Custom Configuration in Spring Applications

Trainer Name:

Shreyansh Kumar

Introduction

Configuration management is a critical aspect of any Spring application. As applications grow in complexity, the need for organized, flexible, and environment-specific configuration becomes increasingly important. Spring Framework provides robust mechanisms for handling configuration properties, allowing developers to externalize configuration and make applications more maintainable.

This document explores how to implement custom configurations in Spring applications using both application.properties files and custom configuration files. We'll dive deep into various approaches, best practices, and provide practical examples to illustrate these concepts.

Understanding Spring Configuration Basics

Before we delve into custom configurations, let's establish a foundation by understanding how Spring handles configuration properties by default.

Spring Boot applications typically use application.properties (or application.yml) files to configure various aspects of the application. These files are automatically detected when placed in standard locations:

- Inside the application's classpath
- In the current directory
- In a /config subdirectory in the current directory
- In the /config directory directly under the application's root

When the application starts, Spring Boot loads these properties and makes them available throughout the application.

Custom Configuration in application.properties

Basic Property Configuration

The simplest form of custom configuration involves adding your own properties to the application.properties file. Let's look at an example:

```
# Standard Spring properties
spring.application.name=my-application
server.port=8080

# Custom properties
app.feature.enabled=true
app.cache.timeout=3600
app.service.url=https://api.example.com
app.max-connections=100
```

Accessing Properties in Spring Components

To access these properties in your Spring components, you can use the @Value annotation:

```java
@RestController
@RequestMapping("/api")
public class ApiController {

    @Value("${app.feature.enabled}")
    private boolean featureEnabled;

    @Value("${app.service.url}")
    private String serviceUrl;

    @GetMapping("/status")
    public String getStatus() {
        if (featureEnabled) {
            return "Feature is enabled. Service URL: " + serviceUrl;
        }
        return "Feature is disabled";
    }
}
```

Using ConfigurationProperties

For more structured configuration, Spring Boot provides the @ConfigurationProperties annotation, which allows binding configuration properties to Java objects:

```java
@Configuration
@ConfigurationProperties(prefix = "app")
public class AppProperties {

    private boolean featureEnabled;
    private int cacheTimeout;
    private String serviceUrl;
    private int maxConnections;

    // Getters and setters
    public boolean isFeatureEnabled() {
        return featureEnabled;
    }

    public void setFeatureEnabled(boolean featureEnabled) {
        this.featureEnabled = featureEnabled;
    }

    public int getCacheTimeout() {
        return cacheTimeout;
    }
    public void setCacheTimeout(int cacheTimeout) {
        this.cacheTimeout = cacheTimeout;
    }

    public String getServiceUrl() {
        return serviceUrl;
    }

    public void setServiceUrl(String serviceUrl) {
        this.serviceUrl = serviceUrl;
    }

    public int getMaxConnections() {
        return maxConnections;
    }

    public void setMaxConnections(int maxConnections) {
        this.maxConnections = maxConnections;
    }
}
```

Then, you can inject and use this configuration class in your components:

```java
@Service
public class DataService {

    private final AppProperties appProperties;

    public DataService(AppProperties appProperties) {
        this.appProperties = appProperties;
    }

    public void processData() {
        if (appProperties.isFeatureEnabled()) {
            // Use the configuration
            System.out.println("Processing with timeout: " +
appProperties.getCacheTimeout());
            System.out.println("Connecting to: " +
appProperties.getServiceUrl());
            System.out.println("Max connections: " +
appProperties.getMaxConnections());
        }
    }
}
```

Nested Properties

You can also define nested properties in your application.properties:

```
app.database.url=jdbc:mysql://localhost:3306/mydb
app.database.username=admin
app.database.password=secret
app.database.pool.max-size=20
app.database.pool.min-idle=5
```

