Configuration Properties

jim stafford

Table of Contents

Chapter 1. Introduction

In the previous chapter we mapped properties from different sources and then mapped them directly into individual component Java class attributes. That showed a lot of power but had at least one flaw — each component would define its own injection of a property. If we changed the structure of a property, we would have many places to update and some of that might not be within our code base.

In this chapter we are going to continue to leverage the same property source(s) as before but remove the individual configuration properties entirely from the component classes and encapsulate them within a configuration class that gets instantiated, populated, and injected into the component at runtime.

We will also explore adding validation of properties and leveraging tooling to automatically generate boilerplate JavaBean constructs.

1.1. Goals

The student will learn to:

- map a Java `@ConfigurationProperties` class to properties
- define validation rules for property values
- leverage tooling to generate boilerplate code for JavaBean classes
- solve more complex property mapping scenarios
- solve injection mapping or ambiguity

1.2. Objectives

At the conclusion of this lecture and related exercises, the student will be able to:

1. map a Java `@ConfigurationProperties` class to a group of properties
   - generate property metadata — used by IDEs for property editors
2. create read-only `@ConfigurationProperties` class using `@ConstructorBinding`
3. define Jakarta EE Java validation rule for property and have validated at runtime
4. generate boilerplate JavaBean methods using Lombok library
5. use relaxed binding to map between JavaBean and property syntax
6. map nested properties to a `@ConfigurationProperties` class
7. map array properties to a `@ConfigurationProperties` class
8. reuse `@ConfigurationProperties` class to map multiple property trees
9. use `@Qualifier` annotation and other techniques to map or disambiguate an injection

Chapter 2. Mapping properties to @ConfigurationProperties class

Starting off simple, we define a property (`app.config.car.name`) in `application.properties` to hold the name of a car.

```
# application.properties
app.config.car.name=Suburban
```

2.1. Mapped Java Class

At this point we now want to create a Java class to be instantiated and be assigned the value(s) from the various property sources — `application.properties` in this case, but as we have seen from earlier lectures properties can come from many places. The class follows standard JavaBean characteristics

- default constructor to instantiate the class in a default state
- "setter"/"getter" methods to set and get the state of the instance

A "toString()" method was also added to self-describe the state of the instance.

```java
import org.springframework.boot.context.properties.ConfigurationProperties;

@ConfigurationProperties("app.config.car") ③
public class CarProperties { ①
    private String name;

    //default ctor ②

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name; ②
    }

    @Override
    public String toString() {
        return "CarProperties{name='" + name + "\'}";
    }
}
```

① class is a standard Java bean with one property

② class designed for us to use its default constructor and a setter() to assign value(s)

③ class annotated with `@ConfigurationProperties` to identify that is mapped to properties and the property prefix that pertains to this class

2.2. Injection Point

We can have Spring instantiate the bean, set the state, and inject that into a component at runtime and have the state of the bean accessible to the component.

```java
...
@Component
public class AppCommand implements CommandLineRunner {
    @Autowired
    private CarProperties carProperties; ①

    public void run(String... args) throws Exception {
        System.out.println("carProperties=" + carProperties); ②
...
```

① Our `@ConfigurationProperties` instance is being injected into a `@Component` class using FIELD injection

② Simple print statement of bean's `toString()` result

2.3. Initial Error

However, if we build and run our application at this point, our injection will fail because Spring was not able to locate what it needed to complete the injection.

```
***************************
APPLICATION FAILED TO START
***************************


Description:

Field carProperties in info.ejava.examples.app.config.configproperties.AppCommand
required a bean
  of type 'info.ejava.examples.app.config.configproperties.properties.CarProperties'
that could
  not be found.

The injection point has the following annotations:
    - @org.springframework.beans.factory.annotation.Autowired(required=true)

Action:

Consider defining a bean of type
  'info.ejava.examples.app.config.configproperties.properties.CarProperties'
  in your configuration. ①
```

① Error message indicates that Spring is not seeing our `@ConfigurationProperties` class

2.4. Registering the @ConfigurationProperties class

We currently have a similar problem that we had when we implemented our first `@Configuration` and `@Component` classes — the bean is not being scanned. Even though we have our `@ConfigurationProperties` class is in the same basic classpath as the `@Configuration` and `@Component` classes — we need a little more to have it processed by Spring. There are several ways to do that:

```
src
`-- main
    |-- java
    |   `-- info
    |       `-- ejava
    |           `-- examples
    |               `-- app
    |                   `-- config
    |                       `-- configproperties
    |                           |-- AppCommand.java
    |                           |-- ConfigurationPropertiesApp.java
    |                           `-- properties
    |                               `-- CarProperties.java
    `-- resources
        `-- application.properties
```

2.4.1. way 1 - Register Class as a @Component

Our package is being scanned by Spring for components, so if we add a `@Component` annotation the `@ConfigurationProperties` class will be automatically picked up.

```
package info.ejava.examples.app.config.configproperties.properties;
...
@Component
@ConfigurationProperties("app.config.car") ①
public class CarProperties {
```

① causes Spring to process the bean and annotation as part of component classpath scanning

