Spring Boot Auto-Configuration ve Properties

@SpringBootApplication, @Conditional ailesi (@ConditionalOnClass /@ConditionalOnMissingBean/@ConditionalOnProperty), @Value, @ConfigurationProperties, @Profile ve ApplicationEvent/@EventListener.

İleri 55 dk

Son güncelleme: 11 Ağustos 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/autoconfiguration-properties

---

Spring Boot Auto-Configuration ve Properties

Component Scanning dersinde bean'lerin container tarafından nasıl bulunduğunu, Spring IoC Container dersinde ise bean'lerin nasıl tanımlandığını ve yaşam döngüsünün nasıl işlediğini gördük. Bu son derste, Spring Boot'un bu ikisinin üzerine kattığı üçüncü katmana bakıyoruz: `@SpringBootApplication` ve auto-configuration'ın perde arkasında nasıl çalıştığına, `application.yml`'den `@Value` / `@ConfigurationProperties` ile property okumaya, `@Profile` ile ortama özel yapılandırmaya, ve `ApplicationEvent` ile container'ın kendi kendine haber vermesine. Bu dersin sonunda, projenin kendi `application.yml` dosyalarının ve `LearningPlatformApplication`'daki tek bir `@SpringBootApplication` satırının aslında neyi temsil ettiğini tam olarak anlayacaksın.

Spring Boot Auto-Configuration Nedir?

Auto-configuration, classpath'te hangi kütüphanelerin bulunduğuna bakarak Spring Boot'un -- sen hiçbir `@Bean` metodu yazmadan -- senin yerine bean'ler kaydetmesidir. Örneğin bu projede `spring-boot-starter-data-jpa` ve `postgresql` bağımlılığı olduğu için, Spring Boot bir `DataSource` bean'i, bir `EntityManagerFactory` bean'i ve bir JPA `TransactionManager` bean'i otomatik olarak kurar -- hiçbirini `WebConfig` gibi bir `@Configuration` sınıfında elle tanımlamadık:

```
// Elle yazsaydık (asla yazmıyoruz, Spring Boot bizim yerimize yapıyor):
@Configuration
class ManualDataSourceConfig {
    @Bean
    DataSource dataSource() {
        HikariDataSource ds = new HikariDataSource();
        ds.setJdbcUrl("jdbc:postgresql://localhost:5433/learning");
        ds.setUsername("learning");
        ds.setPassword("learning");
        return ds;
    }
}
```

`application.yml`'deki `spring.datasource.*` anahtarlarını yazman dışında, yukarıdaki gibi bir sınıfı hiç görmedin -- çünkü auto-configuration, classpath'te `org.postgresql.Driver` ve `spring-boot-starter-data-jpa`'yı görüp bu bean'i senin yerine kaydediyor.

Neden Var?

Component Scanning dersinde Java Config'in tekrarlayıcı olduğunu, component scanning'in bunu sınıfın kendi üzerine taşıyarak azalttığını görmüştük. Ama component scanning yalnızca **senin kendi sınıfların** için işe yarar -- `DataSource`, `EntityManagerFactory`, `RequestMappingHandlerMapping` gibi bean'ler senin yazmadığın, üçüncü parti kütüphanelerin sınıflarıdır; bunlara `@Component` ekleyemezsin (bkz. Component Scanning dersindeki "Component Scanning vs Java Config: Ne Zaman Hangisi?").

Auto-configuration olmasaydı, her yeni Spring Boot projesinde yukarıdaki gibi düzinelerce `@Bean` metodunu -- `DataSource`, `TransactionManager`, `RequestMappingHandlerMapping`, `ViewResolver`, `ObjectMapper`, ve daha fazlasını -- elle yazman gerekirdi. Auto-configuration, "classpath'te şu kütüphane varsa, muhtemelen şu bean'lere ihtiyacın vardır" varsayımını framework'ün kendisine taşır -- sen yalnızca `application.yml`'de birkaç property ile bu varsayılanları özelleştirirsin.

Tarihçe

Spring Boot, 2014'te 1.0 sürümüyle çıktı -- o zamana kadar bir Spring uygulaması kurmak, XML tabanlı yapılandırma (Spring IoC Container dersindeki "Tarihçe" bölümünde bahsettiğimiz `ClassPathXmlApplicationContext` dönemi) ya da onlarca elle yazılmış `@Bean` metoduyla saatler sürebiliyordu. Spring Boot'un temel vaadi "convention over configuration" idi: makul varsayılanlarla başla, yalnızca varsayılandan sapmak istediğinde bir şey yaz.

`@EnableAutoConfiguration` (ve onu saran `@SpringBootApplication`) bu vaadin teknik temelidir. Başlangıçta `META-INF/spring.factories` dosyasında listelenen auto-configuration sınıflarını okuyordu; Spring Boot 2.7'de (2022) bu mekanizma, daha hızlı ve daha açık olan `META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports` dosyasına taşındı

-- bu proje Spring Boot 4.1 kullandığı için (bkz. `pom.xml`), yeni mekanizmayı kullanıyor. `@ConditionalOnClass`, `@ConditionalOnMissingBean` gibi `@Conditional` türevleri de 1.0'dan beri auto-configuration'ın temelini oluşturuyor.

@SpringBootApplication: Üç Anotasyonun Birleşimi

`LearningPlatformApplication` sınıfının üzerindeki tek `@SpringBootApplication` annotation'ı aslında üç ayrı annotation'ın birleşimidir -- ikisini zaten önceki derslerden tanıyoruz:

```
import org.springframework.boot.SpringBootConfiguration;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.annotation.AnnotationUtils;

// @SpringBootApplication is a convenience annotation: it is itself meta-annotated
// with three annotations we can already recognize. This file proves that
// composition with reflection, the same way StereotypeAnnotationsExample (in the
// Component Scanning lesson) proved @Service carries @Component underneath.
@SpringBootApplication
class DemoApplication {
}

class SpringBootApplicationExample {
    public static void main(String[] args) {
        boolean carriesSpringBootConfiguration =
                AnnotationUtils.findAnnotation(DemoApplication.class,
SpringBootConfiguration.class) != null;
        boolean carriesEnableAutoConfiguration =
                AnnotationUtils.findAnnotation(DemoApplication.class,
EnableAutoConfiguration.class) != null;
        boolean carriesComponentScan =
                AnnotationUtils.findAnnotation(DemoApplication.class, ComponentScan.class) !
= null;

        System.out.println("Carries @SpringBootConfiguration: " +
carriesSpringBootConfiguration);
        // Carries @SpringBootConfiguration: true
        System.out.println("Carries @EnableAutoConfiguration: " +
carriesEnableAutoConfiguration);
        // Carries @EnableAutoConfiguration: true
        System.out.println("Carries @ComponentScan: " + carriesComponentScan);
        // Carries @ComponentScan: true

        // @SpringBootConfiguration is itself meta-annotated with @Configuration --
        // that's exactly why a @SpringBootApplication-annotated class (like this
        // project's own LearningPlatformApplication) can be passed directly to
        // an ApplicationContext, wherever a @Configuration class is expected.
        boolean springBootConfigurationIsConfiguration =
                AnnotationUtils.findAnnotation(SpringBootConfiguration.class, Configuration.
class) != null;
        System.out.println("@SpringBootConfiguration carries @Configuration: " +
springBootConfigurationIsConfiguration);
        // @SpringBootConfiguration carries @Configuration: true
    }
}
```

`@SpringBootConfiguration`, `@Configuration`'ın (Spring IoC Container dersi) özel bir türevidir. `@ComponentScan`, Component Scanning dersinde gördüğümüz, argümansız kullanıldığında kendi paketini (ve alt paketlerini) tarayan annotation'ın ta kendisi -- bu yüzden `com.cdurgun.learning` altındaki her `@Controller` / `@Service` elle kaydedilmeden bulunuyor. Üçüncüsü, bu dersin asıl konusu olan `@EnableAutoConfiguration`.