And bind them to nested classes:

```java
@Configuration
@ConfigurationProperties(prefix = "app")
public class AppProperties {

    private Database database = new Database();

    // Getter and setter for database
    public Database getDatabase() {
        return database;
    }

    public void setDatabase(Database database) {
        this.database = database;
    }

    public static class Database {
        private String url;
        private String username;
        private String password;
        private Pool pool = new Pool();

        // Getters and setters
        public String getUrl() {
            return url;
        }

        public void setUrl(String url) {
            this.url = url;
        }

        public String getUsername() {
            return username;
        }

        public void setUsername(String username) {
            this.username = username;
        }

        public String getPassword() {
            return password;
        }
        public void setPassword(String password) {
            this.password = password;
        }

        public Pool getPool() {
            return pool;
        }

        public void setPool(Pool pool) {
            this.pool = pool;
        }

        public static class Pool {
            private int maxSize;
            private int minIdle;

            // Getters and setters
            public int getMaxSize() {
                return maxSize;
            }

            public void setMaxSize(int maxSize) {
                this.maxSize = maxSize;
            }

            public int getMinIdle() {
                return minIdle;
            }

            public void setMinIdle(int minIdle) {
                this.minIdle = minIdle;
            }
        }
    }
}
```

Custom Configuration Files

While application.properties is convenient, sometimes you may want to separate configurations into multiple files for better organization, especially in large applications.

Creating Custom Property Files

Let's create a custom property file named `mail-config.properties` in the `src/main/resources` directory:

```
mail.smtp.host=smtp.example.com
mail.smtp.port=587
mail.username=notification@example.com
mail.password=mailpassword
mail.from=no-reply@example.com
mail.enable-starttls=true
```

Loading Custom Property Files

To load this custom property file, you need to create a configuration class:

```java
@Configuration
@PropertySource("classpath:mail-config.properties")
public class MailConfiguration {

    @Bean
    public static PropertySourcesPlaceholderConfigurer
propertySourcesPlaceholderConfigurer() {
        return new PropertySourcesPlaceholderConfigurer();
    }

    @Bean
    @ConfigurationProperties(prefix = "mail")
    public MailProperties mailProperties() {
        return new MailProperties();
    }
}
```

And create a corresponding properties class:

```java
public class MailProperties {

    private String smtpHost;
    private int smtpPort;
    private String username;
    private String password;
    private String from;
    private boolean enableStarttls;

    // Getters and setters
    public String getSmtpHost() {
        return smtpHost;
    }

    public void setSmtpHost(String smtpHost) {
        this.smtpHost = smtpHost;
    }

    public int getSmtpPort() {
        return smtpPort;
    }
    public void setSmtpPort(int smtpPort) {
        this.smtpPort = smtpPort;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public String getFrom() {
        return from;
    }

    public void setFrom(String from) {
        this.from = from;
    }

    public boolean isEnableStarttls() {
        return enableStarttls;
    }

    public void setEnableStarttls(boolean enableStarttls) {
        this.enableStarttls = enableStarttls;
    }
}
```

Using Environment-Specific Configuration Files

Spring Boot allows you to have environment-specific configuration files. For example:

- `application-dev.properties`
- `application-test.properties`
- `application-prod.properties`

You can activate a specific profile using the `spring.profiles.active` property:

```
# In application.properties
spring.profiles.active=dev
```

Similarly, you can create environment-specific custom property files:

- `mail-config-dev.properties`
- `mail-config-prod.properties`

And load them conditionally:

```
@Configuration
@PropertySource("classpath:mail-config-
${spring.profiles.active:default}.properties")
public class MailConfiguration {
    // Configuration code
}
```

Advanced Configuration Techniques

Dynamic Property Loading

Sometimes, you might need to load properties from locations determined at runtime. Here's how you can achieve this:

```
@Configuration
public class DynamicPropertyConfiguration {

    @Autowired
    private Environment environment;

    @Bean
    public PropertySourcesPlaceholderConfigurer
propertySourcesPlaceholderConfigurer() {
        PropertySourcesPlaceholderConfigurer configurer = new
PropertySourcesPlaceholderConfigurer();

        String configPath = environment.getProperty("config.path",
"/default/path");
        Resource resource = new FileSystemResource(configPath + "/custom-
config.properties");

        configurer.setLocation(resource);
        configurer.setIgnoreResourceNotFound(true);

        return configurer;
    }
}
```

Encrypting Sensitive Properties

For sensitive information, you might want to encrypt your properties. Spring Cloud Config provides support for encrypted properties:

```
app.secret.key={cipher}AQA6+Ic4V9M3qrjJ4HbDgJ6BrNrAY7...
```

To use this feature, you need to include Spring Cloud Config dependencies and configure an encryption key.

