Spring Boot Auto-Configuration & Properties

@SpringBootApplication, the @Conditional family (@ConditionalOnClass /@ConditionalOnMissingBean/@ConditionalOnProperty), @Value, @ConfigurationProperties, @Profile, and ApplicationEvent/@EventListener.

Advanced 55 min

Last updated: August 11, 2026
Read online: https://www.learnforgex.com/en/topics/autoconfiguration-properties

---

Spring Boot Auto-Configuration & Properties

In the Component Scanning lesson we saw how beans get found by the container; in the Spring IoC Container lesson, how beans get defined and what their lifecycle looks like. In this final lesson, we look at the third layer Spring Boot adds on top of both: how `@SpringBootApplication` and auto-configuration work behind the scenes, reading properties from `application.yml` with `@Value` / `@ConfigurationProperties`, environment-specific configuration with `@Profile`, and the container announcing things to itself with `ApplicationEvent`. By the end of this lesson, you'll know exactly what this project's own `application.yml` files and the single `@SpringBootApplication` line on `LearningPlatformApplication` actually represent.

What Is Spring Boot Auto-Configuration?

Auto-configuration is Spring Boot looking at which libraries are present on the classpath and registering beans on your behalf -- without you writing a single `@Bean` method. For example, because this project depends on `spring-boot-starter-data-jpa` and `postgresql`, Spring Boot automatically sets up a `DataSource` bean, an `EntityManagerFactory` bean, and a JPA `TransactionManager` bean -- none of which we ever defined by hand in a `@Configuration` class like `WebConfig`:

```
// If we wrote it by hand (we never do -- Spring Boot does this for us):
@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;
    }
}
```

Other than setting the `spring.datasource.*` keys in `application.yml`, you've never seen a class like the one above -- because auto-configuration sees `org.postgresql.Driver` and `spring-boot-starter-data-jpa` on the classpath and registers this bean for you.

Why Does It Exist?

In the Component Scanning lesson we saw that Java Config is repetitive, and that component scanning reduces this by moving the bean definition onto the class itself. But component scanning only works for **your own classes** -- beans like `DataSource`, `EntityManagerFactory`, or `RequestMappingHandlerMapping` are classes you didn't write, from third-party libraries; you can't add `@Component` to them (see the Component Scanning lesson's "Component Scanning vs. Java Config: Which One, When?").

Without auto-configuration, every new Spring Boot project would need dozens of hand-written `@Bean` methods -- for `DataSource`, `TransactionManager`, `RequestMappingHandlerMapping`, `ViewResolver`, `ObjectMapper`, and more. Auto-configuration moves the assumption "if this library is on the classpath, you probably need these beans" into the framework itself -- you only customize those defaults with a handful of properties in `application.yml`.

History

Spring Boot launched in 2014 with version 1.0 -- until then, setting up a Spring application could take hours, whether through XML-based configuration (the `ClassPathXmlApplicationContext` era we mentioned in the Spring IoC Container lesson's "History" section) or dozens of hand-written `@Bean` methods. Spring Boot's core promise was "convention over configuration": start with sensible defaults, write something only when you need to deviate from them.

`@EnableAutoConfiguration` (and `@SpringBootApplication`, which wraps it) is the technical foundation of that promise. It originally read the list of auto-configuration classes from a `META-INF/spring.factories` file; in Spring Boot 2.7 (2022) this mechanism moved to the faster, more explicit `META-INF/spring/org.springframework.boot.autoconfigure.`

`AutoConfiguration.imports` file -- since this project uses Spring Boot 4.1 (see `pom.xml`), it uses the newer mechanism. `@Conditional` derivatives like `@ConditionalOnClass` and `@ConditionalOnMissingBean` have also been the foundation of auto-configuration since 1.0.

