Chapter 5. Configuring and Inspecting Your Spring Boot App

There are many things that can go wrong with any application, and some of these many things may even have simple solutions. With the rare exception of an occasional good guess, however, one must determine the root cause of a problem before it is possible to truly solve it.

Debugging Java or Kotlin applications—or any other applications, for that matter—is a fundamental skill that every developer should learn very early on in their career and refine and expand throughout. I don't find that to be the case universally, so if you haven't already become handy with the debugging capabilities of your language and tools of choice, please explore the options at your disposal as soon as possible. It really is important in everything you develop and can save you inordinate amounts of time.

That said, debugging code is only one level of establishing, identifying, and isolating behaviors manifested within your application. As applications become more dynamic and distributed, developers often need to do the following:

- Configure and reconfigure applications dynamically
- Determine/confirm current settings and their origins
- Inspect and monitor application environment and health indicators
- Temporarily adjust logging levels of live apps to identify root causes

This chapter demonstrates how to use Spring Boot's built-in configuration capabilities, its Autoconfiguration Report, and Spring Boot Actuator to create, identify, and modify application environment settings flexibly and dynamically.

CODE CHECKOUT CHECKUP

Please check out branch *chapter5begin* from the code repository to begin.

Application Configuration

No app is an island.

Most times when I say that, it's to point out the truism that in nearly every case, an application doesn't provide all of its utility without interaction with other applications/services. But there is another meaning that is just as true: no application can be as useful without access to its environment, in one form or another. A static, unconfigurable app is rigid, inflexible, and hobbled.

Spring Boot applications supply a variety of powerful mechanisms for developers to dynamically configure and reconfigure their applications, even while the app is running. These mechanisms leverage the Spring `Environment` to manage configuration properties from all sources, including the following:

- Spring Boot Developer Tools (devtools) global settings properties in the *$HOME/.config/spring-boot* directory when devtools is active.
- `@TestPropertySource` annotations on tests.
- `properties` attribute on tests, available on `@SpringBootTest` and the various test annotations for testing an application slice.
- Command line arguments.
- Properties from `SPRING_APPLICATION_JSON` (inline JSON embedded in an environment variable or system property).
- `ServletConfig` init parameters.
- `ServletContext` init parameters.
- JNDI attributes from *java:comp/env*.
- Java System properties (`System.getProperties()`).
- OS environment variables.
- A `RandomValuePropertySource` that has properties only in `random.*`.
- Profile-specific application properties outside of the packaged jar (*application-{profile}.properties* and YAML variants).
- Profile-specific application properties packaged inside the jar (*application-{profile}.properties* and YAML variants).
- Application properties outside of the packaged jar (*application.properties* and YAML variants).
- Application properties packaged inside the jar (*application.properties* and YAML variants).
- `@PropertySource` annotations on `@Configuration` classes; note that such property sources are not added to the Environment until the application context is refreshed, which is too late to configure certain properties read before refresh begins, such as `logging.*` and `spring.main.*`.
- Default properties specified by setting `SpringApplication.setDefaultProperties`.

NOTE

The preceding property sources are listed in decreasing order of precedence: properties from sources higher in the list supersede identical properties from lower sources.[1]

All of these can be extremely useful, but I'll choose a few in particular for the code scenarios in this chapter:

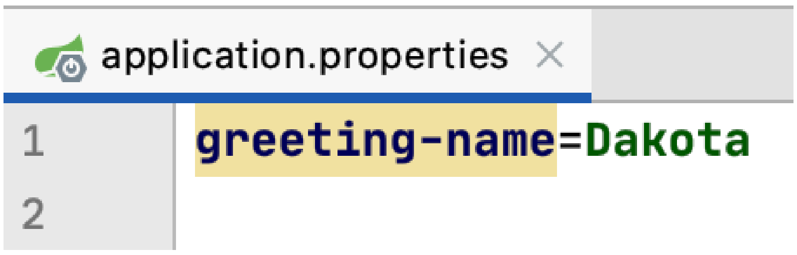
- Command line arguments
- OS environment variables
- Application properties packaged inside the jar (*application.properties* and YAML variants)

Let's begin with properties defined in the app's *application.properties* file and work our way up the food chain.

@Value

The `@Value` annotation is perhaps the most straightforward approach to ingesting configuration settings into your code. Built around pattern-matching and the Spring Expression Language (SpEL), it's simple and powerful.

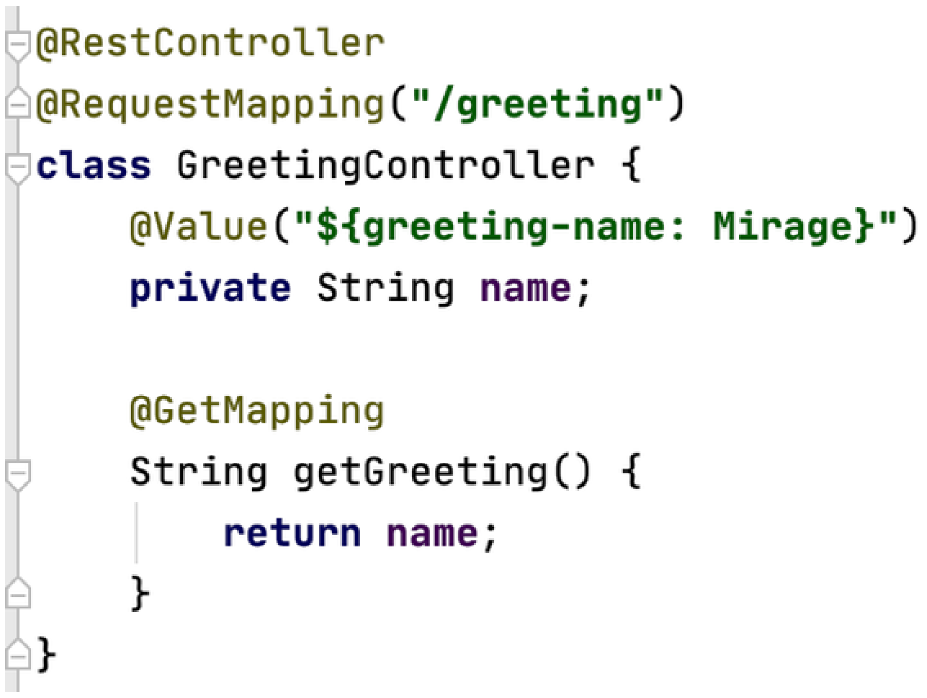
I'll start by defining a single property in our application's *application.properties* file, as shown in Figure 5-1.



```
greeting-name=Dakota
```

Figure 5-1. Defining `greeting-name` in application.properties

To show how to put this property to use, I create an additional `@RestController` within the application to handle tasks related to greeting application users, as demonstrated in Figure 5-2.