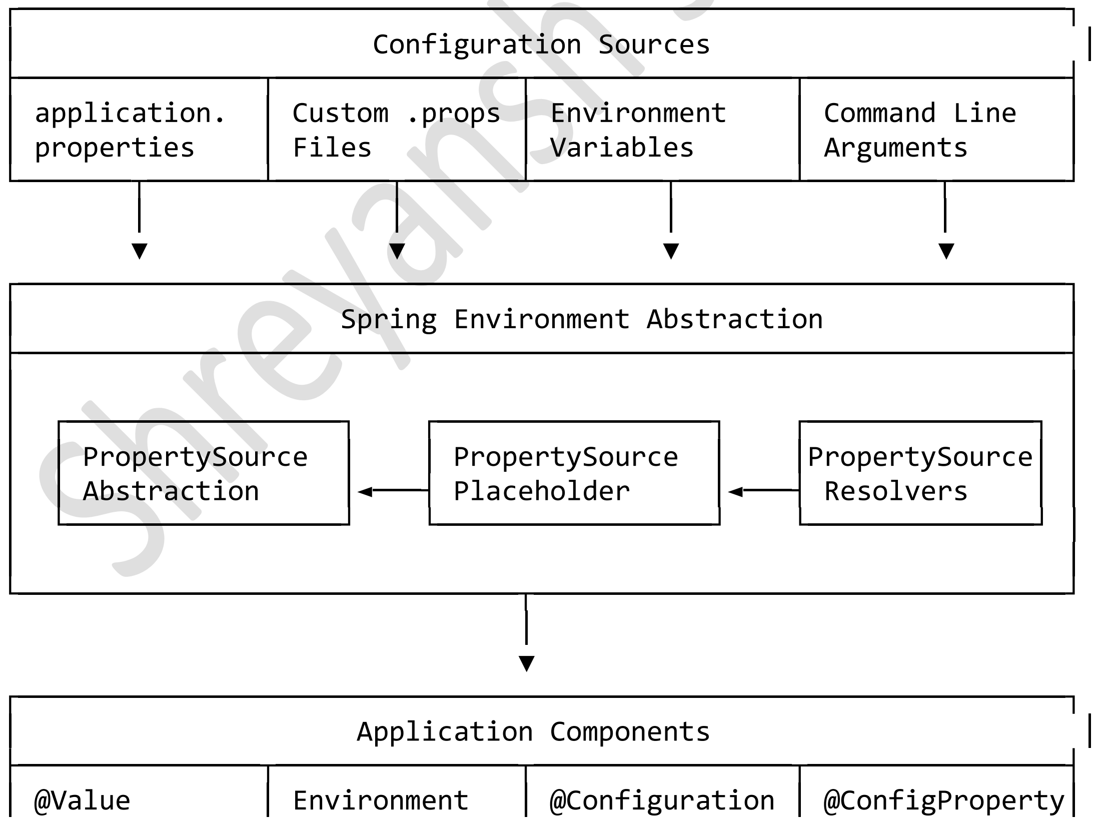
Refreshing Properties at Runtime

Spring Cloud provides the `@RefreshScope` annotation, which allows beans to be refreshed when configuration changes:

```
@RestController
@RefreshScope
public class ApiController {

    @Value("${app.feature.enabled}")
    private boolean featureEnabled;

    // Controller methods
}
```

When you trigger a refresh event (e.g., by calling a refresh endpoint), beans with `@RefreshScope` will be recreated with the new property values.

Architectural Diagram: Configuration Flow in Spring Applications



Annotation	Injection	Properties	Classes

Example: Complete Configuration System

Let's put everything together with a comprehensive example of a custom configuration system for a hypothetical e-commerce application.