@SpringBootApplication: A Combination of Three Annotations

The single `@SpringBootApplication` annotation on `LearningPlatformApplication` is actually a combination of three separate annotations -- we already recognize two of them from earlier lessons:

```
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` is a specialized derivative of `@Configuration` (Spring IoC Container lesson). `@ComponentScan` is the very same annotation we saw in the Component Scanning lesson -- used with no arguments, it scans its own package (and subpackages), which is why every `@Controller` / `@Service` under `com.cdurgun.learning` is found without being registered by hand. The third is `@EnableAutoConfiguration`, the actual subject of this lesson.

The @Conditional Family and How Auto-Configuration Works

At the heart of auto-configuration is the `@Conditional` family: annotations that let a bean, or an entire `@Configuration` class, be registered only when (or only when not) a certain condition holds. When `@EnableAutoConfiguration` is processed, Spring Boot tries each of the hundreds of `@Configuration` classes in its own `spring-boot-autoconfigure` module (`DataSourceAutoConfiguration`, `JpaRepositoriesAutoConfiguration`, `ThymeleafAutoConfiguration` , and so on) in turn -- each one is guarded by its own `@Conditional` annotations, and nothing whose condition fails gets registered. We'll use the two most common derivatives -- `@ConditionalOnClass` and `@ConditionalOnMissingBean` -- with our own hands in the next two sections.

@ConditionalOnClass: When a Class Is on the Classpath

`@ConditionalOnClass` says "only register this bean if the given class is on the classpath" -- the same mechanism the real `DataSourceAutoConfiguration` uses to only kick in when a JDBC driver is actually among the project's dependencies:

```
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` really is on the classpath (Jackson comes in transitively through `spring-boot-starter-web`), so the first bean is registered; the second bean, guarded by a made-up class name, is silently skipped -- no exception is thrown, the bean simply behaves as if it never existed.

@ConditionalOnMissingBean: When the Application Defines Its Own Bean

`@ConditionalOnMissingBean` lets libraries say "I'll offer a sensible default, but use your own bean if you define one" -- in real Spring Boot, many beans like `ObjectMapper` and `RestTemplateBuilder` behave exactly this way:

```
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();
    }
}
```

Order matters here: the application's own `@Configuration` class has to be processed **before** the class defining the library's default -- in real Spring Boot, this is guaranteed by auto-

configuration classes always being processed **after** the application's own `@Configuration` classes, which is exactly why a bean you define yourself always wins over auto-configuration's default.

Writing Our Own Auto-Configuration

Let's see, at small scale, what a real Spring Boot starter looks like on the inside -- `@ConditionalOnProperty` lets a whole feature be switched on or off from `application.yml`:

```
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` means the bean stays **off by default** if the property is never set at all -- the same behavior as many optional real Spring Boot features (`spring.cache.type`, `management.endpoints.web.exposure.include`, and others): it never kicks in unless you explicitly ask for it.

application.properties and application.yml

Spring Boot supports two equivalent file formats: `application.properties`, made up of flat `key=value` lines, and `application.yml`, which expresses nested structure with indentation. This project prefers YAML -- a fragment from its own `application.yml`:

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

server:
  port: 8080
```

The same settings in `.properties` format would look like: `spring.application.name=learning-platform`, `spring.profiles.active=dev`, `spring.thymeleaf.cache=false`, `server.port=8080`. Both resolve to the same flat, dot-separated property keys (like `spring.thymeleaf.cache`) -- YAML just lets you write it with less repetition, through nested indentation. In the following sections we'll see how to read these keys on the Java side with `@Value` and `@ConfigurationProperties`.

Injecting a Single Property with @Value

`@Value` injects a single property from `application.yml` directly into a field or constructor parameter -- the simplest way to read one, but with no grouping at all:

```
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();
    }
}
```

The `:Hello` part of `${app.greeting.prefix:Hello}` specifies the default value to use if the property is never set -- so the application doesn't crash just because an optional property was left out. As the comment in the code example notes, in plain Spring IoC Container (without Spring Boot), you have to define a `PropertySourcesPlaceholderConfigurer` bean by hand for `${...}` placeholders to work at all -- in Spring Boot you never write this yourself, because `@EnableAutoConfiguration` registers it for you automatically. This is exactly the kind of repetition-removal we mentioned in "Why Does It Exist?".

