Task Execution & Scheduling

"Threads"in dil seviyesinde işlediği thread pool temelinin üzerine inşa edilir -- Spring'in TaskExecutor soyutlaması, @Async ve @EnableAsync ile arka plan işi, self-invocation'ın @Async'i neden bozduğu ("Transaction Management"taki AYNI proxy tuzağı), @Scheduled ile fixed rate/fixed delay/initial delay/cron ifadeleri, ve scheduling'in kendi ayrı thread pool'unun (TaskScheduler) yapılandırılması. Advanced Spring serisinin 3.'sü.

İleri 35 dk

Son güncelleme: 24 Ağustos 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/task-execution-and-scheduling

---

Java kursundaki "Threads", dil seviyesinde thread oluşturmayı ve `ExecutorService` ile bir thread havuzunu yönetmeyi işledi. Bu ders, tipik bir Spring Boot uygulaması için Spring'in bu temelin üzerine ne inşa ettiğiyle ilgili: "bu işi arka planda çalıştır" demenin bildirimsel bir yolu (`@Async`) ve "bu işi bir zamanlamaya göre çalıştır" demenin bildirimsel bir yolu (`@Scheduled`) -- ilişkili ama gerçekten farklı iki araç, aynı alttaki thread-havuzu mekanizmasını paylaştıkları için birlikte işleniyor.

Arka Plan ve Zamanlanmış İşler Neden Önemli

Her şeyi eşzamanlı (synchronous) yapan bir HTTP request handler'ı -- bir e-posta göndermek ya da üçüncü taraf bir API çağırmak gibi yavaş işler dahil -- çağıranı, hiç acil bir cevaba ihtiyacı olmayan kısımlar dahil, her şey için bekletir. Ve pek çok gerçek iş bir request tarafından hiç tetiklenmez: eski veriyi temizlemek, günlük bir rapor üretmek, bir zamanlayıcıyla harici bir sistemle senkronize etmek. Task execution (`@Async`) birinci sorunu çözer; scheduling (`@Scheduled`) ikinciyi çözer. Ayrımı net tut: `@Async` , "bunu asenkron olarak, tam şimdi, bir şeye tepki olarak çalıştır" demektir; `@Scheduled` , "bunu belirli bir zamanda ya da aralıkta, kendi başına çalıştır" demektir.

Spring'in TaskExecutor Soyutlaması

`TaskExecutor` , Spring'in işi asenkron çalıştırmak için kendi interface'idir -- altında, `ThreadPoolTaskExecutor` gibi bir implementasyon, "Threads"te işlenenin AYNI türünden gerçek bir `java.util.concurrent` thread havuzunu yönetir, yalnızca `Executors` factory metotlarıyla doğrudan inşa etmek yerine bir Spring bean'i olarak sunulur.

ThreadPoolTaskExecutor ile Bir Thread Havuzu Yapılandırmak

Bir thread havuzu önemlidir çünkü görev başına yeni bir thread oluşturmak pahalı ve sınırsızdır -- bir havuz, sabit bir thread kümesini yeniden kullanır ve hepsi meşgulken işi kuyruğa alır, tam olarak "Threads"in `ExecutorService` seviyesinde zaten işlediği ödünleşim.

```
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`, boşta olduğunda bile canlı kalan thread sayısıdır; `maxPoolSize`, havuzun yük altında büyüyebileceği tavandır; `queueCapacity`, `corePoolSize` thread'lerinin hepsi meşgulken bekleyen görev sayısıdır. Bunu bir `@Bean` olarak tanımlamak, onu kodunun elle inşa ettiği düz bir nesne yerine gerçek, inject edilebilir bir Spring bileşeni yapan şeydir.

Arka Planda İş Çalıştırmak: @Async ve @EnableAsync

Bir `@Configuration` sınıfındaki `@EnableAsync`, tüm uygulama için Spring'in async proxy'lemesini açar -- olmadan, `@Async` sessizce görmezden gelinir, ve annotation'lı her metot, sanki annotation hiç yokmuş gibi eşzamanlı çalışır. Bir metot üzerindeki `@Async` ise gerçekte "bu metoda yapılan çağrıyı, çağıranın kendi thread'i yerine ayrı bir thread'e (yapılandırılmış `TaskExecutor`'dan) gönder" diyen şeydir.