@Conditional Ailesi ve Auto-Configuration Mekanizması

Auto-configuration'ın kalbinde `@Conditional` ailesi yatar: bir bean'in ya da tüm bir `@Configuration` sınıfının, belirli bir koşul sağlandığında (ya da sağlanmadığında) kaydedilmesini sağlayan annotation'lar. `@EnableAutoConfiguration` işleme girdiğinde, Spring Boot'un kendi `spring-boot-autoconfigure` modülündeki yüzlerce `@Configuration` sınıfını (`DataSourceAutoConfiguration`, `JpaRepositoriesAutoConfiguration`, `ThymeleafAutoConfiguration` gibi) sırayla dener -- her biri kendi `@Conditional` annotation'larıyla korunur, ve koşulu sağlamayan hiçbir şey kaydedilmez. En sık kullanılan iki türevi -- `@ConditionalOnClass` ve `@ConditionalOnMissingBean` -- sonraki iki bölümde kendi ellerimizle kullanacağız.

@ConditionalOnClass: Sınıf Classpath'te Varsa

`@ConditionalOnClass`, "bu bean'i yalnızca belirtilen sınıf classpath'te varsa kaydet" der -- gerçek `DataSourceAutoConfiguration`'ın, yalnızca bir JDBC sürücüsü projenin bağımlılıklarında varsa devreye girmesiyle aynı mekanizma:

```
import org.springframework.boot.autoconfigure.condition.ConditionalOnClass;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

// @ConditionalOnClass is the annotation Spring Boot's own auto-configuration
// classes use dozens of times over: "only register this bean if a given class
// is present on the classpath." Here we use it directly on our own @Bean
// methods, with one class we know for certain IS on the classpath and one
// that is NOT, to see both branches.
@Configuration
class JsonSupportConfig {

    // com.fasterxml.jackson.databind.ObjectMapper really is on the classpath --
    // spring-boot-starter-web brings Jackson in transitively. This bean IS
    // registered.
    @Bean
    @ConditionalOnClass(name = "com.fasterxml.jackson.databind.ObjectMapper")
    String jacksonSupportMarker() {
        return "Jackson support enabled";
    }

    // No such class exists anywhere on the classpath -- this bean is silently
    // skipped, exactly like a real auto-configuration class skips registering
    // (say) a DataSource bean when no JDBC driver is present at all.
    @Bean
    @ConditionalOnClass(name = "com.example.NoSuchLibraryEverInstalled")
    String missingLibrarySupportMarker() {
        return "This should never print";
    }
}

class ConditionalOnClassExample {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext
(JsonSupportConfig.class);

        System.out.println(context.containsBean("jacksonSupportMarker"));
        // true
        System.out.println(context.containsBean("missingLibrarySupportMarker"));
        // false

        context.close();
    }
}
```

`com.fasterxml.jackson.databind.ObjectMapper` gerçekten classpath'te olduğu için (Jackson, `spring-boot-starter-web` üzerinden dolaylı olarak geliyor) ilk bean kaydediliyor; uydurma bir sınıf adı verdiğimiz ikinci bean ise sessizce atlanıyor -- hiçbir hata fırlatılmıyor, bean sadece hiç var olmamış gibi davranıyor.

@ConditionalOnMissingBean: Kullanıcı Kendi Bean'ini Tanımladıysa

`@ConditionalOnMissingBean`, kütüphanelerin "makul bir varsayılan sunuyorum, ama sen kendi bean'ini tanımlarsan onu kullan" demesini sağlar -- gerçek Spring Boot'ta `ObjectMapper`, `RestTemplateBuilder` gibi birçok bean tam olarak bu şekilde davranır:

```
import org.springframework.boot.autoconfigure.condition.ConditionalOnMissingBean;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

interface MessageFormatter {
    String format(String message);
}

// Simulates the "library default, application override" pattern used
// everywhere in real Spring Boot auto-configuration: a library ships a
// sensible default bean, marked @ConditionalOnMissingBean, so any bean the
// application itself defines of the same type silently takes priority.
@Configuration
class LibraryDefaultsConfig {
    @Bean
    @ConditionalOnMissingBean
    MessageFormatter messageFormatter() {
        return message -> "[default] " + message;
    }
}

@Configuration
class UserOverrideConfig {
    @Bean
    MessageFormatter messageFormatter() {
        return message -> "[custom] " + message;
    }
}

class ConditionalOnMissingBeanExample {
    public static void main(String[] args) {
        // Case 1: only the library's config is present -- its default wins.
        AnnotationConfigApplicationContext withoutOverride =
                new AnnotationConfigApplicationContext(LibraryDefaultsConfig.class);
        System.out.println(withoutOverride.getBean(MessageFormatter.class).format("hello"));
        // [default] hello
        withoutOverride.close();

        // Case 2: the application also registers its own bean. Order matters:
        // UserOverrideConfig is given first, so its bean definition already
        // exists by the time @ConditionalOnMissingBean is evaluated for
        // LibraryDefaultsConfig -- exactly why real auto-configuration classes
        // are always processed after the application's own @Configuration
        // classes.
        AnnotationConfigApplicationContext withOverride =
                new AnnotationConfigApplicationContext(UserOverrideConfig.class,
LibraryDefaultsConfig.class);
        System.out.println(withOverride.getBean(MessageFormatter.class).format("hello"));
        // [custom] hello
        withOverride.close();
    }
}
```

Sıralama burada kritik: uygulamanın kendi `@Configuration` sınıfı, kütüphanenin varsayılanını tanımlayan sınıftan **önce** işlenmeli -- gerçek Spring Boot'ta bu, auto-configuration sınıflarının

her zaman uygulamanın kendi `@Configuration` sınıflarından **sonra** işlenmesiyle garanti edilir, tam olarak bu yüzden kendi tanımladığın bir bean her zaman auto-configuration'ın varsayılanının önüne geçer.