- benefits: simple
- drawbacks: harder to override when configuration class and component class are in the same Java class package tree

2.4.2. way 2 - Explicitly Register Class

Explicitly register the class using `@EnableConfigurationProperties` annotation on a `@Configuration` class (such as the `@SpringBootApplication` class)

```
import info.ejava.examples.app.config.configproperties.properties.CarProperties;
import org.springframework.boot.context.properties.ConfigurationPropertiesScan;
...
@SpringBootApplication
@EnableConfigurationProperties(CarProperties.class) ①
public class ConfigurationPropertiesApp {
```

① targets a specific `@ConfigurationProperties` class to process

- benefits: `@Configuration` class has explicit control over which configuration properties classes to activate

- drawbacks: application could be coupled with the details if where configurations come from

2.4.3. way 3 - Enable Package Scanning

Enable package scanning for `@ConfigurationProperties` classes with the `@ConfigurationPropertiesScan` annotation

```
@SpringBootApplication
@ConfigurationPropertiesScan ①
public class ConfigurationPropertiesApp {
```

① allows a generalized scan to be defined that is separate for configurations

- benefits: easy to add more configuration classes without changing application

- drawbacks: generalized scan may accidentally pick up an unwanted configuration

2.4.4. way 4 - Use @Bean factory

Create a `@Bean` factory method in a `@Configuration` class for the type .

```
@SpringBootApplication
public class ConfigurationPropertiesApp {
...
    @Bean
    @ConfigurationProperties("app.config.car") ①
    public CarProperties carProperties() {
        return new CarProperties();
    }
```

① gives more control over the runtime mapping of the bean to the `@Configuration` class

- benefits: decouples the `@ConfigurationProperties` class from the specific property prefix used to populate it. This allows for reuse of the same `@ConfigurationProperties` class for multiple prefixes

- drawbacks: implementation spread out between the `@ConfigurationProperties` and `@Configuration` classes. It also prohibits the use of read-only instances since the returned object

is not yet populated

For our solution for this example, I am going to use @ConfigurationPropertiesScan ("way3") and drop multiple @ConfigurationProperties classes into the same classpath and have them automatically scanned for.

2.5. Result

Having things properly in place, we get the instantiated and initialized CarProperties @ConfigurationProperties class injected into out component(s). Our example AppCommand component simply prints the toString() result of the instance and we see the property we set in the applications.property file.

Property Definition

```
# application.properties
app.config.car.name=Suburban
```

Injected @Component Processing the Bean

```
...
@Component
public class AppCommand implements CommandLineRunner {
    @Autowired
    private CarProperties carProperties;

    public void run(String... args) throws Exception {
        System.out.println("carProperties" + carProperties);
...
```

Produced Output

```
$ java -jar target/appconfig-configproperties-example-*-SNAPSHOT-bootexec.jar
...
carProperties=CarProperties{name='Suburban'}
```

Chapter 3. Metadata

IDEs have support for linking Java properties to their `@ConfigurationProperty` class information.
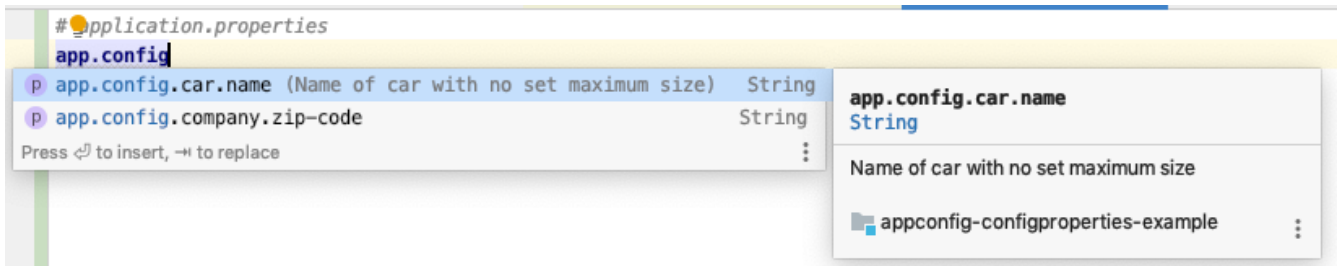

Figure 1. IDE Configuration Property Support

This allows the property editor to know:

- there is a property `app.config.carname`
- any provided Javadoc

> 💡 Spring Configuration Metadata and IDE support is very helpful when faced with configuring dozens of components with hundreds of properties (or more!)

3.1. Spring Configuration Metadata

IDEs rely on a JSON-formatted metadata file located in `META-INF/spring-configuration-metadata.json` to provide that information.

META-INF/spring-configuration-metadata.json Snippet

```
...
"properties": [
    {
        "name": "app.config.car.name",
        "type": "java.lang.String",
        "description": "Name of car with no set maximum size",
        "sourceType":
"info.ejava.examples.app.config.configproperties.properties.CarProperties"
    }
...
```

We can author it manually. However, there are ways to automate this.