```
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";
    }
}
```

`generateReport(...)`'in içindeki `Thread.sleep(3000)`, yalnızca gerçekten yavaş bir iş için (büyük bir veritabanı sorgusu, bir PDF render etmek, başka bir servise yavaş bir çağrı) bir yer tutucudur. (Gerçek uygulama kodunda asla `Thread.sleep(...)` yazma; burada yalnızca asenkron zamanlamayı görünür kılmak için kullanılıyor.) Metot `@Async` olduğu için, Spring metodun TÜM gövdesini -- üç saniyelik sleep dahil -- çağrıldığı anda ayrı bir thread'e gönderir; çağıran thread onu hiç beklemez.

Çağıran gerçekte ne geri alır. Bir sonuç geri vermesi gereken bir `@Async` metodu, bunu sıradan `return` ile yapamaz -- çağıranı, iş bitmeden çoktan devam etmiştir. `generateReport(...)`'in düz bir `String` yerine `CompletableFuture<String>` döndürmesinin tam nedeni budur: bir `CompletableFuture`, HENÜZ var olmayan ama DAHA SONRA var olacak bir sonucu temsil eden bir nesnedir.

Aynı örnekteki `ReportRequestHandler.handle(...)`, çağıranın bununla gerçekte ne yaptığını gösteriyor:

```
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)`, sleep'ten önce, rapor gerçekten hazır olmadan önce HEMEN döner -- çünkü `@Async` gerçek işi zaten başka bir thread'e göndermiştir. `future`, o başka thread'in kendisi DEĞİLDİR; gerçek iş hâlâ başka bir yerde çalışırken çağıran thread'in şu anda elinde tutabileceği bir yer tutucu nesnedir. `thenAccept(...)`, bir callback kaydeder -- "bu future'a er ya da geç bir sonuç geldiğinde, bunu onunla çalıştır" -- çağıran thread'i o an gelene kadar bloklamadan.

```
ReportRequestHandler.handle("123")
        |
        | reportService.generateReport("123")
        v
  CompletableFuture<String> hemen döner
        |                                    \
        v                                     \ (bu arada, başka bir thread'de)
"request thread devam ediyor..."               \
        |                                       v
        v                                  Thread.sleep(3000) ...
   future.thenAccept(...) kaydedildi             |
        |                                        v
        |                                "Report 123 ready"
        |                                        |
        +------------------<-------------------+
        |
        v
"Async result: Report 123 ready"   (future tamamlanınca yazdırılır)
```

@Async vs. CompletableFuture: iki farklı iş. Her parçanın gerçekte neden sorumlu olduğu konusunda net olmakta fayda var, çünkü birbirine karıştırılması kolaydır:

- `@Async`, metot çağrısının NEREDE ve NASIL çalıştırıldığını kontrol eder -- çağıranın kendi thread'i yerine, Spring'in proxy'si üzerinden gönderilen ayrı bir thread'de.
- `CompletableFuture<T>`, o çalıştırmanın NİHAİ SONUCUNU temsil eder -- henüz hazır olmayan, ama çağıranın elinde tutup hazır olduğunda tepki verebileceği bir değer.

`@Async`, ilke olarak, `void` döndüren bir metotta da kullanılabilir (birazdan gelecek `NotificationService.sendPushNotification(...)`'ın yaptığı gibi) -- çağıranın daha sonra tepki vereceği hiçbir şey yoktur basitçe. `CompletableFuture`, özellikle çağıranın nihai sonuca gerçekten ihtiyaç duyduğu durumlarda ortaya çıkar -- `@Async`'in tek geçerli dönüş türü değildir, yalnızca burada önemli olanıdır.

completedFuture(...) neden hiçbir şeyi asenkron yapmaz. `CompletableFuture.completedFuture("Report " + reportId + " ready")`, tam orada, verilen değerle ZATEN tamamlanmış bir `CompletableFuture` oluşturur -- Java'nın elinde zaten olan bir değeri sarmalamaktan başka hiçbir şey yapmaz.

```
@Async                  → "Bu metodu asenkron çalıştır."
CompletableFuture<T>    → "O asenkron işin sonucunu temsil et."
thenAccept(...)         → "O sonuç hazır olduğunda bir şey yap."
```

`generateReport(...)`, `return` satırına ulaştığında, yavaş iş ZATEN gerçekleşmiştir -- `@Async`'in onu gönderdiği ayrı thread'de. `completedFuture(...)`, bunların hiçbirini asenkron yapan şey değildir; bunu, bu metodun gövdesi o thread'de hiç çalışmaya başlamadan önce `@Async` zaten yapmıştır. `completedFuture(...)`'ın tek işi, Java'nın elinde zaten olan bir değeri, metodun imzasının döndürmeyi vaat ettiği `CompletableFuture` şekline paketlemektir.