Kendi Auto-Configuration'ımızı Yazmak

Gerçek bir Spring Boot starter'ının nasıl göründüğünü küçük ölçekte kendimiz yazarak görelim -- `@ConditionalOnProperty`, bir özelliğin tamamen `application.yml`'den açılıp kapatılmasını sağlar:

```
import org.springframework.boot.autoconfigure.condition.ConditionalOnProperty;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.env.ConfigurableEnvironment;
import org.springframework.core.env.MapPropertySource;

import java.util.Map;

interface CacheWarmer {
    void warmUp();
}

// A hand-written stand-in for what a real Spring Boot "starter" auto-configuration
// class looks like: @ConditionalOnProperty lets the application turn a whole
// feature on/off from application.yml, with a safe default (off) if the
// property is never set at all.
@Configuration
class CacheWarmerAutoConfiguration {

    @Bean
    @ConditionalOnProperty(name = "app.cache-warmer.enabled", havingValue = "true",
matchIfMissing = false)
    CacheWarmer cacheWarmer() {
        return () -> System.out.println("Cache warmed up.");
    }
}

class CustomAutoConfigurationExample {
    public static void main(String[] args) {
        // Case 1: the property is set to true -- the bean is registered.
        AnnotationConfigApplicationContext enabledContext = new
AnnotationConfigApplicationContext();
        addProperty(enabledContext, "app.cache-warmer.enabled", "true");
        enabledContext.register(CacheWarmerAutoConfiguration.class);
        enabledContext.refresh();
        System.out.println(enabledContext.containsBean("cacheWarmer"));
        // true
        enabledContext.close();

        // Case 2: the property is never set -- matchIfMissing = false means
        // the bean is skipped, exactly like an optional Spring Boot feature
        // that stays off until the application opts in.
        AnnotationConfigApplicationContext disabledContext = new
AnnotationConfigApplicationContext();
```

```
        disabledContext.register(CacheWarmerAutoConfiguration.class);
        disabledContext.refresh();
        System.out.println(disabledContext.containsBean("cacheWarmer"));
        // false
        disabledContext.close();
    }

    private static void addProperty(AnnotationConfigApplicationContext context, String key,
String value) {
        ConfigurableEnvironment environment = context.getEnvironment();
        environment.getPropertySources().addFirst(new MapPropertySource("test", Map.of(key,
value)));
    }
}
```

`matchIfMissing = false` sayesinde, property hiç tanımlanmamışsa bean varsayılan olarak **kapalı** kalıyor -- gerçek Spring Boot'taki birçok isteğe bağlı özelliğin (`spring.cache.type`, `management.endpoints.web.exposure.include` gibi) davranışıyla aynı: sen açıkça istemeden devreye girmiyor.

application.properties ve application.yml

Spring Boot iki eşdeğer dosya formatını destekler: düz `key=value` satırlarından oluşan `application.properties`, ve iç içe geçmiş yapıyı girintiyle ifade eden `application.yml`. Bu proje YAML'ı tercih ediyor -- kendi `application.yml` dosyasından bir parça:

```
spring:
  application:
    name: learning-platform
  profiles:
    active: dev
  thymeleaf:
    cache: false

server:
  port: 8080
```

Aynı ayarlar `.properties` formatında şöyle görünürdü: `spring.application.name=learning-platform`, `spring.profiles.active=dev`, `spring.thymeleaf.cache=false`, `server.port=8080`. İkisi de aynı düz nokta-ayrılmış property anahtarlarına (`spring.thymeleaf.cache` gibi) çözümlenir -- YAML sadece bunu iç içe girintilerle daha az tekrarlı yazmanı sağlar. Sonraki bölümlerde bu anahtarları `@Value` ve `@ConfigurationProperties` ile Java tarafında nasıl okuyacağımızı göreceğiz.

@Value ile Tekil Property Enjeksiyonu

`@Value`, `application.yml`'den tek bir property'yi doğrudan bir alana ya da constructor parametresine enjekte eder -- en basit okuma yöntemi, ama hiçbir gruplama sunmaz:

```
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.support.PropertySourcesPlaceholderConfigurer;
import org.springframework.core.env.ConfigurableEnvironment;
import org.springframework.core.env.MapPropertySource;

import java.util.Map;

// @Value pulls a single property value into a field or constructor parameter --
// the simplest way to read application.yml/application.properties, but with
// no grouping and no type validation beyond the target field's own type.
class GreetingService {

    @Value("${app.greeting.prefix:Hello}")
    private String prefix;

    // ${...} placeholders are resolved first, and only then is the resulting
    // string evaluated as a SpEL expression (#{...}) -- so this becomes
    // "#{'Hello'.toUpperCase()}" before it is ever evaluated.
    @Value("#{'${app.greeting.prefix:Hello}'.toUpperCase()}")
    private String shoutedPrefix;

    String greet(String name) {
        return prefix + ", " + name + "!";
    }

    String shoutedGreet(String name) {
        return shoutedPrefix + ", " + name + "!";
    }
}

@Configuration
class GreetingConfig {

    // Outside Spring Boot, ${...} placeholders in @Value are NOT resolved
    // automatically -- this bean is what actually makes them work. It must
    // be `static`, so the container can run it very early, before other
    // @Configuration classes are even fully processed. In a real Spring Boot
    // app you never write this yourself: PropertyPlaceholderAutoConfiguration
    // (triggered by @EnableAutoConfiguration) registers it for you -- exactly
    // the kind of boilerplate auto-configuration exists to remove (see "Why
    // Does It Exist?").
    @Bean
    static PropertySourcesPlaceholderConfigurer propertySourcesPlaceholderConfigurer() {
        return new PropertySourcesPlaceholderConfigurer();
    }

    @Bean
    GreetingService greetingService() {
        return new GreetingService();
    }
}

class ValueInjectionExample {
    public static void main(String[] args) {
        // Case 1: the property is set explicitly (simulated here with a
```

```
        // MapPropertySource, standing in for application.yml).
        AnnotationConfigApplicationContext withProperty = new
AnnotationConfigApplicationContext();
        ConfigurableEnvironment env1 = withProperty.getEnvironment();
        env1.getPropertySources().addFirst(new MapPropertySource("test", Map.of("app.
greeting.prefix", "Merhaba")));
        withProperty.register(GreetingConfig.class);
        withProperty.refresh();
        System.out.println(withProperty.getBean(GreetingService.class).greet("Ayse"));
        // Merhaba, Ayse!
        withProperty.close();

        // Case 2: the property is never set -- the ":Hello" default after the
        // colon kicks in, instead of a startup failure.
        AnnotationConfigApplicationContext withoutProperty = new
AnnotationConfigApplicationContext();
        withoutProperty.register(GreetingConfig.class);
        withoutProperty.refresh();
        System.out.println(withoutProperty.getBean(GreetingService.class).greet("Ayse"));
        // Hello, Ayse!
        System.out.println(withoutProperty.getBean(GreetingService.class).shoutedGreet
("Ayse"));
        // HELLO, Ayse!
        withoutProperty.close();
    }
}
```

`${app.greeting.prefix:Hello}` ifadesindeki `:Hello` kısmı, property hiç tanımlanmamışsa kullanılacak varsayılan değeri belirtir -- property zorunlu değilse uygulamanın çökmesini önler. Kod örneğindeki yorumda da belirtildiği gibi, saf Spring IoC Container'da (Spring Boot olmadan) `${...}` yer tutucularının çalışması için `PropertySourcesPlaceholderConfigurer` bean'ini elle tanımlaman gerekir -- Spring Boot'ta bunu hiç yazmazsın, çünkü `@EnableAutoConfiguration` bunu senin için otomatik kaydeder. Bu, "Neden Var?" bölümünde bahsettiğimiz tam olarak o türden bir tekrarı ortadan kaldırma örneği.