3.2. Spring Configuration Processor

To have Maven automatically generate the JSON metadata file, add the following dependency to the project to have additional artifacts generated during Java compilation. The Java compiler will inspect and recognize a type of class inside the dependency and call it to perform additional processing. Make it `optional=true` since it is only needed during compilation and not at runtime.

```xml
<!-- pom.xml dependencies -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-configuration-processor</artifactId> ①
    <optional>true</optional> ②
</dependency>
```

① dependency will generate additional artifacts during compilation

② dependency not required at runtime and can be eliminated from dependents

> 💡 Dependencies labelled `optional=true` or `scope=provided` are not included in the Spring Boot executable JAR or transitive dependencies in downstream deployments without further configuration by downstream dependents.

3.3. Javadoc Supported

As noted earlier, the metadata also supports documentation extracted from Javadoc comments. To demonstrate this, I will add some simple Javadoc to our example property.

```java
@ConfigurationProperties("app.config.car")
public class CarProperties {
    /**
     * Name of car with no set maximum size ①
     */
    private String name;
```

① Javadoc information is extracted from the class and placed in the property metadata

3.4. Rebuild Module

Rebuilding the module with Maven and reloading the module within the IDE should give the IDE additional information it needs to help fill out the properties file.

Metadata File Created During Compilation

```
$ mvn clean compile
```

Produced Metadata File in `target/classes` Tree

```
target/classes/META-INF/
`-- spring-configuration-metadata.json
```

Produced Metadata File Contents

```json
{
  "groups": [
```

```json
    {
      "name": "app.config.car",
      "type":
"info.ejava.examples.app.config.configproperties.properties.CarProperties",
      "sourceType":
"info.ejava.examples.app.config.configproperties.properties.CarProperties"
    }
  ],
  "properties": [
    {
      "name": "app.config.car.name",
      "type": "java.lang.String",
      "description": "Name of car with no set maximum size",
      "sourceType":
"info.ejava.examples.app.config.configproperties.properties.CarProperties"
    }
  ],
  "hints": []
}
```

3.5. IDE Property Help

If your IDE supports Spring Boot and property metadata, the property editor will offer help filling out properties.
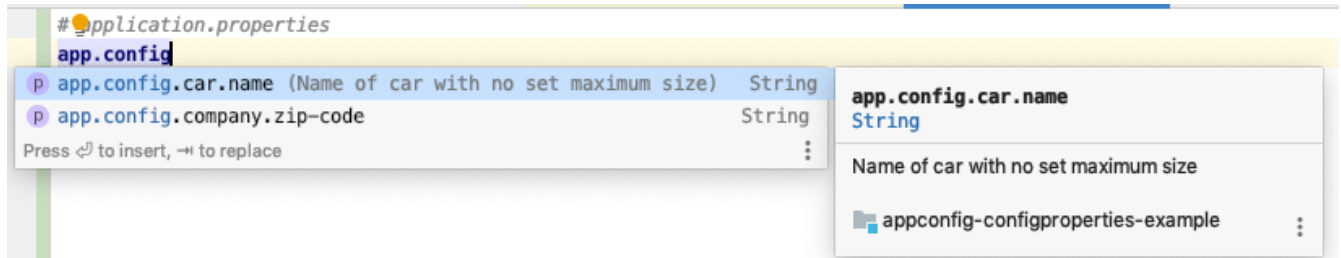
> IntelliJ free Community Edition does not support this feature. The following link provides a comparison with the for-cost Ultimate Edition.

Chapter 4. Constructor Binding

The previous example was a good start. However, I want to create a slight improvement at this point with a similar example and make the JavaBean read-only. This better depicts the contract we have with properties. They are read-only.

To accomplish a read-only JavaBean, we should remove the setter(s), create a custom constructor that will initialize the attributes at instantiation time, and ideally declare the attributes as final to enforce that they get initialized during construction and never changed.

The only requirement Spring places on us is to add a `@ConstructorBinding` annotation to the class or constructor method when using this approach.

Constructor Binding Example

```java
...
import org.springframework.boot.context.properties.ConstructorBinding;

@ConfigurationProperties("app.config.boat")
public class BoatProperties {
    private final String name; ③

    @ConstructorBinding ②
    public BoatProperties(String name) {
        this.name = name;
    }
    //no setter method(s) ①
    public String getName() {
        return name;
    }
    @Override
    public String toString() {
        return "BoatProperties{name='" + name + "\'}";
    }
}
```

① remove setter methods to better advertise the read-only contract of the bean

② add custom constructor and annotate the class or constructor with `@ConstructorBinding`

③ make attributes final to better enforce the read-only nature of the bean

> `@ConstructorBinding` annotation required on the constructor method when more than one constructor is supplied.

4.1. Property Names Bound to Constructor Parameter Names

When using constructor binding, we no longer have the name of the setter method(s) to help map

the properties. The parameter name(s) of the constructor are used instead to resolve the property values.