Grouped Properties with @ConfigurationProperties

Unlike `@Value`, `@ConfigurationProperties` binds an entire family of properties sharing the same prefix into one typed object:

```
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` (kebab-case, as it would appear in YAML) is automatically bound to the `tlsEnabled` field -- Spring Boot's "relaxed binding" rules treat kebab-case, camelCase,

and UPPER_SNAKE_CASE (for environment variables) as the same property. This project doesn't yet define its own `@ConfigurationProperties` class -- we'll come back to that in "This Project's Own application.yml and Config Classes".

Validating @ConfigurationProperties

Real projects validate `@ConfigurationProperties` with `jakarta.validation` annotations (`@NotBlank`, `@Min`, and so on) plus `@Validated` -- that requires the `spring-boot-starter-validation` dependency, which this project doesn't have (see `pom.xml`). We get the same safety net by hand, with `@PostConstruct`:

```
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
        }
    }
}
```

An invalid `max-attempts` value fails loudly at startup (during `context.refresh()`), instead of the application silently running with a nonsensical "0 retries" configuration -- the same `@PostConstruct` / `@PreDestroy` lifecycle hook from the Spring IoC Container lesson, used here to "fail fast."

Profiles: Environment-Specific Beans with @Profile

`@Profile` lets two completely different implementations of the same interface sit side by side in the source code, with only one of them actually registered -- chosen by whichever profile is active:

```
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();
    }
}
```

This is exactly the mechanism this project uses to switch between `application-dev.yml`, `application-test.yml`, and `application-prod.yml` -- not just property values, but even the beans themselves can change based on the environment.

Profile-Specific application-{profile}.yml Files

This project has four `application*.yml` files: a base `application.yml` with shared settings, and three profile-specific files. The `spring.profiles.active: dev` line in `application.yml` determines which profile is active by default:

```
# 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
```

Expressions like `${DB_URL}` in `application-prod.yml` are read from environment variables, as we'll see in "External Configuration: The Priority Order of Property Sources" -- secrets (like a database password) are never written into the repo at all. When the active profile is changed (via `spring.profiles.active` or an environment variable), Spring Boot layers the matching `application-{profile}.yml` file on top of the base `application.yml`.

External Configuration: The Priority Order of Property Sources

If a property is defined in more than one place at once (say, both in `application.yml` and in an environment variable), Spring Boot has a strict priority order to decide which one wins. From highest to lowest priority, the main sources are: command-line arguments, environment variables, `application-{profile}.yml`, and at the bottom, the base `application.yml`. Let's simulate that ordering by hand:

```
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
    }
}
```

This is exactly what explains why the `${DB_URL}` expression from "Profile-Specific application-{profile}.yml Files" works at all: in production, a real environment variable is layered on top of the placeholder in `application-prod.yml`.

Environment Variables and Command-Line Arguments

The two highest-priority property sources -- environment variables and command-line arguments -- are completely independent of the code, decided at deployment time. If a

Spring Boot application is started with `java -jar app.jar --server.port=9090`, that value overrides everything in `application.yml`; an environment variable of `SERVER_PORT=9090` has the exact same effect (Spring Boot automatically translates `SERVER_PORT` into `server.port`). This is the standard way to inject a secret (like a database password) without ever writing it into the repo, only into the deployment environment -- exactly what `${DB_URL}`, `${DB_USERNAME}`, and `${DB_PASSWORD}` in `application-prod.yml` do.

ApplicationEvent and @EventListener

The container publishes events throughout its own lifecycle, and your own classes can publish and listen to their own events too -- with no direct dependency between the publisher and the listener:

```
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` is the modern, annotation-based alternative to implementing `ApplicationListener<T>` directly -- no interface needed at all; the parameter type on the method signature determines which event is being listened for. `ContextRefreshedEvent` is one of the container's own published events; the next section looks at what Spring Boot adds on top of it.

Spring Boot's Own Events (A Quick Look)

On top of plain Spring IoC Container's `ContextRefreshedEvent`, Spring Boot publishes its own chain of events during `SpringApplication.run(...)`: `ApplicationStartingEvent` (at the very start), `ApplicationEnvironmentPreparedEvent` (once the Environment is ready, but before the context itself exists), `ApplicationContextInitializedEvent`, `ApplicationPreparedEvent`, then the container's own `ContextRefreshedEvent`, and finally `ApplicationReadyEvent` -- the

"everything, including the embedded server, is completely ready" signal. These events only occur in an application actually started with `SpringApplication.run(...)` -- the plain `AnnotationConfigApplicationContext` used by this lesson's examples never triggers them, which is why there's no separate code example here. In practice, the two most commonly used are `ApplicationReadyEvent` (for kicking off background work) and `ApplicationFailedEvent` (for cleanup when startup fails).