@ConfigurationProperties ile Gruplanmış Property'ler

`@Value`'nun aksine, `@ConfigurationProperties` aynı önekle (prefix) başlayan bütün bir property ailesini tek, tipli bir nesneye bağlar:

```
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.boot.context.properties.EnableConfigurationProperties;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.env.ConfigurableEnvironment;
import org.springframework.core.env.MapPropertySource;

import java.util.Map;

// @ConfigurationProperties groups a whole family of related settings into one
// typed object, bound from a common prefix -- unlike @Value, which reads one
// property at a time with no structure of its own.
@ConfigurationProperties(prefix = "app.mail")
```

```
class MailProperties {
    private String host = "localhost";
    private int port = 25;
    private boolean tlsEnabled = false;

    public String getHost() { return host; }
    public void setHost(String host) { this.host = host; }

    public int getPort() { return port; }
    public void setPort(int port) { this.port = port; }

    public boolean isTlsEnabled() { return tlsEnabled; }
    public void setTlsEnabled(boolean tlsEnabled) { this.tlsEnabled = tlsEnabled; }

    @Override
    public String toString() {
        return "MailProperties{host='" + host + "', port=" + port + ", tlsEnabled=" +
tlsEnabled + "}";
    }
}

@Configuration
@EnableConfigurationProperties(MailProperties.class)
class MailConfig {
    // Note: no PropertySourcesPlaceholderConfigurer needed here, unlike the
    // @Value example -- @ConfigurationProperties binds directly from the
    // Environment's property sources, it does not go through the ${...}
    // embedded value resolver mechanism @Value relies on.
}

class ConfigurationPropertiesExample {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext();
        ConfigurableEnvironment environment = context.getEnvironment();
        // "tls-enabled" (kebab-case, as it would appear in application.yml)
        // binds to the "tlsEnabled" field automatically -- Spring Boot's
        // relaxed binding rules treat the two as the same property.
        environment.getPropertySources().addFirst(new MapPropertySource("test", Map.of(
                "app.mail.host", "smtp.example.com",
                "app.mail.port", "587",
                "app.mail.tls-enabled", "true"
        )));
        context.register(MailConfig.class);
        context.refresh();

        System.out.println(context.getBean(MailProperties.class));
        // MailProperties{host='smtp.example.com', port=587, tlsEnabled=true}

        context.close();
    }
}
```

`app.mail.tls-enabled` (YAML'da kullanılacağı gibi kebab-case) otomatik olarak `tlsEnabled` alanına bağlanıyor -- Spring Boot'un "relaxed binding" (esnek bağlama) kuralları, kebab-case, camelCase ve UPPER_SNAKE_CASE'i (ortam değişkenleri için) aynı property olarak

kabul eder. Bu proje henüz kendi `@ConfigurationProperties` sınıfını tanımlamıyor -- "Bu Projenin Kendi application.yml ve Config Sınıfları" bölümünde buna tekrar döneceğiz.

@ConfigurationProperties Validasyonu

Gerçek projelerde `@ConfigurationProperties`, `jakarta.validation` annotation'ları (`@NotBlank`, `@Min` gibi) ve `@Validated` ile doğrulanır -- bu, `spring-boot-starter-validation` bağımlılığını gerektirir, ki bu projede yok (bkz. `pom.xml`). Aynı güvenliği elle, `@PostConstruct` ile kuruyoruz:

```
import jakarta.annotation.PostConstruct;
import org.springframework.beans.factory.BeanCreationException;
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.boot.context.properties.EnableConfigurationProperties;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.env.ConfigurableEnvironment;
import org.springframework.core.env.MapPropertySource;

import java.util.Map;

// Real Spring Boot projects validate @ConfigurationProperties with
// jakarta.validation annotations (@NotBlank, @Min...) plus @Validated --
// that needs the spring-boot-starter-validation dependency, which this
// project doesn't have. We get the same safety net by hand instead, with a
// @PostConstruct check that fails fast at startup instead of silently
// running with a broken configuration.
@ConfigurationProperties(prefix = "app.retry")
class RetryProperties {
    private int maxAttempts = 3;
    private long backoffMillis = 500;

    public int getMaxAttempts() { return maxAttempts; }
    public void setMaxAttempts(int maxAttempts) { this.maxAttempts = maxAttempts; }

    public long getBackoffMillis() { return backoffMillis; }
    public void setBackoffMillis(long backoffMillis) { this.backoffMillis = backoffMillis; }

    @PostConstruct
    void validate() {
        if (maxAttempts < 1) {
            throw new IllegalStateException("app.retry.max-attempts must be at least 1, was
" + maxAttempts);
        }
        if (backoffMillis < 0) {
            throw new IllegalStateException("app.retry.backoff-millis cannot be negative,
was " + backoffMillis);
        }
    }
}

@Configuration
@EnableConfigurationProperties(RetryProperties.class)
class RetryConfig {
}
```

```
class ConfigurationPropertiesValidationExample {
    public static void main(String[] args) {
        // Case 1: a valid configuration -- starts up normally.
        AnnotationConfigApplicationContext validContext = new
AnnotationConfigApplicationContext();
        ConfigurableEnvironment validEnv = validContext.getEnvironment();
        validEnv.getPropertySources().addFirst(new MapPropertySource("test", Map.of("app.
retry.max-attempts", "5")));
        validContext.register(RetryConfig.class);
        validContext.refresh();
        System.out.println(validContext.getBean(RetryProperties.class).getMaxAttempts());
        // 5
        validContext.close();

        // Case 2: an invalid configuration -- @PostConstruct fails fast at
        // startup instead of the application running with a nonsensical
        // "0 retries" setting. Spring wraps the exception our own code threw
        // inside a BeanCreationException, the same as it would for any other
        // failing @PostConstruct method (see the Spring IoC Container lesson).
        AnnotationConfigApplicationContext invalidContext = new
AnnotationConfigApplicationContext();
        ConfigurableEnvironment invalidEnv = invalidContext.getEnvironment();
        invalidEnv.getPropertySources().addFirst(new MapPropertySource("test", Map.of("app.
retry.max-attempts", "0")));
        invalidContext.register(RetryConfig.class);
        try {
            invalidContext.refresh();
        } catch (BeanCreationException e) {
            System.out.println("Startup failed: " + e.getRootCause().getMessage());
            // Startup failed: app.retry.max-attempts must be at least 1, was 0
        }
    }
}
```

Geçersiz bir `max-attempts` değeri, uygulamanın "0 deneme hakkı" gibi anlamsız bir durumla sessizce çalışmaya devam etmesi yerine, başlangıçta (`context.refresh()` sırasında) açıkça başarısız oluyor -- Spring IoC Container dersindeki `@PostConstruct` / `@PreDestroy` bölümünde gördüğümüz yaşam döngüsü kancasının, burada "fail fast" (erken başarısız ol) için kullanılmış hâli.

