Class-33: Externalized Configuration in Spring Boot

by Pial Kanti Samadder

Topics Covered

- Configuration file formats: **properties vs YAML**
- **@Value** vs **@ConfigurationProperties**
- **Profile-specific configuration**
- **Environment variable overrides**
- Best practices for real-world applications

Why Externalized Configuration?

- Decouples configuration from compiled code
- Allows deployment across multiple environments without code changes
- Improves maintainability and readability
- Supports secure and dynamic configuration via environment variables
- Works seamlessly in cloud-native and containerized environments

Configuration Sources Hierarchy

Spring Boot resolves configuration from multiple sources in a defined order.

Common Sources

- `application.properties` / `application.yml`
- Profile-specific files (e.g., `application-dev.yml`)
- Environment variables
- Command-line arguments
- System properties
- External configuration files / Config Server

Higher-priority sources override lower ones.

`application.properties` vs `application.yml`

application.properties

- Simple key-value format
- Best for small or flat configurations
- Example:

```
server.port=8080
logging.level.root=INFO
```

application.yml

- Hierarchical and more readable
- Ideal for complex, nested structures
- Example:

```
server:
  port: 8080
logging:
  level:
    root: INFO
```

Recommendation

Use **YAML** for most modern Spring Boot applications due to clarity and structure.

Injecting Configuration with `@Value`

Injects a single configuration value into a field.

Example

```
@Value("${app.version}")
private String version;
```

Advantages

- Simple and fast
- Good for occasional or isolated values

Limitations

- Not type-safe
- Harder to group related configuration
- Can become messy at scale

Strongly-Typed Configuration with @ConfigurationProperties

Binds groups of related properties into a structured POJO.

Example

```
@ConfigurationProperties(prefix = "app")
public class AppProperties {
    private String name;
    private int timeout;
}
```

Benefits

- Type-safe and validated
- Supports hierarchical configuration
- Cleaner and more maintainable
- Easily testable

@Value vs @ConfigurationProperties

Criteria	@Value	@ConfigurationProperties
Single simple values	✔️	—
Large grouped configs	—	✔️
Type safety	❌	✔️
Supports validation	❌	✔️
Readability at scale	❌	✔️
Recommended for production	Limited use	✔️

Access Environment Variables via Environment Class

Spring's Environment allows programmatic access to configuration values, including environment variables.

Example

```
@Autowired
private Environment env;

String dbUrl = env.getProperty("spring.datasource.url");
String port  = env.getProperty("SERVER_PORT");
```

Key Points

- getProperty() reads from **all config sources**
- Env variables **override application.yml**
- Useful for **dynamic or conditional configurations**

Profile-Specific Configuration

Different environments require different settings (dev, test, prod).

Profile Files

- `application-dev.yml`
- `application-test.yml`
- `application-prod.yml`

Activating Profiles

- Via `application.properties`:

```
spring.profiles.active=dev
```

- Via CLI:

```
java -jar app.jar --spring.profiles.active=prod
```

Use Cases

- Dev vs prod database URLs
- Logging level variations
- Feature toggles

Environment Variable Overrides

Spring Boot allows environment variables to override values defined in `application.yml`.
This makes configuration flexible and secure during deployment.

How Values Are Mapped

Spring Boot converts a property key into an environment variable automatically:

Property Key	Environment Variable
`spring.datasource.url`	`SPRING_DATASOURCE_URL`
`server.port`	`SERVER_PORT`

Conversion rules:

- Lowercase → Uppercase
- Dots (`.`) → Underscores (`_`)

Why It's Important

- Change configuration **without modifying code or files**
- Provide **secrets** (DB passwords, API keys) securely
- Works naturally with **Docker, Kubernetes, and cloud platforms**
- Allows different settings for **dev / test / prod**

Default Values with Environment Variables

Using @Value

```
@Value("${app.timeout:30}")  // Uses 30 if env variable APP_TIMEOUT not set
private int timeout;
```

- `${PROPERTY:default}` syntax works for **environment variables** too.
- Spring automatically maps env vars to property keys (uppercase + underscores).

Using @ConfigurationProperties

```
@ConfigurationProperties(prefix = "app")
public class AppConfig {

    private int timeout = 30;      // Default if APP_TIMEOUT not set
    private String name = "MyApp"; // Default if APP_NAME not set

    // getters & setters
}
```

- Default values in the POJO **apply if the env variable or property is missing**.

Using Environment

```
@Autowired
private Environment env;

int timeout = Integer.parseInt(env.getProperty("APP_TIMEOUT", "30"));
```

- `getProperty(key, defaultValue)` allows dynamic defaults
- Works with environment variables or any config source.

Summary

Approach	Default Value Support	Use Case
`@Value`	`${PROPERTY:default}`	Single value
`@ConfigurationProperties`	Field initialization	Grouped/structured config
`Environment`	`getProperty(key, default)`	Dynamic runtime values

Configuration Priority Order (High → Low)

1. **Command-line arguments**
2. **Environment variables**
3. **Profile-specific application files**
4. `application.yml` / `application.properties`
5. **Default properties**

This ensures you can override configuration at deployment time easily.

Best Practices for Externalized Configuration

- Prefer **YAML** for structured configuration
- Group related settings under a prefix
- Use **@ConfigurationProperties** for maintainability
- Store secrets using environment variables or Vault
- Keep environment-specific files minimal and clear
- Avoid hardcoding default values inside classes
- Use profiles to isolate dev/test/prod behavior