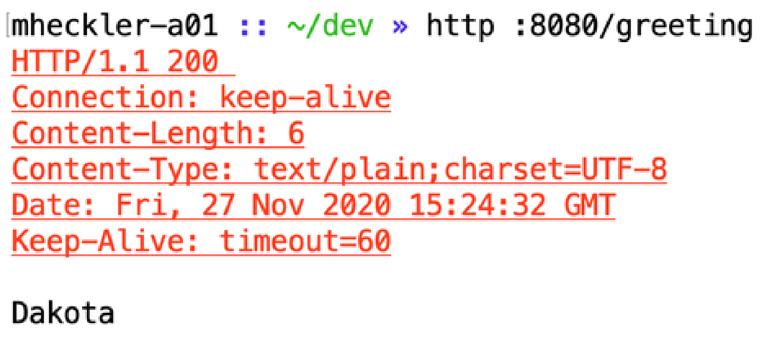


```
@RestController
@RequestMapping("/greeting")
class GreetingController {
    @Value("${greeting-name: Mirage}")
    private String name;

    @GetMapping
    String getGreeting() {
        return name;
    }
}
```

Figure 5-2. Greeting `@RestController` class

Note that the `@Value` annotation applies to the `name` member variable and accepts a single parameter of type `String` called `value`. I define the `value` using SpEL, placing the variable name (as the expression to evaluate) between the delimiters `${` and `}`. One other thing of note: SpEL allows for a default value after the colon—in this example, “Mirage”—for cases in which the variable isn’t defined in the app `Environment`.

Upon executing the application and querying the */greeting* endpoint, the app responds with “Dakota” as expected, shown in Figure 5-3.



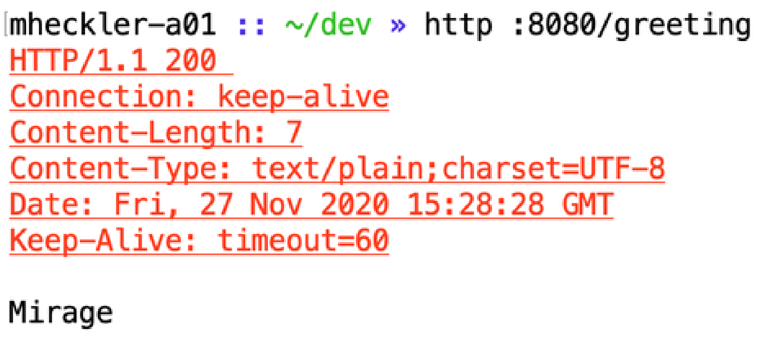
```
mheckler-a01 :: ~/dev » http :8080/greeting
HTTP/1.1 200
Connection: keep-alive
Content-Length: 6
Content-Type: text/plain;charset=UTF-8
Date: Fri, 27 Nov 2020 15:24:32 GMT
Keep-Alive: timeout=60

Dakota
```

Figure 5-3. Greeting response with defined property value

To verify the default value is being evaluated, I comment out the following line in *application.properties* with a `#` as follows and restart the application:

```
#greeting-name=Dakota
```

Querying the *greeting* endpoint now results in the response shown in Figure 5-4. Since `greeting-name` is no longer defined in any source for the application's `Environment`, the default value of "Mirage" kicks in, as expected.

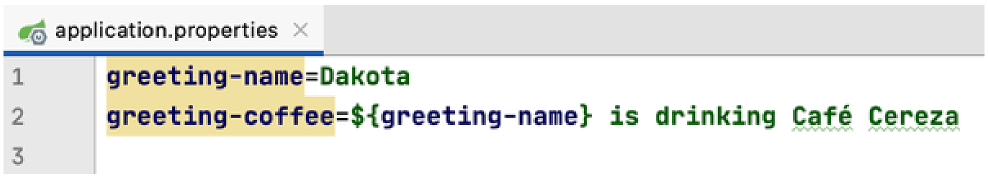


```
mheckler-a01 :: ~/dev » http :8080/greeting
HTTP/1.1 200
Connection: keep-alive
Content-Length: 7
Content-Type: text/plain;charset=UTF-8
Date: Fri, 27 Nov 2020 15:28:28 GMT
Keep-Alive: timeout=60

Mirage
```

Figure 5-4. Greeting response with default value

Using `@Value` with roll-your-own properties provides another useful capability: the value of one property can be derived/built using the value of another.

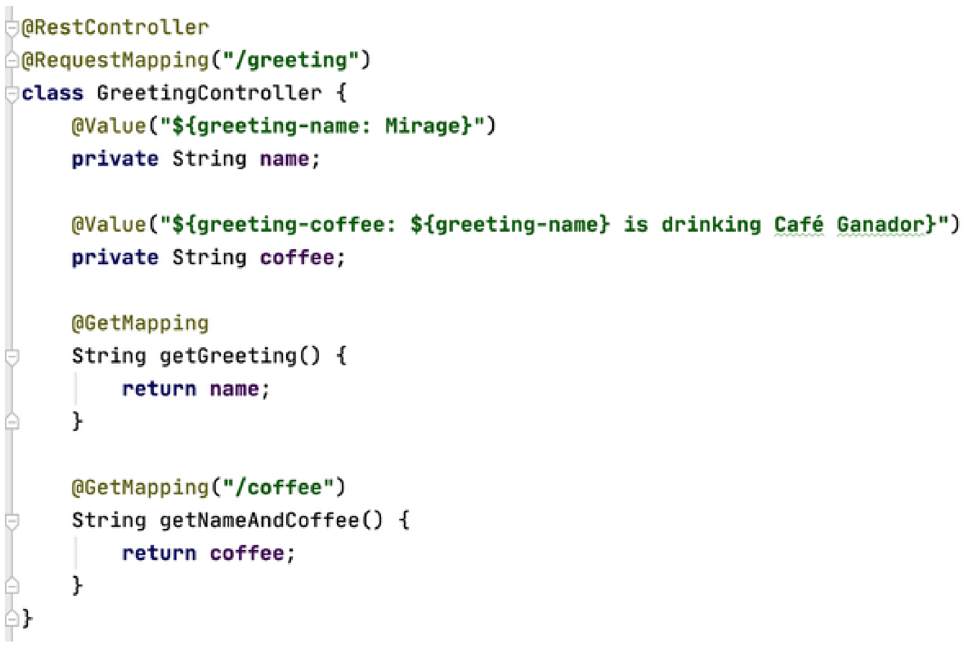
To demonstrate how property nesting works, we'll need at least two properties. I create a second property `greeting-coffee` in *application.properties*, as in Figure 5-5.



```
greeting-name=Dakota
greeting-coffee=${greeting-name} is drinking Café Cereza
```