Profiles: @Profile ile Ortama Özel Bean'ler

`@Profile`, aynı arayüzün birbirinden tamamen farklı iki implementasyonunun kaynak kodda yan yana durmasını, ama yalnızca birinin -- aktif profile göre -- gerçekten kaydedilmesini sağlar:

```
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;

interface PaymentGateway {
    void charge(double amount);
}

// @Profile lets two completely different bean implementations exist side by
// side in the source code, with only one of them ever actually registered --
// chosen by which profile(s) are active. This is exactly how this project
// switches between application-dev.yml, application-test.yml, and
// application-prod.yml.
@Configuration
class PaymentConfig {

    @Bean
    @Profile("dev")
    PaymentGateway sandboxPaymentGateway() {
        return amount -> System.out.println("[sandbox] Pretending to charge $" + amount);
    }

    @Bean
    @Profile("prod")
    PaymentGateway realPaymentGateway() {
        return amount -> System.out.println("[real] Charging $" + amount + " via the
payment provider");
    }
}

class ProfileExample {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext devContext = new
AnnotationConfigApplicationContext();
        devContext.getEnvironment().setActiveProfiles("dev");
        devContext.register(PaymentConfig.class);
        devContext.refresh();
        devContext.getBean(PaymentGateway.class).charge(49.99);
        // [sandbox] Pretending to charge $49.99
        devContext.close();

        AnnotationConfigApplicationContext prodContext = new
AnnotationConfigApplicationContext();
        prodContext.getEnvironment().setActiveProfiles("prod");
        prodContext.register(PaymentConfig.class);
        prodContext.refresh();
        prodContext.getBean(PaymentGateway.class).charge(49.99);
        // [real] Charging $49.99 via the payment provider
        prodContext.close();
    }
}
```

Bu, tam olarak bu projenin `application-dev.yml`, `application-test.yml` ve `application-prod.yml` arasında geçiş yaparken kullandığı mekanizma -- yalnızca property değerleri değil, bean'lerin kendisi bile ortama göre değişebilir.

Profile'a Özel application-{profile}.yml Dosyaları

Bu projenin dört `application*.yml` dosyası var: temel ayarları içeren `application.yml`, ve üç profile özel dosya. `application.yml`'deki `spring.profiles.active: dev` satırı, hangi profilin varsayılan olarak aktif olacağını belirler:

```
# application-dev.yml
spring:
  datasource:
    url: jdbc:postgresql://localhost:5433/learning
  jpa:
    show-sql: true

# application-prod.yml
spring:
  datasource:
    url: ${DB_URL}
  jpa:
    show-sql: false
```

`application-prod.yml`'de `${DB_URL}` gibi ifadeler, "External Configuration: Property Kaynaklarının Öncelik Sırası" bölümünde göreceğimiz ortam değişkenlerinden okunuyor -- gizli bilgiler (veritabanı şifresi gibi) hiçbir zaman repoya yazılmıyor. Aktif profil `spring.profiles.active` ile (ya da bir ortam değişkeniyle) değiştirildiğinde, Spring Boot ilgili `application-{profile}.yml` dosyasını temel `application.yml`'in üzerine katman katman uygular.

External Configuration: Property Kaynaklarının Öncelik Sırası

Bir property birden fazla kaynakta tanımlıysa (mesela hem `application.yml`'de hem bir ortam değişkeninde), Spring Boot hangisinin kazanacağına sıkı bir öncelik sırasıyla karar verir. En yüksek öncelikliden en düşüğe doğru başlıca kaynaklar: komut satırı argümanları, ortam değişkenleri, `application-{profile}.yml`, ve en altta temel `application.yml`. Bu sıralamayı kendi elimizle simüle edelim:

```
import org.springframework.core.env.ConfigurableEnvironment;
import org.springframework.core.env.MapPropertySource;
import org.springframework.core.env.MutablePropertySources;
import org.springframework.core.env.StandardEnvironment;

import java.util.Map;

// Spring Boot reads configuration from many places at once -- command-line
// arguments, environment variables, application-{profile}.yml,
// application.yml, and more -- and needs a strict priority order to pick a
// winner when more than one source defines the same key. We simulate three
// of those sources by hand here, added in *reverse* priority order, to watch
// the highest-priority one win.
class PropertySourceOrderExample {
    public static void main(String[] args) {
        ConfigurableEnvironment environment = new StandardEnvironment();
        MutablePropertySources sources = environment.getPropertySources();

        // Lowest priority: the base application.yml.
        sources.addLast(new MapPropertySource("application.yml", Map.of("server.port",
"8080")));

        // Higher priority: a profile-specific application-prod.yml.
        sources.addBefore("application.yml", new MapPropertySource("application-prod.yml",
Map.of("server.port", "9090")));

        // Highest priority in this example: an environment variable (in a
        // real deployment this would come from the OS itself, via
        // StandardEnvironment's own built-in "systemEnvironment" source).
        sources.addFirst(new MapPropertySource("systemEnvironment", Map.of("server.port",
"443")));

        System.out.println(environment.getProperty("server.port"));
        // 443

        // Remove the environment variable to see the next source in line win.
        sources.remove("systemEnvironment");
        System.out.println(environment.getProperty("server.port"));
        // 9090

        sources.remove("application-prod.yml");
        System.out.println(environment.getProperty("server.port"));
        // 8080
    }
}
```

Bu, tam olarak "Profile'a Özel application-{profile}.yml Dosyaları" bölümünde gördüğümüz `${DB_URL}` ifadesinin neden işe yaradığını açıklıyor: prodüksiyonda gerçek bir ortam değişkeni, `application-prod.yml`'deki yer tutucunun üzerine katmanlanıyor.

Ortam Değişkenleri ve Komut Satırı Argümanları

Property kaynaklarının en yüksek öncelikli ikisi -- ortam değişkenleri ve komut satırı argümanları -- koddan tamamen bağımsız, dağıtım zamanında belirlenir. Bir Spring Boot

uygulaması `java -jar app.jar --server.port=9090` şeklinde başlatılırsa, bu değer `application.yml`'deki her şeyin önüne geçer; aynı şekilde `SERVER_PORT=9090` ortam değişkeni de (Spring Boot, `SERVER_PORT`'u otomatik olarak `server.port`'a çevirir) aynı etkiyi yapar. Bu, sırrı (veritabanı şifresi gibi) hiç repoya yazmadan, sadece dağıtım ortamında enjekte etmenin standart yoludur -- `application-prod.yml`'deki `${DB_URL}`, `${DB_USERNAME}`, `${DB_PASSWORD}` tam olarak bunu yapıyor.

ApplicationEvent ve @EventListener

Container, kendi yaşam döngüsü boyunca event'ler yayınlar, ve senin kendi sınıfların da kendi event'lerini yayınlayıp dinleyebilir -- yayınlayan ile dinleyen arasında hiçbir doğrudan bağımlılık olmadan:

```
import org.springframework.context.ApplicationEvent;
import org.springframework.context.ApplicationEventPublisher;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.event.ContextRefreshedEvent;
import org.springframework.context.event.EventListener;
import org.springframework.stereotype.Component;

// A custom application event -- any object extending ApplicationEvent (or,
// since Spring 4.2, any arbitrary object at all) can be published and picked
// up by listeners, completely decoupling the publisher from whoever reacts
// to it.
class OrderPlacedEvent extends ApplicationEvent {
    private final String orderId;

    OrderPlacedEvent(Object source, String orderId) {
        super(source);
        this.orderId = orderId;
    }

    String getOrderId() {
        return orderId;
    }
}

@Component
class OrderService {
    private final ApplicationEventPublisher publisher;

    OrderService(ApplicationEventPublisher publisher) {
        this.publisher = publisher;
    }

    void placeOrder(String orderId) {
        System.out.println("Order placed: " + orderId);
        publisher.publishEvent(new OrderPlacedEvent(this, orderId));
    }
}

@Component
```

```
class OrderNotificationListener {

    // @EventListener is the modern, annotation-based alternative to
    // implementing ApplicationListener<OrderPlacedEvent> directly -- both
    // work, this one needs no interface at all.
    @EventListener
    void onOrderPlaced(OrderPlacedEvent event) {
        System.out.println("Sending confirmation email for order " + event.getOrderId());
    }

    // The container itself publishes events too -- ContextRefreshedEvent
    // fires once the ApplicationContext has finished starting up. In a full
    // Spring Boot app, ApplicationReadyEvent is the equivalent "everything is
    // completely ready" signal, fired after ContextRefreshedEvent, once the
    // embedded server has also started (see "Spring Boot's Own Events").
    @EventListener
    void onContextRefreshed(ContextRefreshedEvent event) {
        System.out.println("Application context is ready.");
    }
}

@Configuration
@ComponentScan
class AppConfig {
}

class ApplicationEventExample {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext
(AppConfig.class);
        // Application context is ready.

        context.getBean(OrderService.class).placeOrder("ORD-1001");
        // Order placed: ORD-1001
        // Sending confirmation email for order ORD-1001

        context.close();
    }
}
```