This Project's Own application.yml and Config Classes

This project's `application.yml` is a good example of what auto-configuration looks like in real life: none of the `spring.datasource.*`, `spring.jpa.*`, `spring.thymeleaf.*`, or `spring.flyway.*` keys correspond to a hand-written `@Bean` method -- they are all predefined properties read by the relevant auto-configuration classes (`DataSourceAutoConfiguration`, `JpaBaseConfiguration`, `ThymeleafAutoConfiguration`, `FlywayAutoConfiguration`). The only `@Configuration` class this project writes itself is `WebConfig` (from the Spring IoC Container lesson), which defines a `LocaleResolver` bean -- **replacing** Spring Boot's own `LocaleResolver` auto-configuration, because `WebMvcAutoConfiguration`'s own `localeResolver` bean is guarded by exactly `@ConditionalOnMissingBean` (see "@ConditionalOnMissingBean: When the Application Defines Its Own Bean"). The project doesn't use `@Value` or `@ConfigurationProperties` anywhere yet -- every setting is a standard `spring.*`/`server.*` key read directly by Spring Boot's own auto-configuration classes.

Best Practices

- **Understand auto-configuration before trusting it** -- treating it as pure "magic" without knowing which bean is registered and why makes debugging an unexpected outcome nearly impossible (see "The @Conditional Family and How Auto-Configuration Works").
- **Read groups of properties with `@ConfigurationProperties`, single values with `@Value`** -- when several related settings exist together, a grouped class is far easier to maintain than a pile of individual `@Value` fields (see "Grouped Properties with @ConfigurationProperties").
- **Never write secrets (passwords, API keys) into `application.yml` -- read them from environment variables instead** -- this project's own `application-prod.yml` does exactly that (see "Environment Variables and Command-Line Arguments").
- **To override a default guarded by `@ConditionalOnMissingBean`, defining your own bean of the same type is enough** -- there's no need to look for a separate "turn it off" switch (see "@ConditionalOnMissingBean: When the Application Defines Its Own Bean").
- **Validate `@ConfigurationProperties` settings at startup, not at runtime** -- failing early and loudly on an invalid setting catches the mistake at startup instead of in production (see "Validating @ConfigurationProperties").

Common Mistakes

1. Assuming `@Value("${...}")` works automatically in plain Spring IoC Container (without Spring Boot). Without a hand-registered `PropertySourcesPlaceholderConfigurer` bean, `${...}` placeholders are never resolved at all (see "Injecting a Single Property with @Value").

2. Writing a `@ConfigurationProperties` class and forgetting `@EnableConfigurationProperties` (or `@ConfigurationPropertiesScan`). The class itself is not a `@Component` -- no bean is created unless you explicitly tell the container to bind it (see "Grouped Properties with @ConfigurationProperties").

3. Forgetting `matchIfMissing` on `@ConditionalOnProperty`. The default behavior (`matchIfMissing = false`) is to **not** register the bean when the property is never set at all -- if you want a feature that's "on by default," you have to state that explicitly (see "Writing Our Own Auto-Configuration").

4. Trying to `getBean(...)` a bean guarded by `@Profile` while that profile isn't active. Since the bean was never registered, this results in a `NoSuchBeanDefinitionException` -- the same outcome as an unannotated class in the Component Scanning lesson (see "Profiles: Environment-Specific Beans with @Profile").

5. Misremembering the priority order of property sources, and being surprised that "my environment variable isn't overriding application.yml." An environment variable should always outrank `application.yml` -- if it isn't, the variable name is probably misspelled (see "External Configuration: The Priority Order of Property Sources").

6. Trying to test a Spring-Boot-specific event like `ApplicationReadyEvent` with a plain `AnnotationConfigApplicationContext`. These events only fire with a real `SpringApplication.run(...)` -- don't confuse them with `ContextRefreshedEvent` (see "Spring Boot's Own Events (A Quick Look)").

