Spring Boot Configuration & Internals

In diesem Modul

- Spring Boot AutoConfiguration (+ Deep Dive)
- Externalisierte Configuration (+ Demo)
- Eigene Starter/Autoconfiguration
- @ConfigurationPropertiesScan vs. @EnableConfigurationProperties
- Kubernetes-native Konfiguration (ConfigMaps, Secrets)

Einführung in AutoConfiguration

- **Convention over Configuration:** Entwickler sollen so wenig wie möglich konfigurieren müssen.
- **Opinionated Defaults:** Spring Boot trifft sinnvolle Annahmen basierend auf den vorhandenen Libraries (Classpath).
- **Startzeit-Optimierung:** Statt XML-Konfiguration werden Beans dynamisch nur dann erzeugt, wenn sie wirklich gebraucht werden.

Wie funktioniert es? (High Level)

Beim Start der Anwendung scannt Spring Boot:

1. **Classpath:** Ist z.B. `H2` in den Dependencies? -> Dann konfiguriere eine H2 DataSource.
2. **Existing Beans:** Hat der User schon eine eigene `DataSource` definiert? -> Dann mache nichts.
3. **Properties:** Steht in `application.properties` ein bestimmter Schalter?

Config Sources & Reihenfolge (Precedence)

Spring Boot liest Konfigurationen aus vielen Quellen. Die wichtigsten (überschreibend von oben nach unten):

1. **Devtools global settings** (`~/.spring-boot-devtools.properties`)
2. **@TestPropertySource** (in Tests)
3. **Command line arguments** (z.B. `--server.port=9000`)
4. **OS Environment Variables**
5. **Application Properties** (JAR extern)
6. **Application Properties** (JAR intern)

(Insgesamt gibt es über 15 Quellen!)

Immutable Configuration (@ConstructorBinding)

Der "moderne" Weg für Type-Safe Configuration (seit Spring Boot 2.2+ / 3.0 verfügbar).

```
@ConfigurationProperties(prefix = "app.mail")
public record MailProperties(String host, int port, boolean enabled) {
    // Records sind automatisch immutable und haben Konstruktoren
}
```

Aktivierung:

Entweder `@EnableConfigurationProperties(MailProperties.class)` auf einer Konfigurationsklasse oder (neu in Boot 2.2) einfach `@ConfigurationPropertiesScan`.

Profiles

Spring lädt automatisch Dateien basierend auf dem aktiven Profil:

- `application.yml` (Immer geladen)
- `application-dev.yml` (Überschreibt Werte, wenn Profil `dev` aktiv)

Man kann Profile bündeln. In `application.yml`:

```yaml
spring:
  profiles:
    group:
      production:
      - proddb
      - cloudmetrics
      - k8s
```

Aktiviert man nun `-Dspring.profiles.active=production`, werden automatisch `proddb`, `cloudmetrics` und `k8s` mitaktiviert.

Config Server (Git-Backed)

- Trennung von Config und Code: zentrale Git-Repo, Versionierung, Audits.
- Clients ziehen Config pro `spring.profiles.active` vom Server; können sie auch zur Laufzeit refreshen.
- Mehrere Apps, gemeinsame Defaults: `application.yml`, `my-service.yml`, Profil-Dateien wie `my-service-dev.yml`.

Server (Spring Cloud Config Server):

```yaml
spring:
  application:
    name: config-server
  cloud:
    config:
      server:
        git:
          uri: https://github.com/example/config-repo
          search-paths: config
server:
  port: 8888
```

Starter: `spring-cloud-config-server` + `@EnableConfigServer`.

Client-Anbindung

- Seit Boot 2.4: `spring.config.import=optional:configserver:http://localhost:8888`
- Client greift beim Start auf `/{app-name}/{profile}` zu (z.B. `/my-service/dev`).

`application.yml` (Client):

```yaml
spring:
  application:
    name: my-service
  config:
    import: "optional:configserver:http://localhost:8888"
```

In Git-Repo: `my-service-dev.yml`:

```yaml
server:
  port: 8085
custom:
  feature-x-enabled: true
```

Verschlüsselung von Properties

- Config Server kann Secrets serverseitig entschlüsseln.
- Symmetrischer Key im Server (`encrypt.key`) oder besser KMS/HSM.

`application.yml` (Server):

```
encrypt:
  key: my-strong-password
```

Verschlüsselung per CLI/Post:

`curl -X POST localhost:8888/encrypt -d 'db-pass'` → `'{cipher}...`

Im Git-Repo: `password: "{cipher}..."`

Client erhält bereits entschlüsselte Werte.