In the following example, the property `app.config.boat.name` matches the constructor parameter `name`. The result is that we get the output we expect.

```
# application.properties
app.config.boat.name=Maxum
```

Result of Parameter Name Matching Property Name

```
$ java -jar target/appconfig-configproperties-example-*-SNAPSHOT-bootexec.jar
...
boatProperties=BoatProperties{name='Maxum'}
```

4.2. Constructor Parameter Name Mismatch

If we change the constructor parameter name to not match the property name, we will get a null for the property.

```
@ConfigurationProperties("app.config.boat")
public class BoatProperties {
    private final String name;

    @ConstructorBinding
    public BoatProperties(String nameX) { ①
        this.name = nameX;
    }
```

① constructor argument name has been changed to not match the property name from `application.properties`

Result of Parameter Name not Matching Property Name

```
$ java -jar target/appconfig-configproperties-example-*-SNAPSHOT-bootexec.jar
...
boatProperties=BoatProperties{name='null'}
```

> ℹ️ We will discuss relaxed binding soon and see that some syntactical differences between the property name and JavaBean property name are accounted for during `@ConfigurationProperties` binding. However, this was a clear case of a name mis-match that will not be mapped.

Chapter 5. Validation

The error in the last example would have occurred whether we used constructor or setter-based binding. We would have had a possibly vague problem if the property was needed by the application. We can help detect invalid property values for both the setter and constructor approaches by leveraging validation.

Java validation is a JavaEE/ Jakarta EE standard API for expressing validation for JavaBeans. It allows us to express constraints on JavaBeans to help further modularize objects within our application.

To add validation to our application, we start by adding the Spring Boot validation starter (`spring-boot-starter-validation`) to our pom.xml.

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-validation</artifactId>
</dependency>
```

This will bring in three (3) dependencies

- jakarta.validation-api - this is the validation API and is required to compile the module

- hibernate-validator - this is a validation implementation

- tomcat-embed-el - this is required when expressing validations using regular expressions with `@Pattern` annotation

5.1. Validation Annotations

We trigger Spring to validate our JavaBean when instantiated by the container by adding the Spring `@Validated` annotation to the class. We further define the Java attribute with the Jakarta EE `@NotNull` constraint to report an error if the property is ever null.

@ConfigurationProperties JavaBean with Validation

```java
...
import org.springframework.validation.annotation.Validated;
import javax.validation.constraints.NotNull;

@ConfigurationProperties("app.config.boat")
@Validated ①
public class BoatProperties {
    @NotNull ②
    private final String name;

    @ConstructorBinding
    public BoatProperties(String nameX) {
        this.name = nameX;
```

```
        }
...
```

① The Spring `@Validated` annotation tells Spring to validate instances of this class

② The Jakarta EE `@NotNull` annotation tells the validator this field is not allowed to be null

> ℹ You can locate other validation constraints in the Validation API and also extend the API to provide more customized validations using the Validation Spec, Hibernate Validator Documentation, or various web searches.

5.2. Validation Error

The error produced is caught by Spring Boot and turned into a helpful description of the problem clearly stating there is a problem with one of the properties specified (when actually it was a problem with the way the JavaBean class was implemented)

```
$ java -jar target/appconfig-configproperties-example-*-SNAPSHOT-bootexec.jar
***************************
APPLICATION FAILED TO START
***************************
Description:

Binding to target org.springframework.boot.context.properties.bind.BindException:
    Failed to bind properties under 'app.config.boat' to
    info.ejava.examples.app.config.configproperties.properties.BoatProperties failed:

     Property: app.config.boat.name
     Value: null
     Reason: must not be null

Action:

Update your application's configuration
```

> ℹ Notice how the error message output by Spring Boot automatically knew what a validation error was and that the invalid property mapped to a specific property name. That is an example of Spring Boot's FailureAnalyzer framework in action — which aims to make meaningful messages out of what would otherwise be a clunky stack trace.

Chapter 6. Boilerplate JavaBean Methods

Before our implementations gets more complicated, we need to address a simplification we can make to our JavaBean source code which will make all future JavaBean implementations incredibly easy.

Notice all the boilerplate constructor, getter/setter, toString(), etc. methods within our earlier JavaBean classes? These methods are primarily based off the attributes of the class. They are commonly implemented by IDEs during development but then become part of the overall code base that has to be maintained over the lifetime of the class. This will only get worse as we add additional attributes to the class when our code gets more complex.

```java
...
@ConfigurationProperties("app.config.boat")
@Validated
public class BoatProperties {
    @NotNull
    private final String name;

    @ConstructorBinding
    public BoatProperties(String name) { //boilerplate ①
        this.name = name;
    }

    public String getName() {  //boilerplate ①
        return name;
    }

    @Override
    public String toString() { //boilerplate ①
        return "BoatProperties{name='" + name + "\'}";
    }
}
```

① Many boilerplate methods in source code — likely generated by IDE

6.1. Generating Boilerplate Methods with Lombok

These boilerplate methods can be automatically provided for us at compilation using the Lombok library. Lombok is not unique to Spring Boot but has been adopted into Spring Boot's overall opinionated approach to developing software and has been integrated into the popular Java IDEs.

I will introduce various Lombok features during later portions of the course and start with a simple case here where all defaults for a JavaBean are desired. The simple Lombok @Data annotation intelligently inspects the JavaBean class with just an attribute and supplies boilerplate constructs commonly supplied by the IDE:

- constructor to initialize attributes

- getter
- toString()
- hashCode() and equals()

A setter was not defined by Lombok because the `name` attribute is declared final.

Java Bean using Lombok

