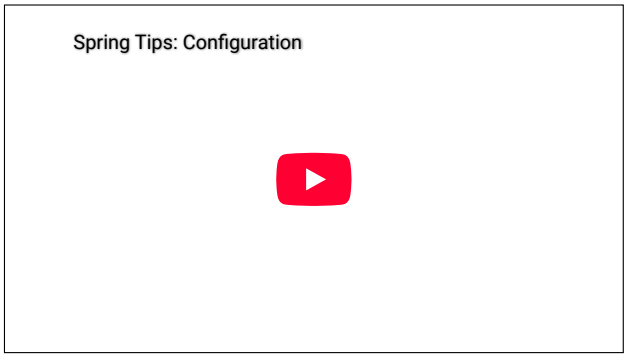
# Spring Tips: Configuration

**ENGINEERING | JOSH LONG | APRIL 23, 2020 | 2 COMMENTS**




speaker: Josh Long (@starbuxman)

Hi, Spring fans! Welcome to another installment of Spring tips! in this installment, we're going to look at something that's rather foundational, and something that I wish I'd addressed earlier: configuration. And no, I don't mean functional configuration or java configuration or anything like that, I'm talking about the string values that inform how your code executes. the stuff that you put in application.properites. *that* configuration.

All configuration in Spring emanates from the Spring `Environment` abstraction. The `Environment` is sort of like a dictionary - a map with keys and values. `Environment` is just an interface through which we can ask questions about, you know, the `Environment`. The abstraction lives in Spring Framework and was introduced in Spring 3, more than a decade ago. up until that point, there was a focused mechanism to allow integration of configuration called property placeholder resolution. This environment mechanism and the constellation of classes around that interface more than supersede that old support. if you find a blog still using those types, may I suggest you move on to newer and greener pastures? :)

Let's get started. Go to the Spring Initializr and generate a new project and make sure to choose `Spring Cloud Vault`, `Lombok`, and `Spring Cloud Config Client`. I named my project `configuration`. Go ahead and click `Generate` the application. Open the project in your favorite IDE. If you want to follow along, be sure to disable the Spring Cloud Vault and Spring Clod Config Lcieny dependencies. We don't need them right now.

The first step for most Spring Boot developers is to use application.properties. The Spring Initializr even puts an empty application.properties in the `src/main/resources/application.properties.` folder when you generate a new project there! Super convenient. You *do* create your projects on the Spring Initializr, don't ya'? You could use appication.properties or application.yml. I don't particularly love `.yml` files, but you can use it if that's more your taste.

Spring Boot automatically loads the `application.properties` whenever it starts up. You can dereference values from the property file in your java code through the environment. Put a property in the `application.properties` file, like this.

```
message-from-application-properties=Hello from application.properties
```

COPY

Now, let's edit the code to read in that value.

```
package com.example.configuration;

import lombok.extern.log4j.Log4j2;
import org.springframework.boot.ApplicationRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.core.env.Environment;

@Log4j2
@SpringBootApplication
public class ConfigurationApplication {

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

    @Bean
    ApplicationRunner applicationRunner(Environment environment) {
        return args -> {
            log.info("message from application.properties " + environment.getProperty("message-from-application
        };
    }
}
```

COPY

Run this, and you'll see the value form the configuration property file in the output of the log. If you want to change which file SPring Boot rads by default, you can do that too. It's a chicken and egg problem, though - you need to specify a property that

Spring Boot will use to figure out where to load all the properties. So you need to specify this outside of the application.properties file. You can use a program argument or an environment variable to fill the `spring.config.name` property.

```
export SPRING_CONFIG_NAME=foo
```

Re-run the application now with that environment variable in scope, and it'll fail because it'll try to load `foo.properties`, not `application.properties`.

Incidentally, you could also run the application with the configuration that lives *outside* the application, adjacent to the jar, like this. If you run the application like this, the values in the external `application.properties` will override the values inside the `.jar`.

```
.
├── application.properties
└── configuration-0.0.1-SNAPSHOT.jar

0 directories, 2 files
```

Spring Boot is aware of Spring profiles, as well. Profiles are a mechanism that lets you tag objects and property files so that they can be selectively activated or deactivated at runtime. This is great if you want to have an environment-specific configuration. You can tag a Spring bean or a configuration file as belonging to a particular profile, and Spring will automatically load it for you when that profile is activated.

Profile names are, basically, arbitrary. Some profiles are magic - that Spring honors in a particular way. The most interesting of these is `default`, which is activated when no other profile is active. But generally, the names are up to you. I find it very useful to map my profiles to different environments: `dev`, `qa`, `staging`, `prod`, etc.

Let's say that there's a profile called `dev`. Spring Boot will automatically load `application-dev.properties`. It'll load that in addition to application.properties. If there are any conflicts between values in the two files, then the more specific file - the one with the profile - wins. You could have a default value that applies absent a particular profile, and then provide specifics in the config for a profile.

You can activate a given profile in several different ways, but the easiest is just to specify it on the command line. Or you could turn it on in your IDE's run configurations dialog box. IntelliJ and Spring Tool Suite both provide a place to specify the profile to sue when running the application. You can also set an env var, `SPRING_PROFILES_ACTIVE`, or specify an argument on the command line `--spring.profiles.active`. Either one accepts a comma-delimited list of profiles - you can activate more than one profile at a time.