Speichern von Secrets in externen Systemen

Beispiel Spring Cloud Vault (`application.yml`):

```yaml
spring:
  cloud:
    vault:
      uri: http://localhost:8200
      authentication: token
      token: s.1234567890
      kv:
        enabled: true
        backend: secret
        application-name: my-service
```

- Secrets aus `secret/my-service` werden als Properties verfügbar, z.B. `secret.datasource.password`.
- Empfohlen: Vault-Agent/K8s Auth statt statischem Token; Maskierung sensibler Werte in Logs/Actuator.

Vault lokal ausführen und testen

1. Vault im Dev-Mode starten (lokal, *nicht* für Prod):
   `vault server -dev -dev-root-token-id=root`
2. In neuer Shell den Token setzen:
   `export VAULT_ADDR=http://127.0.0.1:8200`
   `export VAULT_TOKEN=root`
3. Secret schreiben:
   `vault kv put secret/my-service datasource.password=superSecret api.key=abc123`
4. Spring Boot starten (mit oben gezeigtem `application.yml`):
   `./mvnw spring-boot:run` oder `./gradlew bootRun`
5. Im Code eine Property prüfen, z.B. `@Value("${datasource.password}")`.
6. Live ändern:
   `vault kv put secret/my-service datasource.password=newSecret` → `/actuator/refresh` aufrufen, dann den neuen Wert zeigen.

Vault-Auth im Cluster

In Kubernetes kann man Vault auch im Cluster betreiben.

- **Vault Agent + K8s Auth**: Pod erhält ein ServiceAccount-Token, Vault tauscht es gegen ein kurzlebiges Vault-Token. Secrets landen als Datei/ENV oder werden per Template gerendert.
- Vorteil: Kein statischer Token im Image oder `application.yml`; kurzlebige Tokens, Audit-Log, feingranulare Policies.

Resilienz bei Config-Fehlern

- **Fail Fast**: Bei Spring Cloud Config ist es im Produktivbetrieb sinnvoll, den Start abbrechen zu lassen, wenn der Server nicht erreichbar ist (`spring.cloud.config.fail-fast=true`).
- **Fallback**: Für Dev/CI oder lokale Demos `spring.cloud.config.fail-fast=false` + `spring.config.import=optional:configserver:...` → App startet mit lokalen Defaults.
- **Retry**: `spring.cloud.config.retry.*` aktivieren, damit bei kurzzeitigem Ausfall (Config-Server, Vault) erneut versucht wird.
- **Vault-Ausfall**: `spring.cloud.vault.fail-fast=false` für lokale Demos, sonst true + Retry; sensibel loggen, Secrets nicht dumpen.
- **Circuit Breaker fürs Laden** (Hinweis): Wenn Config/Secrets über HTTP APIs nachgeladen werden, kann ein Resilience4j-CircuitBreaker + Retry/Backoff verhindern, dass Start/Refresh unendlich hängt.

Spring Boot Starter

Ein eigener Starter kapselt wiederkehrende Logik (z.B. Standard-Logging, Security-Wrapper).

Struktur

Best Practice ist ein Multi-Module Maven/Gradle Projekt:

1. `my-feature-autoconfigure`: Enthält den Code und die Config.
2. `my-feature-starter`: Leer, definiert nur `my-feature-autoconfigure` als Dependency.

@Conditional Annotationen

Bedingte Ausführung sind das Zentrum der Auto-Konfiguration.

- `@ConditionalOnClass(name = "com.example.Service")`: Nur wenn Klasse im Classpath.
- `@ConditionalOnMissingBean`: Nur wenn der User keine eigene Bean dieses Typs gebaut hat.
- `@ConditionalOnProperty(prefix = "app", name = "enabled", havingValue = "true")`: Nur wenn Property gesetzt.
- `@ConditionalOnWebApplication`: Nur wenn es eine Web-App ist.

Beispiel: AutoConfiguration Klasse

```
@AutoConfiguration // Alias für @Configuration in Boot 2.7+
@ConditionalOnClass(AuditService.class)
@EnableConfigurationProperties(AuditProperties.class)
public class AuditAutoConfiguration {

    @Bean
    @ConditionalOnMissingBean
    public AuditService auditService(AuditProperties props) {
        return new AuditService(props.getUrl());
    }
}
```

`META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports`

```
com.mycompany.audit.AuditAutoConfiguration
```

Boot Internals: EnvironmentPostProcessor

Wenn man in den Startprozess eingreifen muss, **bevor** der ApplicationContext erstellt wird (z.B. um custom Config-Dateien zu laden oder sensitive Properties zu entschlüsseln).

```
public class MyEnvPostProcessor implements EnvironmentPostProcessor {
    @Override
    public void postProcessEnvironment(ConfigurableEnvironment environment,
                                       SpringApplication application) {
        // Logik hier, z.B. PropertySources hinzufügen
        Map<String, Object> map = new HashMap<>();
        map.put("extra.property", "value");
        environment.getPropertySources()
                   .addLast(new MapPropertySource("myExtra", map));
    }
}
```

Registrierung in `META-INF/spring/org.springframework.boot.env.EnvironmentPostProcessor.imports`.