`@EventListener`, `ApplicationListener<T>` arayüzünü implemente etmenin modern, annotation tabanlı alternatifi -- hiçbir arayüz gerekmiyor, metot imzasındaki parametre tipi hangi event'in dinleneceğini belirliyor. `ContextRefreshedEvent`, container'ın kendi yayınladığı event'lerden biri; bir sonraki bölümde Spring Boot'un bunun üzerine eklediği kendi event'lerine bakıyoruz.

Spring Boot'un Kendi Event'leri (Kısa Bakış)

Saf Spring IoC Container'ın `ContextRefreshedEvent`'ine ek olarak, Spring Boot `SpringApplication.run(...)` sırasında kendi event zincirini de yayınlar: `ApplicationStartingEvent` (en başta), `ApplicationEnvironmentPreparedEvent` (Environment hazırlandığında, ama context henüz oluşmadan), `ApplicationContextInitializedEvent`, `ApplicationPreparedEvent`, ardından container'ın kendi `ContextRefreshedEvent`'i, ve en sonda

`ApplicationReadyEvent` -- "her şey, gömülü sunucu (embedded server) da dahil, tamamen hazır" sinyali. Bu event'ler yalnızca gerçek `SpringApplication.run(...)` ile başlatılan bir uygulamada oluşur -- bu dersteki örneklerin kullandığı sade `AnnotationConfigApplicationContext` bunları tetiklemez, bu yüzden burada ayrı bir kod örneği yok. Pratikte en sık kullanılan ikisi `ApplicationReadyEvent` (arka plan işlerini başlatmak için) ve `ApplicationFailedEvent`'tir (başlatma başarısız olduğunda temizlik yapmak için).

Bu Projenin Kendi application.yml ve Config Sınıfları

Bu projenin `application.yml`'i, auto-configuration'ın gerçek hayatta nasıl kullanıldığının iyi bir örneği: `spring.datasource.*`, `spring.jpa.*`, `spring.thymeleaf.*`, `spring.flyway.*` anahtarlarının hiçbiri elle yazılmış bir `@Bean` metoduna karşılık gelmiyor -- hepsi, ilgili auto-configuration sınıflarının (`DataSourceAutoConfiguration`, `JpaBaseConfiguration`, `ThymeleafAutoConfiguration`, `FlywayAutoConfiguration`) okuduğu, önceden tanımlı property'ler. Projenin kendi yazdığı tek `@Configuration` sınıfı `WebConfig` (Spring IoC Container dersinde gördüğümüz), ve o da bir `LocaleResolver` bean'i tanımlıyor -- Spring Boot'un kendi `LocaleResolver` auto-configuration'ının **yerine** geçiyor, çünkü `WebMvcAutoConfiguration`'ın kendi `localeResolver` bean'i tam olarak `@ConditionalOnMissingBean` ile korunuyor (bkz. "@ConditionalOnMissingBean: Kullanıcı Kendi Bean'ini Tanımladıysa"). Projede henüz hiçbir `@Value` ya da `@ConfigurationProperties` kullanılmıyor -- tüm ayarlar, Spring Boot'un kendi auto-configuration sınıflarının doğrudan okuduğu standart `spring.*` / `server.*` anahtarları.

Best Practices

- **Auto-configuration'ı önce anla, sonra güven** -- hangi bean'in neden kaydedildiğini bilmeden "sihir" gibi görmek, bir şey beklenmedik çalıştığında hata ayıklamayı imkânsız hâle getirir (bkz. "@Conditional Ailesi ve Auto-Configuration Mekanizması").
- **Property gruplarını `@ConfigurationProperties` ile, tekil değerleri `@Value` ile oku** -- birbiriyle ilişkili birden fazla ayar varsa, tek tek `@Value` yerine gruplanmış bir sınıf çok daha bakımı kolay bir yaklaşımdır (bkz. "@ConfigurationProperties ile Gruplanmış Property'ler").
- **Sırları (şifre, API anahtarı) asla `application.yml`'e yazma, ortam değişkenlerinden oku** -- bu projenin `application-prod.yml`'i tam olarak bunu yapıyor (bkz. "Ortam Değişkenleri ve Komut Satırı Argümanları").
- **`@ConditionalOnMissingBean` ile korunan varsayılanları geçersiz kılmak için, aynı tipte kendi bean'ini tanımlamak yeterlidir** -- ekstra bir "kapat" anahtarı aramana gerek yok (bkz. "@ConditionalOnMissingBean: Kullanıcı Kendi Bean'ini Tanımladıysa").
- **`@ConfigurationProperties` ile gelen ayarları başlangıçta doğrula, çalışma zamanında değil** -- geçersiz bir ayarla sessizce çalışmak yerine erken ve açıkça başarısız olmak, hatayı üretimde değil başlangıçta yakalar (bkz. "@ConfigurationProperties Validasyonu").

Yaygın Hatalar

1. `@Value("${...}")`'in saf Spring IoC Container'da (Spring Boot olmadan) otomatik çalışacağını sanmak. `PropertySourcesPlaceholderConfigurer` bean'i elle tanımlanmadan `${...}` yer tutucuları hiç çözümlenmez (bkz. "@Value ile Tekil Property Enjeksiyonu").

2. `@ConfigurationProperties` sınıfını yazıp `@EnableConfigurationProperties` (ya da `@ConfigurationPropertiesScan`) eklemeyi unutmak. Sınıfın kendisi `@Component` değildir -- container'a "bunu bağla" demeden hiçbir bean oluşmaz (bkz. "@ConfigurationProperties ile Gruplanmış Property'ler").

3. `@ConditionalOnProperty`'de `matchIfMissing`'i unutmak. Varsayılan davranış (`matchIfMissing = false`) property hiç tanımlanmamışsa bean'i **kaydetmemektir** -- "varsayılan olarak açık" bir özellik istiyorsan bunu açıkça belirtmen gerekir (bkz. "Kendi Auto-Configuration'ımızı Yazmak").

4. `@Profile` ile korunan bir bean'i, o profil aktif değilken `getBean(...)` ile almaya çalışmak. Bean hiç kaydedilmediği için bu, `NoSuchBeanDefinitionException` ile sonuçlanır -- Component Scanning dersindeki `@Component` eklenmemiş bir sınıfla aynı sonuç (bkz. "Profiles: @Profile ile Ortama Özel Bean'ler").

