Berner
Fachhochschule

Spring Configuration

Stephan Fischli

Configuration Classes

Definition

- As an alternative to annotations, beans can also be created using configuration classes
- Configuration classes are annotated with `@Configuration` and declare beans through factory methods annotated with `@Bean`
- The `name` attribute can be used to specify the name of the bean, otherwise it is set to the name of the method
- The `@Scope` annotation can be used to specify the scope of a bean

```
@Configuration
public class ApplicationConfig {

    @Bean(name = "recommender")
    @Scope("singleton")
    public MovieRecommender movieRecommender() {
        ...
        return new MovieRecommender(...);
        }
}
```

Bean Dependencies

- An `@Bean` method can have parameters that represent the dependencies of the bean to be declared
- The resolution mechanism is based firstly on the parameter type and secondly on the parameter name
- In addition, qualifiers can be used to specify the beans to be injected

```java
@Configuration
public class ApplicationConfig {

    @Bean
    public MovieRecommender movieRecommender(MovieCatalog catalog) {
        ...
        return new MovieRecommender(catalog, ...);
    }
    @Bean
    public MovieCatalog movieCatalog() {
        return new MovieCatalog();
        }
}
```

Profiles

Environment

- The Spring environment allows the configuration of a Spring application for different runtime environments and uses two key aspects:
  - A *profile* is a group of bean definitions to be registered only if a given profile is active
  - *Properties* are used to configure applications and may originate from a variety of sources
- The **`Environment`** interface allows for configuring property sources and resolving properties

The @Profile Annotation

- The `@Profile` annotation indicates that a component is registered only when one or more specific profiles are active
- The profile string may consist of an expression with logical operators

```
@Component
@Profile("dev")
public class StandaloneDataSource {
    ...
}
```

```
@Component
@Profile("prod")
public class JndiDataSourceConfig {
    ...
}
```

The @Profile Annotation (cont.)

- The `@Profile` can also be declared at the method level to include only particular beans of a configuration class

```java
@Configuration
public class ApplicationConfig {

    @Profile("dev")
    @Bean("dataSource")
    public DataSource standaloneDataSource() { ... }

    @Profile("prod")
    @Bean("dataSource")
    public DataSource jndiDataSource() { ... }
}
```

Activating a Profile

- Profiles are activated with the `spring.profiles.active` property which can be specified as a system environment variable

```
> java -jar target/bookstore-1.0.jar --spring.profiles.active=prod
```

- In integration tests, active profiles can be declared with the `@ActiveProfiles` annotation of the `spring-test` module

```
@SpringBootTest
@ActiveProfiles("dev")
public class BookstoreTests { ... }
```

Default Profile

- The default profile represents the profile that is enabled by default
- If any profile is enabled, the default profile does not apply

```
@Configuration
@Profile("default")
public class DefaultDataConfig {
    @Bean
    public DataSource dataSource() { ... }
}
```

Properties

Introduction

- Spring Boot allows the definition of properties in different sources, so that the application code is independent of the runtime environment
- Property values can be injected, accessed through Spring's environment, or be bound to structured objects

Property Usage

- When a component uses a property, the value can be injected using the `@Value` annotation
- The name of the property can be specified with a SpEL expression
- An optional default value can be provided after a colon

```
@Component
public class PaymentService {

    @Value("${payment-limit:1000}")
    private long paymentLimit;
    ...
}
```

Property Sources

- Spring Boot defines a particular order of precedence of property sources:
    1. Command line arguments
    2. Servlet init parameters
    3. JNDI attributes
    4. Java System properties
    5. OS environment variables
    6. Random values (`RandomValuePropertySource`)
    7. Profile-specific application properties outside or inside the JAR
    8. Application properties outside or inside the JAR
    9. Properties from configuration classes (`@PropertySource`)

Command Line Properties

- The `SpringApplication` class converts command line arguments to properties and adds them to the application's environment
- Command line arguments always take precedence over other property source

```
> java -jar target/bookstore-1.0.jar --payment-limit=500
```

OS Environment Variables

- Properties can also be defined as environment variables of the operating system in which the application is running

```
> set PAYMENT_LIMIT=500
> java -jar target/bookstore-1.0.jar
```

Property Files

- The `SpringApplication` class loads properties from `application.properties` files in the current directory, the `config` subdirectory, the `config` package, and the classpath root
- Profile-specific properties from files `application-{profile}.properties` extend or override the general application properties

```
mail.enabled=true
mail.server-address=192.168.1.1
mail.security.username=admin
mail.security.password=********
mail.security.roles[0]=USER
mail.security.roles[1]=ADMIN
```

YAML Files

- YAML is a superset of JSON and provides a convenient format for specifying hierarchical configuration data
- The `SpringApplication` class supports YAML as an alternative format and converts YAML files into properties

```yaml
mail:
    enabled: true
    server-address: 192.168.1.1
    security:
        username: admin
        password: *****
        roles:
            - USER
            - ADMIN
```

Relaxed Name Binding

- The `Environment` property names do not have to match exactly the bean property names
- The following property names are supported
    - Kebab case (`mail.server-address`)
    - Camel case (`mail.serverAddress`)
    - Underscore notation (`mail.server_address`)
    - Upper case format (`MAIL_SERVERADDRESS`)

Type-safe Configuration Properties

- Spring Boot allows configuration properties to be bound to strongly typed classes
- A class annotated with `@ConfigurationProperties` defines a holder for properties with a specific prefix
- In addition, the `@Validated` annotation causes the class properties to be validated

```
@ConfigurationProperties(prefix = "mail")
@Validated
public record MailProperties(boolean enabled,
    @NotNull InetAddress serverAddress,
    @Valid Security security) {
}

record Security(@NotEmpty String username, String password,
List<String> roles) {}
```

Configuration Properties Injection

- If a configuration properties class is to be injected into a component, it must be defined as a bean
- This can be achieved by using the `@ConfigurationPropertiesScan` annotation on a configuration class

```java
@SpringBootApplication
@ConfigurationPropertiesScan
public class BookstoreApplication { ... }

@Service
public class MailService {
    private MailProperties properties;

    public MailService(MailProperties properties) {
        this.properties = properties;
    }
    ...
}
```