Figure 5-5. Property value feeding another property

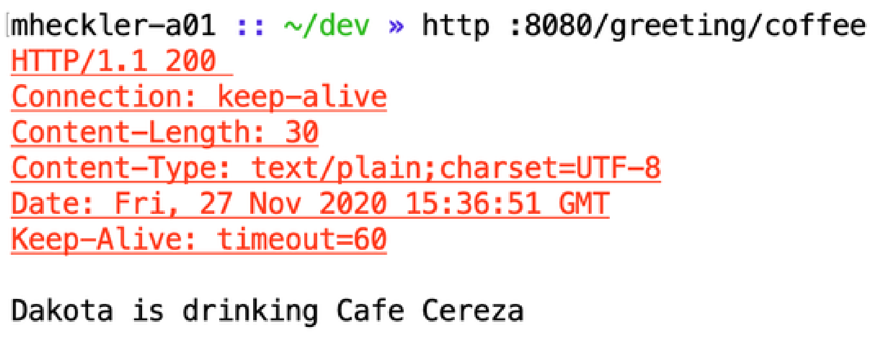
Next, I add a bit of code to our `GreetingController` to represent a coffee-fied greeting and an endpoint we can access to see the results. Note that I provide a default value for `coffee`'s value as well, per Figure 5-6.



```
@RestController
@RequestMapping("/greeting")
class GreetingController {
    @Value("${greeting-name: Mirage}")
    private String name;

    @Value("${greeting-coffee: ${greeting-name} is drinking Café Ganador}")
    private String coffee;

    @GetMapping
    String getGreeting() {
        return name;
    }

    @GetMapping("/coffee")
    String getNameAndCoffee() {
        return coffee;
    }
}
```

Figure 5-6. Adding a coffee greeting to `GreetingController`

To verify the proper outcome, I restart the application and query the new */greeting/coffee* endpoint, resulting in the output shown in Figure 5-7. Note that since both properties in question are defined in *application.properties*, the values displayed are consistent with those values' definitions.



```
mheckler-a01 :: ~/dev » http :8080/greeting/coffee
HTTP/1.1 200
Connection: keep-alive
Content-Length: 30
Content-Type: text/plain;charset=UTF-8
Date: Fri, 27 Nov 2020 15:36:51 GMT
Keep-Alive: timeout=60

Dakota is drinking Cafe Cereza
```

Figure 5-7. Querying the coffee greeting endpoint

As with all things in life and software development, `@Value` does have some limitations. Since we provided a default value for the `greeting-coffee` property, we can comment out its definition in *application.properties*, and the `@Value` annotation still properly processes its (default) value using the `coffee` member variable within `GreetingController`. However, commenting out both `greeting-name` and `greeting-coffee` in the properties file results in no `Environment` source actually defining them, further resulting in the following error when the application attempts to initialize the `GreetingController` bean using a reference to (now-undefined) `greeting-name` within `greeting-coffee`:

```
org.springframework.beans.factory.BeanCreationException:
    Error creating bean with name 'greetingController':
        Injection of autowired dependencies failed; nested exception is
        java.lang.IllegalArgumentException:
            Could not resolve placeholder 'greeting-name' in value
            "greeting-coffee: ${greeting-name} is drinking Cafe Ganador"
```

NOTE

Full stacktrace removed for brevity and clarity.

Another limitation with properties defined in *application.properties* and used solely via `@Value`: they aren't recognized by the IDE as being used by the application, as they're only referenced in the code within quote-delimited `String` variables; as such, there is no direct tie-in to code. Of course, developers can visually check for correct spelling of property names and usage, but this is entirely manual and thus more prone to error.

As you might imagine, a typesafe and tool-verifiable mechanism for property use and definition would be a better all-around option.

@ConfigurationProperties

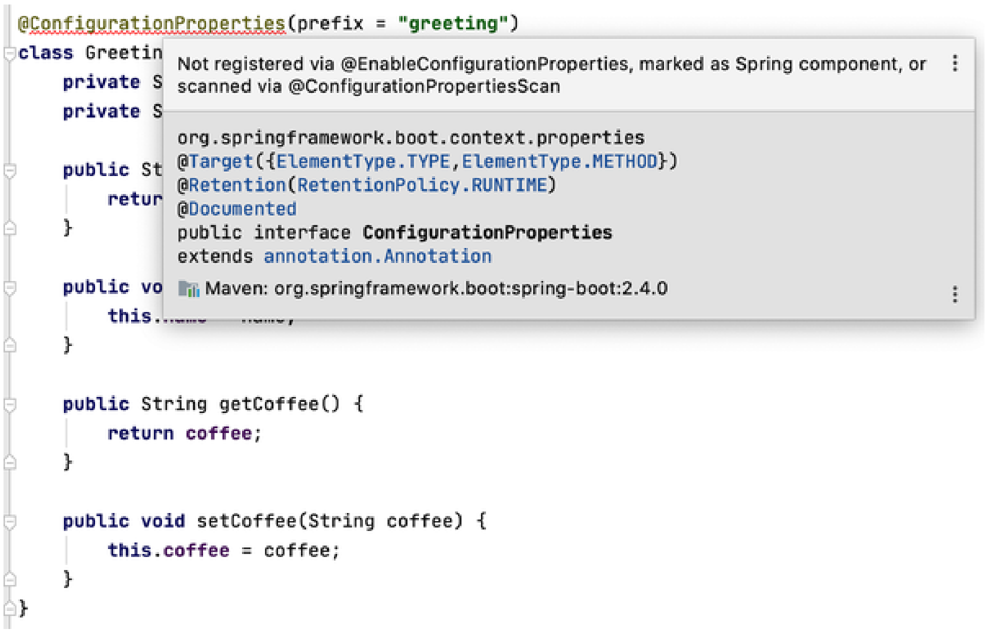
Appreciating the flexibility of `@Value` but recognizing its shortcomings, the Spring team created `@ConfigurationProperties`. Using `@ConfigurationProperties`, a developer can define properties, group related properties, and reference/use them in a tool-verifiable and type-safe way.