```
...
import lombok.Data;

@ConfigurationProperties("app.config.company")
@ConstructorBinding
@Data ①
@Validated
public class CompanyProperties {
    @NotNull
    private final String name;
    //constructor ①
    //getter ①
    //toString ①
    //hashCode and equals ①
}
```

① Lombok `@Data` annotation generated constructor, getter(/setter), toString, hashCode, and equals

6.2. Visible Generated Constructs

The additional methods can be identified in a class structure view of an IDE or using Java disassembler (`javap`) command

Example IDE Class Structure View

> ℹ You may need to locate a compiler option within your IDE properties to make the code generation within your IDE.

```
$ javap -cp target/classes
info.ejava.examples.app.config.configproperties.properties.CompanyProperties
Compiled from "CompanyProperties.java"
public class
info.ejava.examples.app.config.configproperties.properties.CompanyProperties {
  public
info.ejava.examples.app.config.configproperties.properties.CompanyProperties(java.lang
.String);
  public java.lang.String getName();
  public boolean equals(java.lang.Object);
  protected boolean canEqual(java.lang.Object);
  public int hashCode();
  public java.lang.String toString();
}
```

6.3. Lombok Build Dependency

The Lombok annotations are defined with `RetentionPolicy.SOURCE`. That means they are discarded by the compiler and not available at runtime.

Lombok Annotations are only used at Compile-time

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.SOURCE)
public @interface Data {
```

That permits us to declare the dependency as `scope=provided` to eliminate it from the application's executable JAR and transitive dependencies and have no extra bloat in the module as well.

Maven Dependency

```xml
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <scope>provided</scope>
</dependency>
```

6.4. Example Output

Running our example using the same, simple `toString()` print statement and property definitions produces near identical results from the caller's perspective. The only difference here is the specific text used in the returned string.

```
...
@Autowired
```

```java
    private BoatProperties boatProperties;
    @Autowired
    private CompanyProperties companyProperties;

    public void run(String... args) throws Exception {
        System.out.println("boatProperties=" + boatProperties); ①
        System.out.println("====");
        System.out.println("companyProperties=" + companyProperties); ②
...
```

① `BoatProperties` JavaBean methods were provided by hand

② `CompanyProperties` JavaBean methods were provided by Lombok

```properties
# application.properties
app.config.boat.name=Maxum
app.config.company.name=Acme
```

```
$ java -jar target/appconfig-configproperties-example-*-SNAPSHOT-bootexec.jar
...
boatProperties=BoatProperties{name='Maxum'}
====
companyProperties=CompanyProperties(name=Acme)
```

There is a Spring @ConstructorBinding issue that prevents property metadata from being automatically generated. This is due to a Lombok issue where usable argument names are not provided in the generated constructor. The only workaround at this time if you want metadata generated for `@ConstructorBinding` with Lombok is to provide a custom constructor supplying the valid names. The IDE is very good at generating these for you until that issue is corrected.

```java
@ConfigurationProperties("app.config.company")
@ConstructorBinding
@Data
@Validated
public class CompanyProperties {
    @NotNull
    private final String name;

    //https://github.com/spring-projects/spring-boot/issues/18730
    //https://github.com/rzwitserloot/lombok/issues/2275
    public CompanyProperties(String name) {
        this.name = name;
    }
}
```

Lombok ConstructorBinding Issue Listed as Closed

Since providing the warning above, the version of Lombok has advanced in class (`1.18.20`), issue closed, and may have been resolved. Confirmation needed.

With the exception of the property metadata issue just mentioned, adding Lombok to our development approach for JavaBeans is almost a 100% win situation. 80-90% of the JavaBean class is written for us and we can override the defaults at any time with further annotations or custom methods. The fact that Lombok will not replace methods we have manually provided for the class always gives us an escape route in the event something needs to be customized.

Chapter 7. Relaxed Binding

One of the key differences between Spring's `@Value` injection and `@ConfigurationProperties` is the support for relaxed binding by the later. With relaxed binding, property definitions do not have to be an exact match. JavaBean properties are commonly defined with camelCase. Property definitions can come in a number of different case formats. Here is a few.

- camelCase

- UpperCamelCase

- kebab-case

- snake_case

- UPPERCASE

7.1. Relaxed Binding Example JavaBean

In this example, I am going to add a class to express many different properties of a business. Each of the attributes is expressed using camelCase to be consistent with common Java coding conventions and further validated using Jakarta EE Validation.

JavaBean Attributes using camelCase

```java
@ConfigurationProperties("app.config.business")
@ConstructorBinding
@Data
@Validated
public class BusinessProperties {
    @NotNull
    private final String name;
    @NotNull
    private final String streetAddress;
    @NotNull
    private final String city;
    @NotNull
    private final String state;
    @NotNull
    private final String zipCode;
    private final String notes;
}
```

7.2. Relaxed Binding Example Properties

The properties supplied provide an example of the relaxed binding Spring implements between property and JavaBean definitions.