5. Property kaynaklarının önceliğini yanlış hatırlamak, ve "neden ortam değişkenim `application.yml`'i geçersiz kılmıyor" diye şaşırmak. Ortam değişkenleri `application.yml`'den her zaman daha yüksek öncelikli olmalı -- eğer geçersiz kılmıyorsa, muhtemelen değişken adı yanlış yazılmıştır (bkz. "External Configuration: Property Kaynaklarının Öncelik Sırası").

6. `ApplicationReadyEvent` gibi Spring Boot'a özel bir event'i, sade bir `AnnotationConfigApplicationContext` ile test etmeye çalışmak. Bu event'ler yalnızca gerçek `SpringApplication.run(...)` ile tetiklenir -- `ContextRefreshedEvent` ile karıştırılmamalı (bkz. "Spring Boot'un Kendi Event'leri (Kısa Bakış)").

Özet, Cheat Sheet ve Terimler Sözlüğü

Auto-configuration, Spring Boot'un classpath'teki kütüphanelere bakarak senin yerine bean kaydetmesidir; `@Value` ve `@ConfigurationProperties`, `application.yml`'deki ayarları Java koduna taşımanın iki yolu; `@Profile` ortama göre farklı bean'ler seçmeyi, `ApplicationEvent`/`@EventListener` ise container'ın (ve senin kendi kodunun) birbirine gevşek bağlı şekilde haber vermesini sağlar. Önemli noktalar:

- `@SpringBootApplication` = `@SpringBootConfiguration` + `@EnableAutoConfiguration` + `@ComponentScan`
- `@ConditionalOnClass`/`@ConditionalOnMissingBean`/`@ConditionalOnProperty`: auto-configuration'ın bean kaydedip kaydetmeyeceğine karar verdiği koşullar
- `@Value("${key:default}")`: tekil property, isteğe bağlı varsayılan değerle

- `@ConfigurationProperties(prefix = "...")` + `@EnableConfigurationProperties`: gruplanmış, tipli property ailesi
- `@Profile("name")`: yalnızca belirtilen profil aktifken kaydedilen bean
- Property kaynak önceliği (yüksekten düşüğe): komut satırı argümanları > ortam değişkenleri > `application-{profile}.yml` > `application.yml`
- `ApplicationEvent` + `ApplicationEventPublisher` + `@EventListener`: yayıncı ile dinleyici arasında doğrudan bağımlılık olmadan haberleşme

Hızlı referans:

```
@SpringBootApplication  // = @SpringBootConfiguration + @EnableAutoConfiguration +
@ComponentScan
class MyApplication { }

@Configuration
class MyAutoConfiguration {
    @Bean
    @ConditionalOnClass(name = "some.library.Class")
    @ConditionalOnMissingBean
    @ConditionalOnProperty(name = "app.feature.enabled", havingValue = "true",
matchIfMissing = false)
    MyBean myBean() { return new MyBean(); }
}

class MyService {
    @Value("${app.setting:default}")
    private String setting;
}

@ConfigurationProperties(prefix = "app.settings")
class MySettings {
    private String name;
    // getter/setter
}

@Configuration
@EnableConfigurationProperties(MySettings.class)
class SettingsConfig {
    @Bean
    @Profile("prod")
    MyBean prodBean() { return new MyBean(); }
}

@Component
class MyListener {
    @EventListener
    void onEvent(MyEvent event) { }
}
```

Terimler Sözlüğü

Auto-configuration — Spring Boot'un, classpath'teki kütüphanelere bakarak senin yerine bean kaydetmesi.

`@SpringBootApplication` — `@SpringBootConfiguration`, `@EnableAutoConfiguration` ve `@ComponentScan`'i tek bir annotation'da birleştiren kolaylık annotation'ı.

`@Conditional` — Bir bean'in ya da `@Configuration` sınıfının, belirli bir koşul sağlandığında (ya da sağlanmadığında) kaydedilmesini sağlayan annotation ailesinin temeli.

`@ConditionalOnClass` — Belirtilen sınıf classpath'te varsa bean'i kaydeden koşul.

`@ConditionalOnMissingBean` — Belirtilen tipte başka bir bean henüz yoksa kaydeden koşul; kütüphane varsayılanlarının kullanıcı tanımlı bean'lere yenilmesini sağlar.

`@ConditionalOnProperty` — Belirtilen property belirli bir değere sahipse (ya da hiç yoksa `matchIfMissing`'e göre) bean'i kaydeden koşul.

`@Value` — `application.yml`'den tek bir property'yi bir alana/parametreye enjekte eden annotation.

`@ConfigurationProperties` — Ortak bir önekle başlayan bir property ailesini tek, tipli bir nesneye bağlayan annotation.

`@Profile` — Bir bean'in yalnızca belirtilen profil(ler) aktifken kaydedilmesini sağlayan annotation.

`ApplicationEvent` — Container tarafından ya da uygulama kodu tarafından yayınlanabilen, `@EventListener` / `ApplicationListener` ile dinlenebilen olay nesnesi.

Ek: Mini Proje — Feature Toggle Sistemi

Bu mini proje, `@ConfigurationProperties` (bir feature flag ailesi) ile `@ConditionalOnProperty` (tek bir flag'in bütün bir bean'in var olup olmayacağına karar vermesi) ve Component Scanning dersindeki `@Primary`'yi bir araya getiriyor:

```
import org.springframework.boot.autoconfigure.condition.ConditionalOnProperty;
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.boot.context.properties.EnableConfigurationProperties;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Primary;

import java.util.HashMap;
import java.util.Map;

// Mini project: a small feature-toggle system, tying together
// @ConfigurationProperties (a whole family of on/off switches, grouped under
// one prefix) with @ConditionalOnProperty (one specific feature deciding, at
// startup, whether an entire bean should exist at all).
@ConfigurationProperties(prefix = "app.features")
class FeatureToggles {
    private Map<String, Boolean> flags = new HashMap<>();

    public Map<String, Boolean> getFlags() { return flags; }
    public void setFlags(Map<String, Boolean> flags) { this.flags = flags; }

    boolean isEnabled(String feature) {
        return flags.getOrDefault(feature, false);
    }
}

interface RecommendationEngine {
    String recommend(String userId);
}

@Configuration
@EnableConfigurationProperties(FeatureToggles.class)
class FeatureToggleConfig {

    // Registered unconditionally -- always available, whatever the feature
    // flags say.
    @Bean
    RecommendationEngine basicRecommendationEngine() {
        return userId -> "Popular items for you, " + userId;
    }

    // Registered only when the property is explicitly turned on. @Primary
    // (from the Component Scanning lesson) resolves the ambiguity when both
    // beans exist: the AI engine wins any plain-type injection whenever it's
    // present at all.
    @Bean
    @Primary
    @ConditionalOnProperty(name = "app.features.ai-recommendations", havingValue = "true")
    RecommendationEngine aiRecommendationEngine() {
        return userId -> "AI-personalized picks for you, " + userId;
    }
}
```

```
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.core.env.ConfigurableEnvironment;
import org.springframework.core.env.MapPropertySource;

import java.util.Map;

class FeatureToggleDemo {
    public static void main(String[] args) {
        // Two independent property paths under the same "app.features" prefix:
        // "flags.ai-recommendations" binds into FeatureToggles' Map field (for
        // the application's own bookkeeping/UI), while the flat
        // "ai-recommendations" key separately drives @ConditionalOnProperty
        // on the bean itself. They happen to carry the same value here, but
        // they are two different mechanisms answering two different questions.

        // Case 1: AI recommendations turned off -- only the basic engine exists.
        AnnotationConfigApplicationContext offContext = new
AnnotationConfigApplicationContext();
        ConfigurableEnvironment offEnv = offContext.getEnvironment();
        offEnv.getPropertySources().addFirst(new MapPropertySource("test", Map.of(
                "app.features.flags.ai-recommendations", "false",
                "app.features.ai-recommendations", "false"
        )));
        offContext.register(FeatureToggleConfig.class);
        offContext.refresh();
        System.out.println(offContext.getBean(FeatureToggles.class).isEnabled("ai-
recommendations"));
        // false
        System.out.println(offContext.getBean(RecommendationEngine.class).recommend("user-
42"));
        // Popular items for you, user-42
        offContext.close();

        // Case 2: AI recommendations turned on -- both beans exist, @Primary
        // decides which one wins the ambiguous injection.
        AnnotationConfigApplicationContext onContext = new
AnnotationConfigApplicationContext();
        ConfigurableEnvironment onEnv = onContext.getEnvironment();
        onEnv.getPropertySources().addFirst(new MapPropertySource("test", Map.of(
                "app.features.flags.ai-recommendations", "true",
                "app.features.ai-recommendations", "true"
        )));
        onContext.register(FeatureToggleConfig.class);
        onContext.refresh();
        System.out.println(onContext.getBean(FeatureToggles.class).isEnabled("ai-
recommendations"));
        // true
        System.out.println(onContext.getBean(RecommendationEngine.class).recommend("user-
42"));
        // AI-personalized picks for you, user-42
        onContext.close();
    }
}
```