For example, if a property is defined in an app's *application.properties* file that isn't used in code, the developer will see the name highlighted to flag it as a confirmed unused property. Similarly, if the property is defined as a `String` but associated with a differently typed member variable, the IDE will point out the type mismatch. These are valuable helps that catch simple, but frequent, mistakes.

To demonstrate how to put `@ConfigurationProperties` to work, I'll start by defining a POJO to encapsulate the desired related properties: in this case, our `greeting-name` and `greeting-coffee` properties previously referenced. As shown in the code that follows, I create a `Greeting` class to hold both:

```
class Greeting {
    private String name;
    private String coffee;

    public String getName() {
        return name;
    }

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

    public String getCoffee() {
        return coffee;
    }

    public void setCoffee(String coffee) {
        this.coffee = coffee;
    }
}
```

In order to register `Greeting` to manage configuration properties, I add the `@ConfigurationProperties` annotation shown in Figure 5-8 and specify the `prefix` to use for all `Greeting` properties. This annotation prepares the class for use only with configuration properties; the application also must be told to process classes annotated in such manner for properties to include in the application `Environment`. Note the helpful error message that results:



```
@ConfigurationProperties(prefix = "greeting")
class Greetin
    private S
    private S

    public St
        retur
    }

    public vo
        this.
    }

    public String getCoffee() {
        return coffee;
    }

    public void setCoffee(String coffee) {
        this.coffee = coffee;
    }
}
```

```
Not registered via @EnableConfigurationProperties, marked as Spring component, or
scanned via @ConfigurationPropertiesScan

org.springframework.boot.context.properties
@Target({ElementType.TYPE,ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public interface ConfigurationProperties
extends annotation.Annotation
Maven: org.springframework.boot:spring-boot:2.4.0
```

Figure 5-8. Annotation and error

Instructing the application to process `@ConfigurationProperties` classes and add their properties to the app's `Environment` is, in most cases, best accomplished by adding the `@ConfigurationPropertiesScan` annotation to the main application class, as demonstrated here:

```
@SpringBootApplication
@ConfigurationPropertiesScan
public class SburRestDemoApplication {

    public static void main(String[] args) {
        SpringApplication.run(SburRestDemoApplication.class, args);
    }

}
```

NOTE

The exceptions to the rule of having Boot scan for `@ConfigurationProperties` classes are if you need to enable certain `@ConfigurationProperties` classes conditionally or if you are creating your own autoconfiguration. In all other cases, however, `@ConfigurationPropertiesScan` should be used to scan for and enable `@ConfigurationProperties` classes in like manner to Boot's component scanning mechanism.

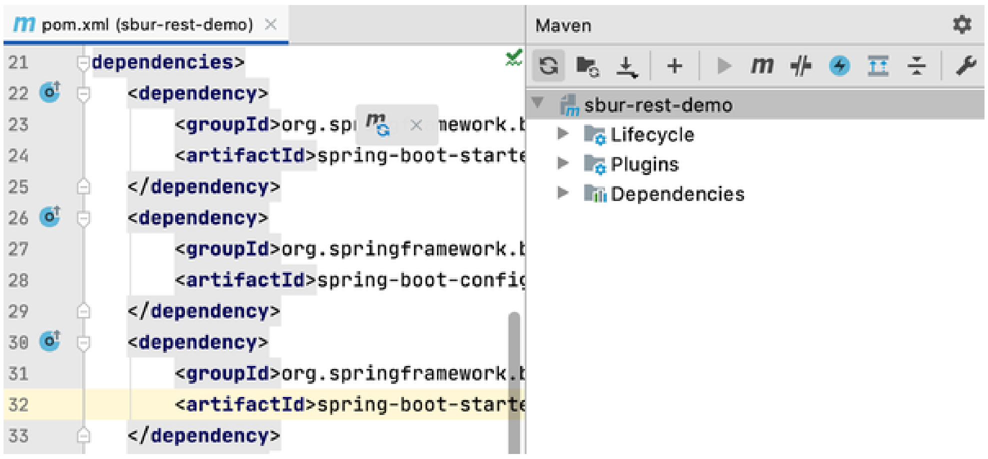
In order to generate metadata using the annotation processor, enabling the IDE to connect the dots between `@ConfigurationProperties` classes and related properties defined in the *application.properties* file, I add the following dependency to the project's *pom.xml* build file:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-configuration-processor</artifactId>
    <optional>true</optional>
</dependency>
```

NOTE

This dependency also can be selected and added automatically from the Spring Initializr at the time of project creation.

Once the configuration processor dependency is added to the build file, it's necessary to refresh/reimport the dependencies and rebuild the project to take advantage of them. To reimport deps, I open the Maven menu in IntelliJ and click the Reimport button at the top left, as shown in Figure 5-9.



```
<dependencies>
    <dependency>
        <groupId>org.spr         nework.b
        <artifactId>spring-boot-starte
    </dependency>
    <dependency>
        <groupId>org.springframework.b
        <artifactId>spring-boot-config
    </dependency>
    <dependency>
        <groupId>org.springframework.b
        <artifactId>spring-boot-starte
    </dependency>
```

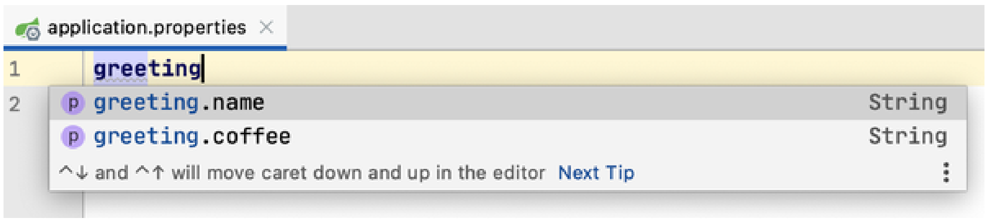
Figure 5-9. Reimporting project dependencies

NOTE

Unless the option is disabled, IntelliJ also presents a small button over the changed *pom.xml* to allow for a quick reimport without needing to open the Maven menu. The overlaid reimport button, a small *m* with a circular arrow on its bottom left portion, can be seen in Figure 5-9 hovering over the first dependency's `<groupid>` entry; it disappears when reimport is complete.

Once dependencies are updated, I rebuild the project from the IDE to incorporate the configuration processor.

Now, to define some values for these properties. Returning to *application.properties*, when I begin typing `greeting`, the IDE helpfully shows property names that match, as demonstrated in Figure 5-10.