Example Properties to Demonstrate Relaxed Binding

```
# application.properties
app.config.business.name=Acme
app.config.business.street-address=100 Suburban Dr
app.config.business.CITY=Newark
app.config.business.State=DE
app.config.business.zip_code=19711
app.config.business.notess=This is a property name typo
```

- kebab-case `street-address` matched Java camelCase `streetAddress`

- UPPERCASE `CITY` matched Java camelCase `city`

- UpperCamelCase `State` matched Java camelCase `state`

- snake_case `zip_code` matched Java camelCase `zipCode`

- typo `notess` does not match Java camelCase `notes`

7.3. Relaxed Binding Example Output

These relaxed bindings are shown in the following output. However, the `note` attribute is an example that there is no magic when it comes to correcting typo errors. The extra character in `notess` prevented a mapping to the `notes` attribute. The IDE/metadata can help avoid the error and validation can identify when the error exists.

```
$ java -jar target/appconfig-configproperties-example-*-SNAPSHOT-bootexec.jar
...
businessProperties=BusinessProperties(name=Acme, streetAddress=100 Suburban Dr,
city=Newark, state=DE, zipCode=19711, notes=null)
```

Chapter 8. Nested Properties

The previous examples used a flat property model. That may not always be the case. In this example we will look into mapping nested properties.

Nested Properties Example

```
①
app.config.corp.name=Acme
②
app.config.corp.address.street=100 Suburban Dr
app.config.corp.address.city=Newark
app.config.corp.address.state=DE
app.config.corp.address.zip=19711
```

① `name` is part of a flat property model below `corp`

② `address` is a container of nested properties

8.1. Nested Properties JavaBean Mapping

The mapping of the nested class is no surprise. We supply a JavaBean to hold their nested properties and reference it from the host/outer-class.

Nested Property Mapping

```java
...
@Data
@ConstructorBinding
public class AddressProperties {
    private final String street;
    @NotNull
    private final String city;
    @NotNull
    private final String state;
    @NotNull
    private final String zip;
}
```

> ℹ️ In this specific case we are using a read-only JavaBean and need to supply the `@ConstructorBinding` annotation.

8.2. Nested Properties Host JavaBean Mapping

The host class (`CorporateProperties`) declares the base property prefix and a reference (`address`) to the nested class.

Host Property Mapping

```
...
import org.springframework.boot.context.properties.NestedConfigurationProperty;

@ConfigurationProperties("app.config.corp")
@ConstructorBinding
@Data
@Validated
public class CorporationProperties {
    @NotNull
    private final String name;
    @NestedConfigurationProperty //needed for metadata
    @NotNull
    //@Valid
    private final AddressProperties address;
```

> ℹ The `@NestedConfigurationProperty` is only supplied to generate correct metadata — otherwise only a single `address` property will be identified to exist within the generated metadata.

> ℹ The validation initiated by the `@Validated` annotation seems to automatically propagate into the nested `AddressProperties` class without the need to add `@Valid` annotation.

8.3. Nested Properties Output

The defined properties are populated within the host and nested bean and accessible to components within the application.

Nested Property Example Output

```
$ java -jar target/appconfig-configproperties-example-*-SNAPSHOT-bootexec.jar
...
corporationProperties=CorporationProperties(name=Acme,
    address=AddressProperties(street=null, city=Newark, state=DE, zip=19711))
```

Chapter 9. Property Arrays

As the previous example begins to show, property mapping can begin to get complex. I won't demonstrate all of them. Please consult documentation available on the Internet for a complete view. However, I will demonstrate an initial collection mapping to arrays to get started going a level deeper.

In this example, RouteProperties hosts a local name property and a list of stops that are of type AddressProperties that we used before.

Property Array JavaBean Mapping

```
...
@ConfigurationProperties("app.config.route")
@ConstructorBinding
@Data
@Validated
public class RouteProperties {
    @NotNull
    private String name;
    @NestedConfigurationProperty
    @NotNull
    @Size(min = 1)
    private List<AddressProperties> stops;  ①
...
```

① RouteProperties hosts list of stops as AddressProperties

9.1. Property Arrays Definition

The above can be mapped using a properties format.

Property Arrays Example Properties Definition

```
# application.properties
app.config.route.name: Superbowl
app.config.route.stops[0].street: 1101 Russell St
app.config.route.stops[0].city: Baltimore
app.config.route.stops[0].state: MD
app.config.route.stops[0].zip: 21230
app.config.route.stops[1].street: 347 Don Shula Drive
app.config.route.stops[1].city: Miami
app.config.route.stops[1].state: FLA
app.config.route.stops[1].zip: 33056
```

However, it may be easier to map using YAML.

Property Arrays Example YAML Definition

```yaml
# application.yml
app:
  config:
    route:
      name: Superbowl
      stops:
        - street: 1101 Russell St
          city: Baltimore
          state: MD
          zip: 21230
        - street: 347 Don Shula Drive
          city: Miami
          state: FLA
          zip: 33056
```

9.2. Property Arrays Output

Injecting that into our application and printing the state of the bean (with a little formatting) produces the following output showing that each of the `stops` were added to the `route` using the `AddressProperty`.