Failure Analyzers

Eigene Fehlermeldungen beim Absturz direkt nach dem Start.

```
public class PortInUseFailureAnalyzer
        extends AbstractFailureAnalyzer<PortInUseException> {

    @Override
    protected FailureAnalysis analyze(Throwable rootFailure,
                                      PortInUseException cause) {
        return new FailureAnalysis(
            "Port " + cause.getPort() + " is already in use.",
            "Please identify and stop the process or change the port.",
            cause);
    }
}
```

Dies verwandelt einen Stacktrace in eine besser lesbare Fehlermeldung in der Konsole.

Configuration Properties: Scan vs. Enable

@ConfigurationPropertiesScan vs. @EnableConfigurationProperties

Zwei Wege, um `@ConfigurationProperties`-Klassen zu aktivieren:

Annotation	Verwendung
`@EnableConfigurationProperties(MyProps.class)`	Explizit einzelne Klassen registrieren
`@ConfigurationPropertiesScan`	Automatisch alle im Package scannen

@EnableConfigurationProperties

Explizite Registrierung – volle Kontrolle, aber mehr Boilerplate.

```
@Configuration
@EnableConfigurationProperties({
    MailProperties.class,
    StorageProperties.class
})
public class AppConfig {
    // Jede neue Properties-Klasse muss hier hinzugefügt werden
}
```

Vorteil: Klar ersichtlich, welche Properties aktiv sind.

Nachteil: Vergisst man eine Klasse, wird sie nicht gebunden.

@ConfigurationPropertiesScan

Automatisches Scanning – weniger Boilerplate, wie `@ComponentScan`.

```
@SpringBootApplication
@ConfigurationPropertiesScan("com.example.config")
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

Vorteil: Neue Properties-Klassen werden automatisch erkannt.

Nachteil: Weniger explizit; kann unerwartete Beans erzeugen.

Empfehlung

Szenario	Empfehlung
Kleine Anwendung, wenige Props	`@ConfigurationPropertiesScan`
Library / Starter	`@EnableConfigurationProperties` (explizit)
Strikte Kontrolle erforderlich	`@EnableConfigurationProperties`
Viele Properties-Klassen	`@ConfigurationPropertiesScan`

Kubernetes-native Konfiguration

Rückblick: Externalized Configuration

Wir haben bereits **Spring Cloud Config** und **Vault** als externe Konfigurationsquellen kennengelernt.

In Kubernetes-Umgebungen gibt es eine **native Alternative**: ConfigMaps und Secrets direkt importieren – ohne zusätzliche Infrastruktur wie Config Server.

spring.config.import für Kubernetes

Seit Spring Boot 2.4+ können ConfigMaps und Secrets **direkt** importiert werden – ohne Spring Cloud Kubernetes.

```
spring:
  config:
    import:
      - "optional:configtree:/etc/config/"
```

configtree: Liest Dateien aus einem Verzeichnis als Properties.

Jede Datei wird zum Property-Namen, der Inhalt zum Wert.

Kubernetes Volume Mount

```
# Kubernetes Deployment
spec:
  containers:
    - name: app
      volumeMounts:
        - name: config-volume
          mountPath: /etc/config
  volumes:
    - name: config-volume
      configMap:
        name: my-app-config
```

ConfigMap:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-app-config
data:
  database.url: "jdbc:postgresql://db:5432/mydb"
  cache.enabled: "true"
```

Resultat im Spring Boot

Die Dateien unter `/etc/config/` werden zu Properties:

Datei	Property
`/etc/config/database.url`	`database.url`
`/etc/config/cache.enabled`	`cache.enabled`

```
@Value("${database.url}")
private String dbUrl;  // "jdbc:postgresql://db:5432/mydb"
```

Secrets als Config importieren

Für sensible Daten funktioniert es identisch mit Kubernetes Secrets:

```
spring:
  config:
    import:
      - "optional:configtree:/etc/secrets/"
```

```
# Kubernetes Secret (base64-encoded)
apiVersion: v1
kind: Secret
metadata:
  name: db-credentials
data:
  spring.datasource.password: c3VwZXJTZWNyZXQ=  # "superSecret"
```

Vorteil: Keine zusätzlichen Dependencies, nativer Kubernetes-Support.