```
greeting
  greeting.name        String
  greeting.coffee      String
```

application.properties
^↓ and ^↑ will move caret down and up in the editor Next Tip

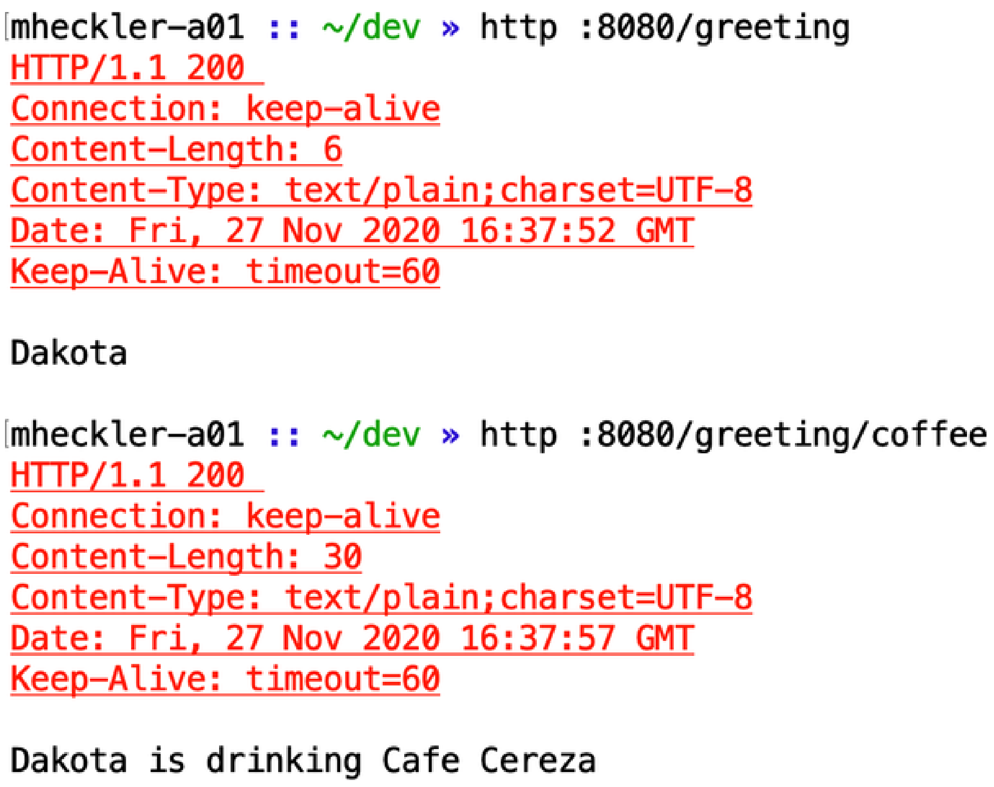
Figure 5-10. Full IDE property support for `@ConfigurationProperties`

To use these properties instead of the ones we were using before, a bit of refactoring is required.

I can do away entirely with `GreetingController`'s own member variables `name` and `coffee` along with their `@Value` annotations; instead, I create a member variable for the `Greeting` bean that now manages `greeting.name` and `greeting.coffee` properties and inject it into `GreetingController` via constructor injection, as shown in the following code:

```
@RestController
@RequestMapping("/greeting")
class GreetingController {
    private final Greeting greeting;

    public GreetingController(Greeting greeting) {
        this.greeting = greeting;
    }

    @GetMapping
    String getGreeting() {
        return greeting.getName();
    }

    @GetMapping("/coffee")
    String getNameAndCoffee() {
        return greeting.getCoffee();
    }
}
```

Running the application and querying the *greeting* and *greeting/coffee* endpoints results in the outcomes captured in Figure 5-11.



```
mheckler-a01 :: ~/dev » http :8080/greeting
HTTP/1.1 200
Connection: keep-alive
Content-Length: 6
Content-Type: text/plain;charset=UTF-8
Date: Fri, 27 Nov 2020 16:37:52 GMT
Keep-Alive: timeout=60

Dakota

mheckler-a01 :: ~/dev » http :8080/greeting/coffee
HTTP/1.1 200
Connection: keep-alive
Content-Length: 30
Content-Type: text/plain;charset=UTF-8
Date: Fri, 27 Nov 2020 16:37:57 GMT
Keep-Alive: timeout=60

Dakota is drinking Cafe Cereza
```

Figure 5-11. Retrieving `Greeting` properties

Properties managed by an `@ConfigurationProperties` bean still obtain their values from the `Environment` and all of its potential sources; the only significant thing missing in comparison to `@Value`-based properties is the ability to specify a default value at the annotated member variable. That is less of a sacrifice than it might appear to be at first glance because the app's *application.properties* file typically serves as the place for defining sensible defaults for an application. If there is a need for different property values to accommodate different deployment environments, those environment-specific values are ingested into the application's `Environment` via other sources, e.g., environment variables or command line parameters. In short, `@ConfigurationProperties` simply enforces the better practice for default property values.

Potential Third-Party Option

A further extension to the already impressive usefulness of `@ConfigurationProperties` is the ability to wrap third-party components and incorporate their properties into the application's `Environment`. To demonstrate how, I create a POJO to simulate a component that might be incorporated into the application. Note that in typical use cases where this feature is handiest, one would add an external dependency to the project and consult the component's documentation to determine the class from which to create a Spring bean, rather than creating one by hand as I do here.

In the code listing that follows, I create the simulated third-party component called `Droid` with two properties—`id` and `description`—and their associated accessor and mutator methods:

```
class Droid {
    private String id, description;