`app.features.flags.*` altındaki her anahtar `FeatureToggles` bean'inin `flags` map'ine bağlanırken, `app.features.ai-recommendations` bambaşka bir mekanizmayla --

`@ConditionalOnProperty` ile -- bir bean'in var olup olmayacağına karar veriyor. İkisi aynı prefix'i paylaşsa da, birbirinden tamamen bağımsız iki yol: biri bir Java nesnesine bağlanan veri, diğeri container'ın kendisine "bu bean'i hiç oluşturma" diyen bir koşul.

> `aiRecommendationEngine()` üzerindeki `@Primary`, tam olarak Component Scanning dersindeki "@Primary: Varsayılan Aday Belirlemek" bölümünde gördüğümüz mekanizma -- iki bean aynı anda var olduğunda, hangisinin varsayılan olarak kazanacağını belirliyor.

Ek: Mini Proje — Bildirim Ayarları Yöneticisi

Son mini proje, bu dersin neredeyse tüm konularını bir araya getiriyor: `@ConfigurationProperties` ile gruplanmış ayarlar, `@Profile` ile ortama özel davranış, ve ayarlar yüklendiğinde yayınlanan bir `ApplicationEvent`:

```
import jakarta.annotation.PostConstruct;
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.boot.context.properties.EnableConfigurationProperties;
import org.springframework.context.ApplicationEvent;
import org.springframework.context.ApplicationEventPublisher;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;
import org.springframework.context.event.EventListener;
import org.springframework.stereotype.Component;

// Mini project: a notification settings manager that ties together most of
// this lesson at once -- grouped settings via @ConfigurationProperties,
// environment-specific overrides via @Profile, and an event published once
// the settings are loaded, so other beans can react without depending on
// this one directly.
@ConfigurationProperties(prefix = "app.notifications")
class NotificationSettings {
    private int retryAttempts = 3;
    private long timeoutMillis = 2000;

    public int getRetryAttempts() { return retryAttempts; }
    public void setRetryAttempts(int retryAttempts) { this.retryAttempts = retryAttempts; }

    public long getTimeoutMillis() { return timeoutMillis; }
    public void setTimeoutMillis(long timeoutMillis) { this.timeoutMillis = timeoutMillis; }

    @Override
    public String toString() {
        return "NotificationSettings{retryAttempts=" + retryAttempts + ", timeoutMillis=" +
timeoutMillis + "}";
    }
}

class SettingsLoadedEvent extends ApplicationEvent {
    private final NotificationSettings settings;
```

```
    SettingsLoadedEvent(Object source, NotificationSettings settings) {
        super(source);
        this.settings = settings;
    }

    NotificationSettings getSettings() {
        return settings;
    }
}

@Component
class SettingsLoader {
    private final NotificationSettings settings;
    private final ApplicationEventPublisher publisher;

    SettingsLoader(NotificationSettings settings, ApplicationEventPublisher publisher) {
        this.settings = settings;
        this.publisher = publisher;
    }

    @PostConstruct
    void publishOnceLoaded() {
        publisher.publishEvent(new SettingsLoadedEvent(this, settings));
    }
}

@Component
class SettingsAuditListener {
    @EventListener
    void onSettingsLoaded(SettingsLoadedEvent event) {
        System.out.println("Settings loaded: " + event.getSettings());
    }
}

@Configuration
@EnableConfigurationProperties(NotificationSettings.class)
@ComponentScan
class NotificationSettingsConfig {

    // A more patient retry policy only in production, layered on top of the
    // defaults from application.yml -- @Profile deciding between two
    // completely different Runnable strategies, the same idea as
    // PaymentConfig earlier in this lesson.
    @Bean
    @Profile("prod")
    Runnable slowRetryWarning() {
        return () -> System.out.println("Production mode: retries will be slower and more
patient.");
    }

    @Bean
    @Profile("!prod")
    Runnable fastRetryWarning() {
        return () -> System.out.println("Non-production mode: retries are fast, for quicker
feedback.");
    }
}
```

```
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.core.env.ConfigurableEnvironment;
import org.springframework.core.env.MapPropertySource;

import java.util.Map;

class NotificationSettingsDemo {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext();
        ConfigurableEnvironment environment = context.getEnvironment();
        environment.getPropertySources().addFirst(new MapPropertySource("test", Map.of(
                "app.notifications.retry-attempts", "5",
                "app.notifications.timeout-millis", "5000"
        )));
        environment.setActiveProfiles("prod");
        context.register(NotificationSettingsConfig.class);
        context.refresh();
        // Settings loaded: NotificationSettings{retryAttempts=5, timeoutMillis=5000}

        context.getBean(Runnable.class).run();
        // Production mode: retries will be slower and more patient.

        context.close();
    }
}
```

`SettingsLoader`, `@PostConstruct` ile (Spring IoC Container dersindeki "@PostConstruct ve @PreDestroy" bölümü) ayarlar enjekte edildikten hemen sonra bir `SettingsLoadedEvent` yayınlıyor -- `SettingsAuditListener` bu event'i, `SettingsLoader` sınıfının varlığından bile haberdar olmadan dinliyor. `prod` profili aktifken `slowRetryWarning` bean'i, başka herhangi bir profilde (`!prod`) ise `fastRetryWarning` bean'i kaydediliyor -- ikisi asla aynı anda var olmuyor.

> `@Profile("!prod")` gibi olumsuzlama ifadeleri kullanışlıdır, ama dikkatli kullanılmalı: `dev` , `test` , ya da hiçbir profil aktif değilken bile `!prod` koşulu sağlanır -- "prod olmayan her durum" ile "yalnızca dev" aynı şey değildir, ve bu ikisini karıştırmak yanlış bean'in yanlış ortamda kaydedilmesine yol açabilir.