Summary, Cheat Sheet, and Glossary

Auto-configuration is Spring Boot registering beans on your behalf by looking at which libraries are on the classpath; `@Value` and `@ConfigurationProperties` are two ways to bring `application.yml` settings into Java code; `@Profile` picks different beans based on the environment, while `ApplicationEvent` / `@EventListener` let the container (and your own code) communicate loosely. Key points:

- `@SpringBootApplication` = `@SpringBootConfiguration` + `@EnableAutoConfiguration` + `@ComponentScan`
- `@ConditionalOnClass` / `@ConditionalOnMissingBean` / `@ConditionalOnProperty`: the conditions auto-configuration uses to decide whether to register a bean
- `@Value("${key:default}")`: a single property, with an optional default value

- `@ConfigurationProperties(prefix = "...")` + `@EnableConfigurationProperties`: a grouped, typed family of properties
- `@Profile("name")`: a bean registered only while the given profile is active
- Property source priority (highest to lowest): command-line arguments > environment variables > `application-{profile}.yml` > `application.yml`
- `ApplicationEvent` + `ApplicationEventPublisher` + `@EventListener`: communication between a publisher and a listener with no direct dependency between them

Quick reference:

```
@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) { }
}
```

Glossary

Auto-configuration — Spring Boot registering beans on your behalf by looking at which libraries are present on the classpath.

`@SpringBootApplication` — The convenience annotation combining `@SpringBootConfiguration`, `@EnableAutoConfiguration`, and `@ComponentScan` into one.

`@Conditional` — The base of the annotation family that lets a bean or a `@Configuration` class be registered only when (or only when not) a given condition holds.

`@ConditionalOnClass` — A condition that registers a bean only if the given class is on the classpath.

`@ConditionalOnMissingBean` — A condition that registers a bean only if no other bean of the given type exists yet; lets library defaults yield to user-defined beans.

`@ConditionalOnProperty` — A condition that registers a bean only if a given property has a specific value (or is missing, depending on `matchIfMissing`).

`@Value` — The annotation that injects a single property from `application.yml` into a field or parameter.

`@ConfigurationProperties` — The annotation that binds a whole family of properties sharing a common prefix into one typed object.

`@Profile` — The annotation that registers a bean only while the given profile(s) are active.

`ApplicationEvent` — An event object that can be published by the container or by application code, and listened to with `@EventListener` / `ApplicationListener`.

Appendix: Mini Project — A Feature Toggle System

This mini project brings together `@ConfigurationProperties` (a family of feature flags), `@ConditionalOnProperty` (one specific flag deciding whether an entire bean exists at all), and `@Primary` from the Component Scanning lesson:

```
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();
    }
}
```

Every key under `app.features.flags.*` binds into the `FeatureToggles` bean's `flags` map, while `app.features.ai-recommendations` decides, through a completely different mechanism

-- `@ConditionalOnProperty` -- whether a bean exists at all. Even though they share a prefix, these are two entirely independent paths: one is data bound into a Java object, the other is a condition telling the container "don't even create this bean."

> The `@Primary` on `aiRecommendationEngine()` is exactly the mechanism from the Component Scanning lesson's "@Primary: Marking a Default Candidate" section -- when both beans exist at once, it decides which one wins by default.

Appendix: Mini Project — A Notification Settings Manager

The final mini project ties together almost everything from this lesson: `@ConfigurationProperties` for grouped settings, `@Profile` for environment-specific behavior, and an `ApplicationEvent` published once the settings are loaded:

```
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` publishes a `SettingsLoadedEvent` right after the settings are injected, using `@PostConstruct` (the Spring IoC Container lesson's "@PostConstruct and @PreDestroy" section) -- `SettingsAuditListener` listens for that event without even knowing `SettingsLoader` exists. While the `prod` profile is active, the `slowRetryWarning` bean is registered; in any other profile (`!prod`), it's `fastRetryWarning` instead -- the two never exist at the same time.

> Negation expressions like `@Profile("!prod")` are useful, but need care: the `!prod` condition holds while `dev` , `test` , or even no profile at all is active -- "every case that isn't prod" is not the same thing as "only dev," and confusing the two can register the wrong bean in the wrong environment.