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }
}
```

The next step falls into place the same way a true third-party component would: instantiating the component as a Spring bean. Spring beans can be created from defined POJOs in several ways, but the one most appropriate to this particular use case is to create an `@Bean`-annotated method within a class annotated with `@Configuration`, either directly or via a meta-annotation.

One meta-annotation that incorporates `@Configuration` within its definition is `@SpringBootApplication`, which is found on the main application class. That's why developers often place bean creation methods there.

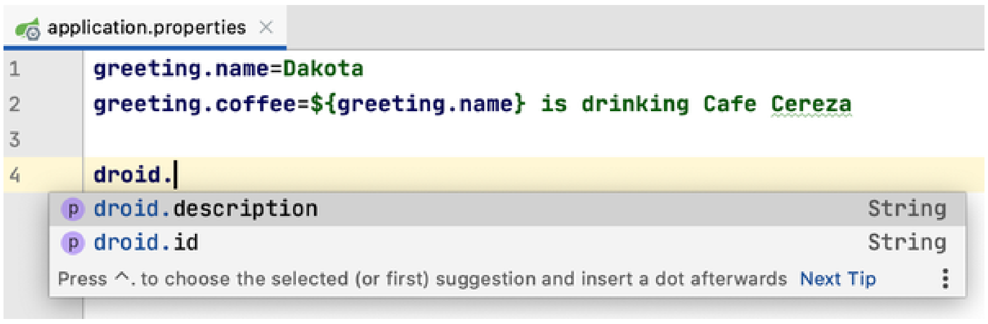
NOTE

Within IntelliJ and most other IDEs and advanced text editors with solid Spring support, it's possible to drill into Spring meta-annotations to explore the annotations nested within. In IntelliJ, Cmd+LeftMouseClick (on MacOS) will expand the annotation. `@SpringBootApplication` includes `@SpringBootConfiguration`, which includes `@Configuration`, making only two degrees of separation from Kevin Bacon.

In the following code listing I demonstrate the bean creation method and the requisite `@ConfigurationProperties` annotation and `prefix` parameter, indicating that `Droid` properties should be incorporated within the `Environment` under the top-level property grouping `droid`:

```
@SpringBootApplication
@ConfigurationPropertiesScan
public class SburRestDemoApplication {

    public static void main(String[] args) {
        SpringApplication.run(SburRestDemoApplication.class, args);
    }

    @Bean
    @ConfigurationProperties(prefix = "droid")
    Droid createDroid() {
        return new Droid();
    }
}
```

As before, it is necessary to rebuild the project for the configuration processor to detect the properties exposed by this new source of configuration properties. After executing a build, we can return to *application.properties* and see that both `droid` properties have now surfaced complete with type information, as per Figure 5-12.



application.properties

```
greeting.name=Dakota
greeting.coffee=${greeting.name} is drinking Cafe Cereza

droid.
```

p droid.description String
p droid.id String
Press ^. to choose the selected (or first) suggestion and insert a dot afterwards Next Tip

Figure 5-12. `droid` properties and type information now visible in *application.properties*

I assign some default values to `droid.id` and `droid.description` for use as defaults, as shown in Figure 5-13. This is a good habit to adopt for all `Environment` properties, even those obtained from third parties.

Figure 5-13. `droid` properties with default values assigned in *application.properties*

In order to verify that everything works as expected with the `Droid`'s properties, I create a very simple `@RestController` with a single `@GetMapping` method, as shown in the code that follows:

```
@RestController
@RequestMapping("/droid")
class DroidController {
    private final Droid droid;

    public DroidController(Droid droid) {
        this.droid = droid;
    }

    @GetMapping
    Droid getDroid() {
        return droid;
    }
}
```

After building and running the project, I query the new */droid* endpoint and confirm the appropriate response, as indicated in Figure 5-14.

Figure 5-14. Querying the */droid* endpoint to retrieve properties from the Droid

Autoconfiguration Report

As mentioned previously, Boot does a *lot* on behalf of developers via auto-configuration: setting up the application with the beans it needs to fulfill the functionalities that are part and parcel to chosen capabilities, dependencies, and code. Also mentioned earlier is the ability to override any bit of autoconfiguration necessary to implement functionality in a more specific (to your use case) manner. But how can one see what beans are created, what beans aren't created, and what conditions prompted either outcome?

It's a simple matter to produce the autoconfiguration report using the `debug` flag in one of several ways, owing to the flexibility of the JVM:

- Executing the application's jar file with the `--debug` option: `java -jar bootapplication.jar --debug`
- Executing the application's jar file with a JVM parameter: `java -Ddebug=true` `-jar bootapplication.jar`
- Adding `debug=true` to your application's *application.properties* file
- Executing `export DEBUG=true` in your shell (Linux or Mac) or adding it to your Windows environment, then running `java -jar bootapplication.jar`

NOTE

Any way to add an affirmative value for `debug` to the application's `Environment`, as discussed earlier, will provide the same results. These are just more frequently used options.

The autoconfiguration report's section listing positive matches—those conditions that evaluated to true and caused an action to take place—are listed within a section headed by "Positive matches." I've copied that section header here, along with one example of a positive match and its resultant autoconfiguration action:

```
============================
CONDITIONS EVALUATION REPORT
============================


Positive matches:
-----------------
   DataSourceAutoConfiguration matched:
      - @ConditionalOnClass found required classes 'javax.sql.DataSource',
      'org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseType'
      (OnClassCondition)
```

This particular match demonstrates what we expected to happen, although it's always good to confirm the following:

- JPA and H2 are application dependencies.
- JPA works with SQL datasources.

- H2 is an embedded database.
- Classes were found that support embedded SQL datasources.

As a result, `DataSourceAutoConfiguration` is invoked.

Similarly, the “Negative matches” section displays actions not taken by Spring Boot’s autoconfiguration and why, as illustrated in the following:

```
Negative matches:
-----------------
   ActiveMQAutoConfiguration:
      Did not match:
         - @ConditionalOnClass did not find required class
          'javax.jms.ConnectionFactory' (OnClassCondition)
```

In this case, `ActiveMQAutoConfiguration` was not performed because the application didn’t find the JMS `ConnectionFactory` class upon startup.

Another useful tidbit is the section listing “Unconditional classes,” which are created without having to satisfy any conditions. I’ve listed one next that is of particular interest given the previous section:

```
Unconditional classes:
----------------------
    org.springframework.boot.autoconfigure.context
     .ConfigurationPropertiesAutoConfiguration
```

As you can see, `ConfigurationPropertiesAutoConfiguration` is always instantiated to manage any `ConfigurationProperties` created and referenced within a Spring Boot application; it’s integral to every Spring Boot app.

Actuator

> *actuator*
>
> n. One that actuates specifically: a mechanical device for moving or controlling something

The original version of Spring Boot Actuator reached General Availability (GA) in 2014 and was hailed for providing valuable insights into production Boot applications. Providing monitoring and management capabilities of running apps via HTTP endpoints or Java Management Extensions (JMX), Actuator encompasses and exposes all of Spring Boot’s production-ready features.

Completely retooled with the 2.0 version of Spring Boot, Actuator now leverages the Micrometer instrumentation library to provide metrics via a consistent façade from numerous leading monitoring systems, similar to how SLF4J operates with regard to various logging mechanisms. This dramatically extends the scope of things that can be integrated, monitored, and exposed via Actuator within any given Spring Boot application.

To get started with Actuator, I add another dependency to the current project's `pom.xml` dependencies section. As shown in the following snippet, the `spring-boot-starter-actuator` dependency provides the necessary capabilities; to do so, it brings with it both Actuator itself and Micrometer, along with autoconfiguration capabilities to slide into place within a Spring Boot application with nearly zero effort:

```
<dependencies>
    ... (other dependencies omitted for brevity)
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>
</dependencies>
```

After refreshing/reimporting dependencies once more, I rerun the application. With the application running, we can see what information Actuator exposes by default by accessing its primary endpoint. Again, I use HTTPie to accomplish this, as per Figure 5-15.

Figure 5-15. Accessing Actuator endpoint, default configuration

NOTE

All Actuator information is grouped together under the app's */actuator* endpoint by default, but this too is configurable.

This doesn't seem like much information for the fanfare (and fanbase) that Actuator has created. But this terseness is intentional.

Actuator has access to and can expose a great deal of information about running applications. This information can be incredibly useful to developers, operational personnel, and also nefarious individuals who might desire to threaten your application's security. Following Spring Security's de facto goal of *secure by default*, Actuator's autoconfiguration exposes very limited *health* and *info* responses—in fact, *info* defaults to an empty set—that provide an application heartbeat and little else out of the box (OOTB).

As with most things Spring, you can create some pretty sophisticated mechanisms for controlling access to various Actuator data feeds, but there are fast, consistent, and low-friction options available as well. Let's take a look at those now.

It's possible to easily configure Actuator via properties with either a set of included endpoints or a set of excluded endpoints. For simplicity's sake, I choose the inclusion route, adding the following to *application.properties*:

```
management.endpoints.web.exposure.include=env, info, health
```

In this example, I direct the app (and Actuator) to expose only the */actuator/env*, */actuator/info*, and */actuator/health* endpoints (and any subordinate endpoints).

Figure 5-16 confirms the expected outcome upon rerunning the application and querying its */actuator* endpoint.

In order to fully demonstrate Actuator's OOTB capabilities, I can go a step further and disable security entirely *for demonstration purposes only* by using a wildcard with the aforementioned *application.properties* setting:

```
management.endpoints.web.exposure.include=*
```

WARNING

This point is impossible to overstate: security mechanisms for sensitive data should be disabled only for demonstration or verification purposes. *NEVER DISABLE SECURITY FOR PRODUCTION APPLICATIONS*.

Figure 5-16. Accessing the Actuator after specifying endpoints to include

For verification when starting the application, Actuator dutifully reports the number of endpoints it is currently exposing and the root path to reach them—in this case, the default of */actuator*—as shown in the startup report fragment that follows. This is a useful reminder/warning to provide a quick visual check that no more endpoints are being exposed than desired before advancing the application to a target deployment:

```
INFO 22115 --- [           main] o.s.b.a.e.web.EndpointLinksResolver      :
    Exposing 13 endpoint(s) beneath base path '/actuator'
```

To examine all mappings currently accessible via Actuator, simply query the provided Actuator root path to retrieve a full listing:

```
mheckler-a01 :: ~/dev » http :8080/actuator
HTTP/1.1 200
Connection: keep-alive
Content-Type: application/vnd.spring-boot.actuator.v3+json
Date: Fri, 27 Nov 2020 17:43:27 GMT
Keep-Alive: timeout=60
Transfer-Encoding: chunked

{
    "_links": {
        "beans": {
            "href": "http://localhost:8080/actuator/beans",
            "templated": false
        },
        "caches": {
            "href": "http://localhost:8080/actuator/caches",
            "templated": false
        },
        "caches-cache": {
            "href": "http://localhost:8080/actuator/caches/{cache}",
            "templated": true
        },
        "conditions": {
            "href": "http://localhost:8080/actuator/conditions",
            "templated": false
        },
        "configprops": {
            "href": "http://localhost:8080/actuator/configprops",
            "templated": false
        },
        "env": {
            "href": "http://localhost:8080/actuator/env",
            "templated": false
        },
        "env-toMatch": {
            "href": "http://localhost:8080/actuator/env/{toMatch}",
            "templated": true
        },
        "health": {
            "href": "http://localhost:8080/actuator/health",
            "templated": false
        },
        "health-path": {
            "href": "http://localhost:8080/actuator/health/{*path}",
            "templated": true
        },
        "heapdump": {
            "href": "http://localhost:8080/actuator/heapdump",
            "templated": false
        },
        "info": {
            "href": "http://localhost:8080/actuator/info",
            "templated": false
        },
        "loggers": {
            "href": "http://localhost:8080/actuator/loggers",
            "templated": false
        },
        "loggers-name": {
            "href": "http://localhost:8080/actuator/loggers/{name}",
            "templated": true
        },
        "mappings": {
            "href": "http://localhost:8080/actuator/mappings",
            "templated": false
        },
        "metrics": {
            "href": "http://localhost:8080/actuator/metrics",
            "templated": false
        },
        "metrics-requiredMetricName": {
            "href": "http://localhost:8080/actuator/metrics/{requiredMetricN
            "templated": true
        },
        "scheduledtasks": {
            "href": "http://localhost:8080/actuator/scheduledtasks",
            "templated": false
        },
        "self": {
            "href": "http://localhost:8080/actuator",
            "templated": false
        },
        "threaddump": {
            "href": "http://localhost:8080/actuator/threaddump",
            "templated": false
        }
    }
}
```

The listing of Actuator endpoints provides a good idea of the scope of information captured and exposed for examination, but of particular usefulness to actors good and bad are the following:

/actuator/beans

All Spring beans created by the application

/actuator/conditions

Conditions met (or not) to create Spring beans; similar to the Conditions Evaluation Report discussed previously

/actuator/configprops

All `Environment` properties accessible by the application

/actuator/env

Myriad aspects of the environment in which the application is operating; especially useful to see where each individual `configprop` value originates

/actuator/health

Health info (basic or expanded, depending on settings)

/actuator/heapdump

Initiates heap dump for troubleshooting and/or analysis

/actuator/loggers

Logging levels for every component

/actuator/mappings

All endpoint mappings and supporting details

/actuator/metrics

Metrics currently being captured by the application

/actuator/threaddump

Initiates thread dump for troubleshooting and/or analysis

These, and all of the remaining preconfigured Actuator endpoints, are handy when needed and easy to access for examination. Continuing to focus on the application's environment, even among these endpoints there are firsts among peers.

Getting Actuator to Open Up

As mentioned, Actuator's default security posture intentionally exposes only very limited *health* and *info* responses. In fact, the */actuator/health* endpoint provides a rather utilitarian "UP" or "DOWN" application status out of the box.

With most applications, however, there are dependencies for which Actuator tracks health information; it simply doesn't expose that additional information unless authorized to do so. To show expanded health information for preconfigured dependencies, I add the following property to *application.properties*:

```
management.endpoint.health.show-details=always
```

NOTE

There are three possible values for the health indicator's `show-details` property: `never` (default), `when_authorized`, and `always`. For this example, I choose `always` simply to demonstrate the possible, but for every application put into production, the correct choices would be either `never` or `when_authorized` in order to limit visibility to the application's expanded health information.

Restarting the application results in the addition of health information for the application's primary components to the overall application health summary when accessing the */actuator/health* endpoint, per Figure 5-17.

Figure 5-17. Expanded health information

Becoming More Environmentally Aware Using Actuator

One malady that often afflicts developers—present company included—is an assumption of complete knowledge of current application environment/state when behavior doesn't match expectations. This isn't entirely unexpected, especially if one wrote the anomalous code oneself. A relatively quick and invaluable first step is to *check all assumptions*. Do you *know* what that value is? Or are you just really certain you know?

Have you checked?

Especially in code in which outcomes are driven by inputs, this should be a required starting point. Actuator helps make this painless. Querying the application's */actuator/env* endpoint returns all environmental information; following is the portion of that result that shows only the properties set in the application to this point:

```
{
    "name": "Config resource 'classpath:/application.properties' via locatio
     'optional:classpath:/'",
    "properties": {
        "droid.description": {
            "origin": "class path resource [application.properties] - 5:19",
            "value": "Small, rolling android. Probably doesn't drink coffee.
        },
        "droid.id": {
            "origin": "class path resource [application.properties] - 4:10",
            "value": "BB-8"
        },
        "greeting.coffee": {
            "origin": "class path resource [application.properties] - 2:17",
            "value": "Dakota is drinking Cafe Cereza"
        },
        "greeting.name": {
            "origin": "class path resource [application.properties] - 1:15",
            "value": "Dakota"
        },
        "management.endpoint.health.show-details": {
            "origin": "class path resource [application.properties] - 8:41",
            "value": "always"
        },
        "management.endpoints.web.exposure.include": {
            "origin": "class path resource [application.properties] - 7:43",
            "value": "*"
        }
    }
}
```

Actuator shows not only the current value for each defined property but also its source, down to the line and column number where each value is defined. But what happens if one or more of those values is overridden by another source, e.g., an external environment variable or command line argument when executing the application?

To demonstrate a typical production-bound application scenario, I run `mvn clean package` from the application's directory at the command line, then execute the app with the following command:

```
java -jar target/sbur-rest-demo-0.0.1-SNAPSHOT.jar --greeting.name=Sertanejo
```

Querying */actuator/env* once more, you can see that there is a new section for command line arguments with a single entry for `greeting.name`:

```
{
    "name": "commandLineArgs",
    "properties": {
        "greeting.name": {
            "value": "Sertanejo"
        }
    }
}
```

Following the order of precedence for `Environment` inputs that was referenced earlier, command line arguments should override the value set from within *application.properties*. Querying the */greeting* endpoint returns “Sertanejo” as expected; querying */greeting/coffee* likewise results in the overridden value being incorporated into that response via the SpEL expression as well: `Sertanejo is drinking Cafe Cereza`.

Trying to get to the bottom of errant, data-driven behavior just got a lot simpler thanks to Spring Boot Actuator.

Turning Up the Volume on Logging with Actuator

Like many other choices in developing and deploying software, choosing logging levels for production applications involves tradeoffs. Opting for more logging results in more system-level work and storage consumption and the capture of both more relevant and irrelevant data. This in turn can make it considerably more difficult to find an elusive issue.

As part of its mission of providing Boot’s production-ready features, Actuator addresses this as well, allowing developers to set a typical logging level like “INFO” for most or all components and change that level temporarily when a critical issue surfaces…all in live, production Spring Boot applications. Actuator facilitates the setting and resetting of logging levels with a simple `POST` to the applicable endpoint. For example, Figure 5-18 shows the default logging level for `org.springframework.data.web`.

Figure 5-18. Default logging level for `org.springframework.data.web`

Of particular interest is that since a logging level wasn’t configured for this component, an effective level of “INFO” is used. Again, Spring Boot provides a sensible default when specifics aren’t provided.

If I’m notified of an issue with a running app and would like to increase logging to help diagnose and resolve it, all that is necessary to do so for a particular component is to `POST` a new JSON-formatted value for `configuredLevel` to its */actuator/loggers* endpoint, as shown here:

```
echo '{"configuredLevel": "TRACE"}'
  | http :8080/actuator/loggers/org.springframework.data.web
```

Requerying the logging level now returns confirmation that the logger for `org.springframework.data.web` is now set to “TRACE” and will provide intensive diagnostic logging for the application, as shown in Figure 5-19.

WARNING

"TRACE" can be essential in pinning down an elusive issue, but it is a rather heavyweight level of logging, capturing even finer-grained information than "DEBUG"—use in production applications can provide essential information, but be mindful of the impact.

Figure 5-19. New "TRACE" Logging Level for `org.springframework.data.web`

CODE CHECKOUT CHECKUP

For complete chapter code, please check out branch *chapter5end* from the code repository.

Summary

It's critical for a developer to have useful tools to establish, identify, and isolate behaviors manifested within production applications. As applications become more dynamic and distributed, there is often a need to do the following:

- Configure and reconfigure applications dynamically
- Determine/confirm current settings and their origins
- Inspect and monitor application environment and health indicators
- Temporarily adjust logging levels of live apps to identify root causes

This chapter demonstrated how to use Spring Boot's built-in configuration capabilities, its Autoconfiguration Report, and Spring Boot Actuator to create, identify, and modify application environment settings flexibly and dynamically.

In the next chapter, I dive deep into data: how to define its storage and retrieval using various industry standards and leading database engines and the Spring Data projects and facilities that enable their use in the most streamlined and powerful ways possible.

[1] Order of precedence for Spring Boot PropertySources.