Project Structure

```
src/
├── main/
│   ├── java/
│   │   └── com/
│   │       └── example/
│   │           └── ecommerce/
│   │               ├── EcommerceApplication.java
│   │               ├── config/
│   │               │   ├── AppConfig.java
│   │               │   ├── DatabaseConfig.java
│   │               │   ├── PaymentConfig.java
│   │               │   └── properties/
│   │               │       ├── AppProperties.java
│   │               │       ├── DatabaseProperties.java
│   │               │       └── PaymentProperties.java
│   │               ├── service/
│   │               │   ├── ProductService.java
│   │               │   ├── OrderService.java
│   │               │   └── PaymentService.java
│   │               └── controller/
│   │                   └── OrderController.java
│   └── resources/
│       ├── application.properties
│       ├── application-dev.properties
│       ├── application-prod.properties
│       ├── database-config.properties
│       └── payment-config.properties
│
```

Configuration Files

application.properties:

```
spring.application.name=ecommerce-service
server.port=8080
spring.profiles.active=dev

# Common application settings
app.name=E-Commerce Platform
app.version=1.0.0
app.support-email=support@example.com
app.feature.recommendations-enabled=true
app.feature.live-chat-enabled=false
app.session-timeout=30
```

application-dev.properties:

```
# Development-specific settings
logging.level.root=DEBUG
app.feature.live-chat-enabled=true
```

application-prod.properties:

```
# Production-specific settings
logging.level.root=INFO
server.tomcat.max-threads=200
app.session-timeout=15
```

database-config.properties:

```
db.driver=com.mysql.cj.jdbc.Driver
db.url=jdbc:mysql://localhost:3306/ecommerce
db.username=dbuser
db.password=dbpass
db.pool.initial-size=5
db.pool.max-size=20
db.pool.connection-timeout=30000
```

payment-config.properties:

```
payment.gateway.url=https://payment.example.com/api
payment.gateway.api-key=test-api-key
payment.gateway.timeout=5000
payment.methods.credit-card=true
payment.methods.paypal=true
payment.methods.crypto=false
```

Configuration Classes

AppConfig.java:

```java
@Configuration
@EnableConfigurationProperties(AppProperties.class)
public class AppConfig {

    @Autowired
    private AppProperties appProperties;

    @Bean
    public SupportService supportService() {
        return new SupportService(appProperties.getSupportEmail());
    }
    @Bean
    public SessionManager sessionManager() {
        return new SessionManager(appProperties.getSessionTimeout());
    }
}
```

DatabaseConfig.java:

```
@Configuration
@PropertySource("classpath:database-config.properties")
@EnableConfigurationProperties(DatabaseProperties.class)
public class DatabaseConfig {

    @Autowired
    private DatabaseProperties dbProperties;

    @Bean
    public DataSource dataSource() {
        HikariConfig config = new HikariConfig();
        config.setDriverClassName(dbProperties.getDriver());
        config.setJdbcUrl(dbProperties.getUrl());
        config.setUsername(dbProperties.getUsername());
        config.setPassword(dbProperties.getPassword());
        config.setMinimumIdle(dbProperties.getPool().getInitialSize());
        config.setMaximumPoolSize(dbProperties.getPool().getMaxSize());

config.setConnectionTimeout(dbProperties.getPool().getConnectionTimeout());

        return new HikariDataSource(config);
    }
}
```

PaymentConfig.java:

```
@Configuration
@PropertySource("classpath:payment-config.properties")
@EnableConfigurationProperties(PaymentProperties.class)
public class PaymentConfig {

    @Autowired
    private PaymentProperties paymentProperties;

    @Bean
    public PaymentGatewayClient paymentGatewayClient() {
        return new PaymentGatewayClient(
            paymentProperties.getGateway().getUrl(),
            paymentProperties.getGateway().getApiKey(),
            paymentProperties.getGateway().getTimeout()
        );
    }

    @Bean
    public PaymentMethodsService paymentMethodsService() {
        return new PaymentMethodsService(paymentProperties.getMethods());
    }
}
```

Property Classes

AppProperties.java:

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

    private String name;
    private String version;
    private String supportEmail;
    private int sessionTimeout;
    private FeatureFlags feature = new FeatureFlags();

    // Getters and setters

    public static class FeatureFlags {
        private boolean recommendationsEnabled;
        private boolean liveChatEnabled;

        // Getters and setters
    }
}
```

DatabaseProperties.java:

```
@ConfigurationProperties(prefix = "db")
public class DatabaseProperties {

    private String driver;
    private String url;
    private String username;
    private String password;
    private Pool pool = new Pool();

    // Getters and setters

    public static class Pool {
        private int initialSize;
        private int maxSize;
        private int connectionTimeout;
```

```java
        // Getters and setters
    }
}
```

PaymentProperties.java:

```java
@ConfigurationProperties(prefix = "payment")
public class PaymentProperties {

    private Gateway gateway = new Gateway();
    private Methods methods = new Methods();

