@RestController
class SignupController {

    private final SignupConfirmationService confirmationService;

    SignupController(SignupConfirmationService confirmationService) {
        this.confirmationService = confirmationService;
    }

    @PostMapping("/signup")
    public String signup(@RequestParam String email) {
        confirmationService.sendConfirmationEmail(email); // returns instantly, email sends
in the background
        return "Signup received for " + email;
    }
}
```

`SignupController.signup(...)` needs to respond immediately, so sending the confirmation email is `@Async` — the HTTP response doesn't wait for it. Cleaning up stale, never-confirmed signups isn't triggered by any request at all, so it's `@Scheduled` with a nightly cron expression instead. Neither annotation could do the other's job here.

Notice that `signup(...)` calls `sendConfirmationEmail(...)` and simply discards the returned `CompletableFuture<Void>` entirely — and that's perfectly reasonable here, because the controller has nothing left to do once the email is sent. Contrast that with `ReportRequestHandler.handle(...)` from earlier, which held onto its `CompletableFuture<String>` and attached a `thenAccept(...)` callback specifically because it DID need to react once the result became available. The rule of thumb: if the caller doesn't care about the result, calling the `@Async` method and moving on is enough; if the caller needs to react to the result later, keep the returned future and register a callback on it.

Best Practices

- Always call an `@Async` or `@Transactional` method through an injected bean reference, never through `this` — self-invocation silently skips the proxy.
- Give a `ThreadPoolTaskExecutor` (and a `ThreadPoolTaskScheduler`, if you need one) an explicit, bounded configuration — an unconfigured default is rarely what a real application actually needs.
- Reach for `fixedDelay` when a task's own duration should never overlap into its next run, and `fixedRate` when a consistent cadence matters more than that guarantee.
- Configure a dedicated `TaskScheduler` with a real pool size the moment an application has more than one `@Scheduled` method that genuinely needs to run concurrently.

Common Mistakes

- Forgetting `@EnableAsync` or `@EnableScheduling` entirely, then being confused why `@Async` / `@Scheduled` methods just run like ordinary synchronous methods.
- Calling an `@Async` method from within the same class and being surprised it ran synchronously — this is the self-invocation pitfall, not a bug.
- Confusing `fixedRate` and `fixedDelay` — assuming `fixedRate` guarantees a gap between runs, when it actually measures from the previous run's START, not its end.
- Assuming multiple `@Scheduled` methods automatically run in parallel, when Spring's default `TaskScheduler` runs them all on a single shared thread unless configured otherwise.

Summary, Cheat Sheet, and Glossary

Summary

- `TaskExecutor` (commonly `ThreadPoolTaskExecutor`) is Spring's bean-based wrapper around the same thread-pool machinery "Threads" covers at the language level.
- `@EnableAsync` + `@Async` runs a method in the background, returning immediately instead of blocking the caller.
- `@Async` controls WHERE/HOW a call executes; `CompletableFuture<T>` represents the result of that execution, to be consumed later (with `thenAccept(...)`, for example) — the two are separate, complementary concerns.
- `CompletableFuture.completedFuture(...)` does not make anything asynchronous by itself; it only wraps an already-known value — `@Async` is what dispatched the work to another thread in the first place.
- If the caller doesn't need the eventual result, it can call an `@Async` method and ignore the returned future entirely; if it does, it holds onto the future and reacts to it later.
- `@Async` (like `@Transactional`) works through a proxy — self-invocation from inside the same class bypasses it silently.
- `@EnableScheduling` + `@Scheduled` runs a method on a schedule: `fixedRate` (interval from the previous start), `fixedDelay` (interval from the previous finish), `initialDelay` (delay before the first run), or a cron expression for calendar-based timing.
- `@Scheduled` uses its own `TaskScheduler`, separate from `@Async`'s `TaskExecutor`, and defaults to a single shared thread unless configured with a real pool size.

Cheat Sheet

```
// Thread pool for @Async
@Bean
ThreadPoolTaskExecutor taskExecutor() {
    var executor = new ThreadPoolTaskExecutor();
    executor.setCorePoolSize(4);
    executor.setMaxPoolSize(8);
    executor.initialize();
    return executor;
}

// @Async, called through an injected bean (never "this")
@EnableAsync
@Async
CompletableFuture<String> generateReport(String id) { ... }

// The caller's side: receive the future, keep going, react later
CompletableFuture<String> future = reportService.generateReport(id);
// ... do other work here, without waiting ...
future.thenAccept(result -> System.out.println(result));

// If the caller doesn't need the result, it can just ignore the future
reportService.generateReport(id); // fire-and-forget

// @Scheduled timing strategies
@Scheduled(fixedRate = 5000)                        // every 5s, from previous START
@Scheduled(fixedDelay = 10000)                        // 10s after previous FINISH
@Scheduled(initialDelay = 30000, fixedDelay = 10000)  // wait 30s, then every 10s
@Scheduled(cron = "0 0 2 * * *")                      // every day at 2 AM

// A dedicated pool for @Scheduled
@Bean
ThreadPoolTaskScheduler taskScheduler() {
    var scheduler = new ThreadPoolTaskScheduler();
    scheduler.setPoolSize(5);
    scheduler.initialize();
    return scheduler;
}
```

Glossary

- **TaskExecutor**: Spring's bean-based abstraction over a thread pool, used to run work asynchronously.
- **@Async**: an annotation marking a method to run on a separate thread when called through its Spring proxy.
- **CompletableFuture<T>**: an object representing a result that doesn't exist yet but will exist later — what an `@Async` method typically returns instead of `T` directly.
- **thenAccept(...)**: registers a callback on a `CompletableFuture` to run once its result becomes available, without blocking the calling thread.
- **completedFuture(...)**: creates a `CompletableFuture` that is already completed with a given value — it does not, by itself, make anything asynchronous.

- **Self-invocation**: calling an annotated method through `this` instead of through an injected bean, silently bypassing its proxy.
- **fixedRate vs. fixedDelay**: `fixedRate` schedules from the previous run's start; `fixedDelay` schedules from the previous run's finish.
- **Cron expression**: a six-field schedule (second, minute, hour, day-of-month, month, day-of-week) for calendar-based, rather than interval-based, timing.
- **TaskScheduler**: the separate abstraction `@Scheduled` runs on, independent of `@Async`'s `TaskExecutor`.