Property Arrays Example Output

```
$ java -jar target/appconfig-configproperties-example-*-SNAPSHOT-bootexec.jar
...
routeProperties=RouteProperties(name=Superbowl, stops=[
  AddressProperties(street=1101 Russell St, city=Baltimore, state=MD, zip=21230),
  AddressProperties(street=347 Don Shula Drive, city=Miami, state=FLA, zip=33056)
])
```

Chapter 10. System Properties

Note that Java properties can come from several sources and we are able to map them from standard Java system properties as well.

The following example shows mapping three (3) system properties: `user.name`, `user.home`, and `user.timezone` to a `@ConfigurationProperties` class.

Example System Properties JavaBean

```
@ConfigurationProperties("user")
@ConstructorBinding
@Data
public class UserProperties {
    @NotNull
    private final String name;  ①
    @NotNull
    private final String home;  ②
    @NotNull
    private final String timezone;  ③
```

① mapped to SystemProperty `user.name`

② mapped to SystemProperty `user.home`

③ mapped to SystemProperty `user.timezone`

10.1. System Properties Usage

Injecting that into our components give us access to mapped properties and, of course, access to them using standard getters and not just `toString()` output.

Example System Properties Usage

```
@Component
public class AppCommand implements CommandLineRunner {
...
    @Autowired
    private UserProperties userProps;

    public void run(String... args) throws Exception {
...
        System.out.println(userProps);  ①
        System.out.println("user.home=" + userProps.getHome());  ②
```

① output `UserProperties toString`

② get specific value mapped from `user.home`

System Properties Example Output

```
$ java -jar target/appconfig-configproperties-example-*-SNAPSHOT-bootexec.jar
...
UserProperties(name=jim, home=/Users/jim, timezone=America/New_York)
user.home=/Users/jim
```

```
$ java -jar target/appconfig-configproperties-example-*-SNAPSHOT-bootexec.jar
...
UserProperties(name=jim, home=/Users/jim, timezone=America/New_York)
user.home=/Users/jim
```

Chapter 11. @ConfigurationProperties Class Reuse

The examples to date have been singleton values mapped to one root source. However, as we saw with `AddressProperties`, we could have multiple groups of properties with the same structure and different root prefix.

In the following example we have two instances of person. One has the prefix of `owner` and the other `manager`, but they both follow the same structural schema.

Example Properties with Common Structure

```
# application.yml
owner: ①
  name: Steve Bushati
  address:
    city: Millersville
    state: MD
    zip: 21108

manager: ①
  name: Eric Decosta
  address:
    city: Owings Mills
    state: MD
    zip: 21117
```

① `owner` and `manager` root prefixes both follow the same structural schema

11.1. @ConfigurationProperties Class Reuse Mapping

We would like two (2) bean instances that represent their respective person implemented as one JavaBean class. We can structurally map both to the same class and create two instances of that class. However when we do that — we can no longer apply the `@ConfigurationProperties` annotation and prefix to the bean class because the prefix will be instance-specific

@ConfigurationProperties Class Reuse JavaBean Mapping

```
//@ConfigurationProperties("???") multiple prefixes mapped  ①
@Data
@Validated
public class PersonProperties {
    @NotNull
    private String name;
    @NestedConfigurationProperty
    @NotNull
    private AddressProperties address;
```

① unable to apply root prefix-specific `@ConfigurationProperties` to class

11.2. @ConfigurationProperties @Bean Factory

We can solve the issue of having two (2) separate leading prefixes by adding a `@Bean` factory method for each use and we can use our root-level application class to host those factory methods.

@Bean Factory Methods for Separate Property Root Prefixes

```java
@SpringBootApplication
@ConfigurationPropertiesScan
public class ConfigurationPropertiesApp {
...
    @Bean
    @ConfigurationProperties("owner") ②
    public PersonProperties ownerProps() {
        return new PersonProperties(); ①
    }

    @Bean
    @ConfigurationProperties("manager") ②
    public PersonProperties managerProps() {
        return new PersonProperties(); ①
    }
```

① `@Bean` factory method returns JavaBean instance to use

② Spring populates the JavaBean according to the `ConfigurationProperties` annotation

> ℹ️ We are no longer able to use read-only JavaBeans when using the `@Bean` factory method in this way. We are returning a default instance for Spring to populate based on the specified `@ConfigurationProperties` prefix of the factory method.

11.3. Injecting ownerProps

Taking this one instance at a time, when we inject an instance of `PersonProperties` into the `ownerProps` attribute of our component, the `ownerProps` `@Bean` factory is called and we get the information for our owner.

Owner Person Injection

```java
@Component
public class AppCommand implements CommandLineRunner {
    @Autowired
    private PersonProperties ownerProps;
```

Owner Person Injection Result

```
$ java -jar target/appconfig-configproperties-example-*-SNAPSHOT-bootexec.jar
```

```
...
PersonProperties(name=Steve Bushati, address=AddressProperties(street=null,
city=Millersville, state=MD, zip=21108))
```

Great! However, there was something subtle there that allowed things to work.