    // Getters and setters

    public static class Gateway {
        private String url;
        private String apiKey;
        private int timeout;

        // Getters and setters
    }

    public static class Methods {
        private boolean creditCard;
        private boolean paypal;
        private boolean crypto;

        // Getters and setters
    }
}
```

Using the Configuration in Services

PaymentService.java:

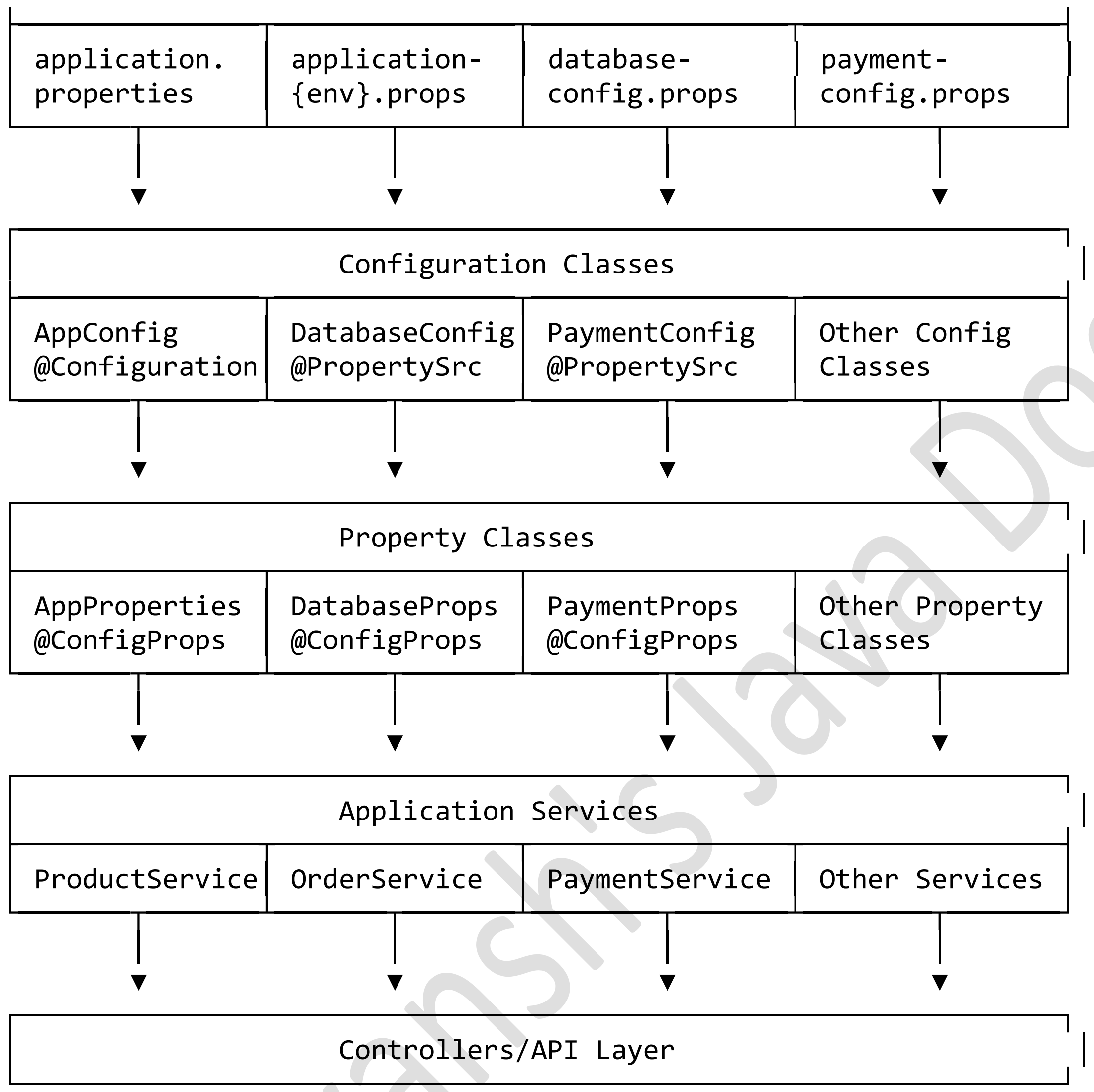
```java
@Service
public class PaymentService {