> Bu, `CompletableFuture`'a dar, bilinçli olarak minimal bir giriş -- yalnızca onu `@Async`'in dönüş türü olarak kullanmaya yetecek kadar. Birden fazla future'ı birleştirmek, `join()` / `get()` ile bloklamak, ya da sonuçları `allOf()` / `anyOf()` ile birleştirmek, bu derse değil, ayrı, özel bir concurrency dersine ait.

Self-Invocation @Async'i Neden Bozuyor

`@Async`, "Transaction Management"te işlenen `@Transactional` ile TAM OLARAK AYNI proxy mekanizması üzerinden çalışır -- Spring, gerçek bean'in etrafına bir proxy sarar, ve bir çağrıyı gerçekten yakalayıp ayrı bir thread'e gönderen PROXY'dir.

```
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(...)`, `sendPushNotification(...)`'ı `this` üzerinden -- aynı sınıfın içinden -- çağırır, ki bu proxy'yi tamamen atlar, bu yüzden `@Async`'in hiçbir etkisi olmaz ve çağrı eşzamanlı çalışır. `processOrder_working(...)` ise AYNI metodu bunun yerine inject edilmiş bir `NotificationService` bean'i üzerinden çağırır, gerçek proxy'den geçer, ve gerçekten ayrı bir thread'e gönderir. Bu, `@Transactional` için zaten işlenen aynı self-invocation tuzağının, burada `@Async` için ortaya çıkması.

> Self-invocation, `@Async`'in (ya da `@Transactional`'ın) "çalışmıyor gibi görünmesinin" en yaygın tek nedenidir. Annotation'lı bir metodun çağıranı aynı sınıfın içinde yaşıyorsa, annotation sessizce atlanır -- onu her zaman inject edilmiş bir bean referansı üzerinden çağır, asla `this` üzerinden değil.

İşi Zamanlamak: @Scheduled, Fixed Rate, Fixed Delay ve Initial Delay

`@EnableScheduling`, `@EnableAsync`'in `@Async` için yaptığı gibi, Spring'in zamanlama altyapısını açar. `@Scheduled`, o zaman birkaç farklı zamanlama stratejisini kabul eder.

```
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`, ÖNCEKİ çalışmanın BAŞLADIĞI andan ölçülen, her N milisaniyede bir yeni çalışma başlatır -- bir çalışma orandan daha uzun sürerse, bir sonraki, şimdiki bitince hiç boşluk olmadan hemen başlar. `fixedDelay`, önceki çalışma BİTTİKTEN N milisaniye sonra yeni bir çalışma başlatır -- bu, her çalışmanın ne kadar sürdüğünden bağımsız olarak gerçek bir boşluğu garanti eder. `initialDelay`, basitçe başlangıçtan sonraki İLK çalışmayı geciktirir, bir görev başka başlangıç işinin önce bitmesine bağlıysa faydalıdır.

Cron İfadeleri

Bir cron ifadesi, önceki çalışmaya göreli bir aralık yerine, gerçekten takvim-tabanlı gereksinimler için, gerçek bir ZAMANLAMAYI -- belirli saatler ve günler -- tanımlar.

```
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'in cron formatının altı alanı vardır: saniye, dakika, saat, ayın günü, ay, haftanın günü. `"0 0 2 * * *"`, "her gün saat 2:00:00'de" demektir; `"0 0 9 * * MON-FRI"`, haftanın belirli günleriyle sınırlar -- hiçbir sabit aralığın tek başına ifade edemeyeceği bir şey.

Bir @Scheduled Görevini Gerçekte Hangi Thread Havuzu Çalıştırır?

Kaçırması kolay bir detay: VARSAYILAN olarak, Spring her tek `@Scheduled` metodunu TEK, paylaşılan bir thread'de çalıştırır -- yavaş bir zamanlanmış görev, kendi tetikleme zamanı çoktan gelmiş olsa bile, arkasındaki her diğer zamanlanmış görevi geciktirebilir. Bunun daha önce yapılandırılan `@Async` `TaskExecutor`'ıyla hiçbir ilgisi yok -- `@Scheduled`, tamamen ayrı bir `TaskScheduler` kullanır.