11.4. Injection Matching

Spring had two `@Bean` factory methods to chose from to produce an instance of `PersonProperties`.

Two PersonProperties Sources

```
    @Bean
    @ConfigurationProperties("owner")
    public PersonProperties ownerProps() {
...
    @Bean
    @ConfigurationProperties("manager")
    public PersonProperties managerProps() {
...
```

The `ownerProps` `@Bean` factory method name happened to match the `ownerProps` Java attribute name and that resolved the ambiguity.

Target Attribute Name for Injection provides Qualifier

```
@Component
public class AppCommand implements CommandLineRunner {
    @Autowired
    private PersonProperties ownerProps; ①
```

① Attribute name of injected bean matches `@Bean` factory method name

11.5. Ambiguous Injection

If we were to add the `manager` and specifically not make the two names match, there will be ambiguity as to which `@Bean` factory to use. The injected attribute name is `manager` and the desired `@Bean` factory method name is `managerProps`.

Manager Person Injection

```
@Component
public class AppCommand implements CommandLineRunner {
    @Autowired
    private PersonProperties manager; ①
```

① Java attribute name does not match `@Bean` factory method name

```
$ java -jar target/appconfig-configproperties-example-*-SNAPSHOT-bootexec.jar
**************************
APPLICATION FAILED TO START
**************************
Description:

Field manager in info.ejava.examples.app.config.configproperties.AppCommand
    required a single bean, but 2 were found:
      - ownerProps: defined by method 'ownerProps' in
        info.ejava.examples.app.config.configproperties.ConfigurationPropertiesApp
      - managerProps: defined by method 'managerProps' in
        info.ejava.examples.app.config.configproperties.ConfigurationPropertiesApp


Action:

Consider marking one of the beans as @Primary, updating the consumer to accept
multiple beans,
or using @Qualifier to identify the bean that should be consumed
```

11.6. Injection @Qualifier

As the error message states, we can solve this one of several ways. The `@Qualifier` route is mostly what we want and can do that one of at least three ways.

11.7. way1: Create Custom @Qualifier Annotation

Create a custom `@Qualifier` annotation and apply that to the `@Bean` factory and injection point.

- benefits: eliminates string name matching between factory mechanism and attribute

- drawbacks: new annotation must be created and applied to both factory and injection point

Custom @Manager Qualifier Annotation

```java
package info.ejava.examples.app.config.configproperties.properties;

import org.springframework.beans.factory.annotation.Qualifier;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;


@Qualifier
@Target({ElementType.METHOD, ElementType.FIELD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
public @interface Manager {
}
```

@Manager Annotation Applied to @Bean Factory Method

```
@Bean
@ConfigurationProperties("manager")
@Manager ①
public PersonProperties managerProps() {
    return new PersonProperties();
}
```

① @Manager annotation used to add additional qualification beyond just type

@Manager Annotation Applied to Injection Point

```
@Autowired
private PersonProperties ownerProps;
@Autowired
@Manager ①
private PersonProperties manager;
```

① @Manager annotation is used to disambiguate the factory choices

11.8. way2: @Bean Factory Method Name as Qualifier

Use the name of the @Bean factory method as a qualifier.

- benefits: no custom qualifier class required and factory signature does not need to be modified

- drawbacks: text string must match factory method name

```
@Autowired
private PersonProperties ownerProps;
@Autowired
@Qualifier("managerProps") ①
private PersonProperties manager;
```

① @Bean factory name is being applied as a qualifier versus defining a type

11.9. way3: Match @Bean Factory Method Name

Change the name of the injected attribute to match the @Bean factory method name

- benefits: simple and properly represents the semantics of the singleton property
- drawbacks: injected attribute name must match factory method name

PersonProperties Sources

```
    @Bean
    @ConfigurationProperties("owner")
    public PersonProperties ownerProps() {
```

```
...
    @Bean
    @ConfigurationProperties("manager")
    public PersonProperties managerProps() {
...
```

Injection Points

```
    @Autowired
    private PersonProperties ownerProps;
    @Autowired
    private PersonProperties managerProps; ①
```

① Attribute name of injected bean matches `@Bean` factory method name

11.10. Ambiguous Injection Summary

Factory choices and qualifiers is a whole topic within itself. However, this set of examples showed how `@ConfigurationProperties` can leverage `@Bean` factories to assist in additional complex property mappings. We likely will be happy taking the simple `way3` solution but it is good to know there is an easy way to use a `@Qualifier` annotation when we do not want to rely on a textual name match.

Chapter 12. Summary

In this module we

- mapped properties from property sources to JavaBean classes annotated with `@ConfigurationProperties` and injected them into component classes
- generated property metadata that can be used by IDEs to provide an aid to configuring properties
- implemented a read-only JavaBean
- defined property validation using Jakarta EE Java Validation framework
- generated boilerplate JavaBean constructs with the Lombok library
- demonstrated how relaxed binding can lead to more flexible property names
- mapped flat/simple properties, nested properties, and collections of properties
- leveraged custom `@Bean` factories to reuse common property structure for different root instances
- leveraged `@Qualifier` s in order to map or disambiguate injections