    private final PaymentGatewayClient paymentGatewayClient;
    private final PaymentMethodsService paymentMethodsService;

    public PaymentService(
            PaymentGatewayClient paymentGatewayClient,
            PaymentMethodsService paymentMethodsService) {
        this.paymentGatewayClient = paymentGatewayClient;
        this.paymentMethodsService = paymentMethodsService;
    }

    public List<String> getAvailablePaymentMethods() {
        return paymentMethodsService.getEnabledMethods();
    }

    public PaymentResult processPayment(Order order, PaymentDetails details)
{
        // Use the payment gateway client to process payment
        return paymentGatewayClient.processPayment(order.getTotalAmount(),
details);
    }
}
```

ProductService.java:

```java
@Service
public class ProductService {

    private final DataSource dataSource;
    private final AppProperties appProperties;

    public ProductService(DataSource dataSource, AppProperties appProperties)
{
        this.dataSource = dataSource;
        this.appProperties = appProperties;
    }

    public List<Product> getRecommendedProducts(long userId) {
        if (appProperties.getFeature().isRecommendationsEnabled()) {
            // Fetch personalized recommendations from database
            return fetchRecommendationsFromDb(userId);
        } else {
            // Return generic popular products
            return fetchPopularProducts();
        }
    }

    private List<Product> fetchRecommendationsFromDb(long userId) {
        // Implementation using dataSource
        return new ArrayList<>();
    }

    private List<Product> fetchPopularProducts() {
        // Implementation using dataSource
        return new ArrayList<>();
    }
}
```

Architectural Diagram: Custom Configuration System

```
┌──────────────────────────────────────────────────────────────┐
│                    Configuration Files                         │
```



Best Practices for Custom Configuration

1. Organize Properties Logically

Group related properties together with meaningful prefixes. For example, all database-related properties should start with `db.` or `database.`.

2. Validate Configuration Properties

Use validation annotations to ensure that your configuration properties meet your requirements:

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

    @NotBlank
```

```java
    private String name;

    @Email
    private String supportEmail;

    @Min(5)
    @Max(60)
    private int sessionTimeout;

    // Getters and setters
}
```

3. Document Your Properties

Use the `@ConfigurationProperties` metadata to document your properties:

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

    /**
     * The name of the application.
     */
    private String name;

    /**
     * Email address for customer support inquiries.
     */
    private String supportEmail;

    /**
     * Session timeout in minutes. Must be between 5 and 60.
     */
    private int sessionTimeout;

    // Getters and setters
}
```

4. Use Relaxed Binding

Spring Boot's relaxed binding allows properties to be specified in different formats:

- Kebab case: `app.support-email`
- Camel case: `app.supportEmail`
- Underscore notation: `app.support_email`
- Environment variables: `APP_SUPPORT_EMAIL`

5. Provide Default Values

Always provide sensible default values for your properties:

```java
@Value("${app.feature.enabled:false}")
private boolean featureEnabled;
```

Or in configuration classes:

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

    private String name = "Default App";
    private int timeout = 30;

    // Getters and setters
}
```

6. Use Profiles for Environment-Specific Configuration

Leverage Spring profiles to manage environment-specific configurations:

```
# application-dev.properties
app.feature.debug=true

# application-prod.properties
app.feature.debug=false
```

7. Secure Sensitive Properties

For sensitive information like passwords and API keys:

- Use environment variables
- Use encrypted properties
- Use a secure vault like HashiCorp Vault or AWS Secrets Manager

8. Centralize Configuration for Microservices

For microservice architectures, consider using Spring Cloud Config Server to centralize configuration management.

Conclusion

Custom configuration in Spring applications provides flexibility and organization for your application settings. By leveraging Spring's powerful configuration mechanisms, you can create a maintainable, environment-aware configuration system that adapts to your application's needs.

This document has covered:

- Basic property configuration in application.properties
- Creating and using custom configuration files
- Binding properties to Java objects with @ConfigurationProperties

- Environment-specific configuration
- Advanced techniques like dynamic property loading and property encryption
- A comprehensive example of a complete configuration system
- Best practices for managing configuration in Spring applications

By following these patterns and practices, you can build a robust configuration system that grows with your application and supports your development workflow across different environments.