```
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());
    }
}
```

Gerçek bir `poolSize` ile bir `ThreadPoolTaskScheduler` bean'i yapılandırmak, ve onu `SchedulingConfigurer` üzerinden kaydetmek, birden fazla `@Scheduled` metodunun Spring'in tek varsayılan thread'inde birbirinin arkasında kuyruğa girmek yerine gerçekten eş zamanlı çalışmasına izin veren şeydir.

Pratik Bir Örnek

`@Async` ve `@Scheduled`, genelde aynı küçük özellikte birlikte ortaya çıkar, her biri yalnızca kendisinin yapabileceği işi yaparak.

```
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(...)`'ın hemen yanıt vermesi gerekir, bu yüzden onay e-postasını göndermek `@Async`'tir -- HTTP yanıtı onu beklemez. Eski, hiç onaylanmamış kayıtları temizlemek ise hiçbir request tarafından tetiklenmez, bu yüzden bunun yerine gece çalışan bir cron ifadesiyle `@Scheduled`'dır. Burada ne annotation diğerinin işini yapabilirdi.

`signup(...)`'ın `sendConfirmationEmail(...)`'ı çağırıp döndürülen `CompletableFuture<Void>`'ı tamamen görmezden geldiğine dikkat et -- ve bu burada tamamen makuldür, çünkü controller e-posta gönderildikten sonra yapacak başka bir şey bırakmaz. Bunu, `CompletableFuture<String>`'ını elinde tutup, sonuç hazır olduğunda gerçekten tepki vermesi GEREKTİĞİ için özellikle bir `thenAccept(...)` callback'i ekleyen az önceki `ReportRequestHandler.handle(...)` ile karşılaştır. Genel kural: çağıran sonucu umursamıyorsa, `@Async` metodu çağırıp devam etmek yeterlidir; çağıranın sonuca daha sonra tepki vermesi gerekiyorsa, döndürülen future'ı elinde tutup üzerine bir callback kaydeder.

Best Practices

- Bir `@Async` ya da `@Transactional` metodunu her zaman inject edilmiş bir bean referansı üzerinden çağır, asla `this` üzerinden değil -- self-invocation, proxy'yi sessizce atlar.
- Bir `ThreadPoolTaskExecutor`'a (ve gerekirse bir `ThreadPoolTaskScheduler`'a) açık, sınırlı bir yapılandırma ver -- yapılandırılmamış bir varsayılan, gerçek bir uygulamanın gerçekten ihtiyaç duyduğu şey nadiren olur.
- Bir görevin kendi süresinin bir sonraki çalışmasına asla taşmaması gerektiğinde `fixedDelay`'e, tutarlı bir ritim bu garantiden daha önemli olduğunda `fixedRate`'e başvur.
- Bir uygulamanın gerçekten eş zamanlı çalışması gereken birden fazla `@Scheduled` metodu olduğu anda, gerçek bir pool size ile özel bir `TaskScheduler` yapılandır.

Yaygın Hatalar

- `@EnableAsync` ya da `@EnableScheduling`'i tamamen unutup, sonra `@Async` / `@Scheduled` metotlarının neden sıradan eşzamanlı metotlar gibi çalıştığına şaşırmak.
- Aynı sınıfın içinden bir `@Async` metodu çağırıp eşzamanlı çalışmasına şaşırmak -- bu bir hata değil, self-invocation tuzağı.
- `fixedRate` ve `fixedDelay`'i karıştırmak -- `fixedRate`'in, gerçekte önceki çalışmanın bitişinden değil BAŞLANGICINDAN ölçüldüğü hâlde, çalışmalar arasında bir boşluk garanti ettiğini varsaymak.
- Birden fazla `@Scheduled` metodunun otomatik olarak paralel çalıştığını varsaymak, oysa Spring'in varsayılan `TaskScheduler`'ı, aksi yapılandırılmadıkça hepsini tek, paylaşılan bir thread'de çalıştırır.

Özet, Cheat Sheet ve Terimler Sözlüğü

Özet

- `TaskExecutor` (genelde `ThreadPoolTaskExecutor`), "Threads"in dil seviyesinde işlediği AYNI thread-havuzu mekanizmasının Spring'in bean-tabanlı sarmalayıcısıdır.
- `@EnableAsync` + `@Async`, bir metodu arka planda çalıştırır, çağıranı bloklamak yerine hemen döner.
- `@Async`, bir çağrının NEREDE/NASIL çalıştırıldığını kontrol eder; `CompletableFuture<T>`, o çalıştırmanın sonucunu, daha sonra tüketilmek üzere (örneğin `thenAccept(...)` ile) temsil eder -- ikisi ayrı, birbirini tamamlayan konulardır.
- `CompletableFuture.completedFuture(...)`, tek başına hiçbir şeyi asenkron yapmaz; yalnızca zaten bilinen bir değeri sarmalar -- işi baştan başka bir thread'e gönderen `@Async`'tir.
- Çağıranın nihai sonuca ihtiyacı yoksa, bir `@Async` metodu çağırıp döndürülen future'ı tamamen görmezden gelebilir; ihtiyacı varsa, future'ı elinde tutup daha sonra ona tepki verir.
- `@Async` (tıpkı `@Transactional` gibi) bir proxy üzerinden çalışır -- aynı sınıfın içinden self-invocation onu sessizce atlar.
- `@EnableScheduling` + `@Scheduled`, bir metodu bir zamanlamaya göre çalıştırır: `fixedRate` (önceki başlangıçtan aralık), `fixedDelay` (önceki bitişten aralık), `initialDelay` (ilk çalışmadan önceki gecikme), ya da takvim-tabanlı zamanlama için bir cron ifadesi.
- `@Scheduled`, `@Async`'in `TaskExecutor`'ından ayrı kendi `TaskScheduler`'ını kullanır, ve aksi yapılandırılmadıkça tek, paylaşılan bir thread'e varsayılan olur.

Cheat Sheet

```
// @Async için thread havuzu
@Bean
ThreadPoolTaskExecutor taskExecutor() {
    var executor = new ThreadPoolTaskExecutor();
    executor.setCorePoolSize(4);
    executor.setMaxPoolSize(8);
    executor.initialize();
    return executor;
}

// @Async, inject edilmiş bir bean üzerinden çağrılır (asla "this" değil)
@EnableAsync
@Async
CompletableFuture<String> generateReport(String id) { ... }

// Çağıranın tarafı: future'ı al, devam et, daha sonra tepki ver
CompletableFuture<String> future = reportService.generateReport(id);
// ... beklemeden burada başka iş yap ...
future.thenAccept(result -> System.out.println(result));

// Çağıranın sonuca ihtiyacı yoksa, future'ı basitçe görmezden gelebilir
reportService.generateReport(id); // fire-and-forget

// @Scheduled zamanlama stratejileri
@Scheduled(fixedRate = 5000)                        // her 5sn, önceki BAŞLANGIÇTAN
@Scheduled(fixedDelay = 10000)                        // önceki BİTİŞTEN 10sn sonra
@Scheduled(initialDelay = 30000, fixedDelay = 10000)  // 30sn bekle, sonra her 10sn
@Scheduled(cron = "0 0 2 * * *")                      // her gün saat 2'de

// @Scheduled için özel bir havuz
@Bean
ThreadPoolTaskScheduler taskScheduler() {
    var scheduler = new ThreadPoolTaskScheduler();
    scheduler.setPoolSize(5);
    scheduler.initialize();
    return scheduler;
}
```

Terimler Sözlüğü

- **TaskExecutor**: işi asenkron çalıştırmak için kullanılan, Spring'in bean-tabanlı thread havuzu soyutlaması.
- **@Async**: Spring proxy'si üzerinden çağrıldığında bir metodu ayrı bir thread'de çalışacak şekilde işaretleyen annotation.
- **CompletableFuture<T>**: henüz var olmayan ama daha sonra var olacak bir sonucu temsil eden nesne -- bir `@Async` metodunun genelde `T`'yi doğrudan döndürmek yerine döndürdüğü şey.
- **thenAccept(...)**: bir `CompletableFuture` üzerine, sonucu hazır olduğunda -- çağıran thread'i bloklamadan -- çalışacak bir callback kaydeder.
- **completedFuture(...)**: verilen bir değerle zaten tamamlanmış bir `CompletableFuture` oluşturur -- tek başına hiçbir şeyi asenkron yapmaz.

- **Self-invocation**: annotation'lı bir metodu inject edilmiş bir bean üzerinden değil `this` üzerinden çağırmak, proxy'sini sessizce atlamak.
- **fixedRate vs. fixedDelay**: `fixedRate`, önceki çalışmanın başlangıcından zamanlar; `fixedDelay`, önceki çalışmanın bitişinden zamanlar.
- **Cron ifadesi**: aralık-tabanlı değil takvim-tabanlı zamanlama için altı alanlı bir zamanlama (saniye, dakika, saat, ayın günü, ay, haftanın günü).
- **TaskScheduler**: `@Async`'in `TaskExecutor`'ından bağımsız, `@Scheduled`'ın üzerinde çalıştığı ayrı soyutlama.