Le'ts try that out. Create a file called `application-dev.properties`. Put the following value in it.

```
message-from-application-properties=Hello from dev application.properties
```

This property has the same key like the one in `application.properties`. The java code here is identical to what we had before. Just be sure to specify the profile before you start the Spring application. You can use the environment variable, properties, etc. You can even define it programmatically when building the `SpringApplication` in the `main()` method.

```
package com.example.configuration.profiles;

import lombok.extern.log4j.Log4j2;
import org.springframework.boot.ApplicationRunner;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.builder.SpringApplicationBuilder;
import org.springframework.context.annotation.Bean;
import org.springframework.core.env.Environment;

@Log4j2
@SpringBootApplication
public class ConfigurationApplication {

    public static void main(String[] args) {
        // this works
        // export SPRING_PROFILES_ACTIVE=dev
        // System.setProperty("spring.profiles.active", "dev"); // so does this
        new SpringApplicationBuilder()
            .profiles("dev") // and so does this
            .sources(ConfigurationApplication.class)
            .run(args);
    }

    @Bean
    ApplicationRunner applicationRunner(Environment environment) {
        return args -> {
            log.info("message from application.properties " + environment.getProperty("message-from-application
        };
    }
}
```

Run the application, and you'll see the specialized message reflected in the output.

So far, we've been using the ENvironment to inject the configuration. You can also use `@Value` annotation to inject the value as a parameter. You probably already know that. But did you know that you can also specify default values to be returned if there are no other values that match? There are a lot of reasons why you might want to do this. You could use it to provide fallback values and make it more transparent when somebody fat fingers the spelling of a property. It is also useful because you are given a value that might be useful if somebody doesn't know that they need to activate a profile or something,.

```
package com.example.configuration.value;

import lombok.extern.log4j.Log4j2;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.ApplicationRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;

@Log4j2
@SpringBootApplication
public class ConfigurationApplication {

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

    @Bean
    ApplicationRunner applicationRunner(
        @Value("${message-from-application-properties:OOPS!}") String valueDoesExist,
        @Value("${mesage-from-application-properties:OOPS!}") String valueDoesNotExist) {
        return args -> {
            log.info("message from application.properties " + valueDoesExist);
            log.info("missing message from application.properties " + valueDoesNotExist);
        };
    }
}
```

Convenient, eh? Also, note that the default String that you provide can, in turn, interpolate some other property. So you could do something like this, assuming a key like `default-error-message` does exist somewhere in your application configuration:

```
${message-from-application-properties:${default-error-message:YIKES!}}
```

That will evaluate the first property if it exists, then the second and then the String `YIKES!`, finally.

Earlier, we looked at how to specify a profile using an environment variable or program argument. This mechanism - configuring Spring Boot with environment variables or program arguments - is a general-purpose. You can use it for any arbitrary key, and Spring Boot will normalize the configuration for you. Any key that you would out in application.properties can be specified externally in this way. Let's see some examples. Let's suppose you want to specify the URL for a data source connection. You *could* hardcode that value in the application.properties, but that's not very secure. It might be much better to create instead an environment variable that only exists in production. That way, the developers don't have access to the keys to the production database and so on.

Let's try it out. Heres the java code fo the example.

```
package com.example.configuration.envvars;

import lombok.extern.log4j.Log4j2;
import org.springframework.boot.ApplicationRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.core.env.Environment;

@Log4j2
@SpringBootApplication
public class ConfigurationApplication {

    public static void main(String[] args) {
        // simulate program arguments
        String[] actualArgs = new String[]{"spring.datasource.url=jdbc:postgres://localhost/some-prod-db"};
        SpringApplication.run(ConfigurationApplication.class, actualArgs);
    }

    @Bean
    ApplicationRunner applicationRunner(Environment environment) {
        return args -> {
            log.info("our database URL connection will be " + environment.getProperty("spring.datasource.url"))
        };
    }
}
```

Before you run it, be sure to either export an environment variable in the shell that you use to run your application or to specify a program argument. I simulate the latter - the program arguments - by intercepting the `public static void main(String [] args)` that we pass into the Spring Boot application here. You can also specify an env variable like this:

```
export SPRING_DATASOURCE_URL=some-arbitrary-value
mvn -DskipTests=true spring-boot:run
```

Run the program multiple times, trying out the different approaches, and you will see the values in the output. There's no autoconfiguration in the application that will connect to a database, so we're using this property as an example. The URL doesn't have to be a valid URL (at least not until you add Spring's JDBC support and a JDBC driver to the classpath).

Spring Boot is *very* flexible in its sourcing of the values. It doesn't care if you do `SPRING_DATASOURCE_URL`, `spring.datasource.url`, etc. Spring Boot calls this *relaxed binding*. It allows you to do things in a way that's most natural for different environments, while still working for Spring Boot.

This idea - of externalizing configuration for an application from the environment - is not new. It's well understood and described in the 12-factor manifesto. The 12-factor manifesto says that environment-specific config should live in that environment, not in the code itself. This is because we want one build for all the environments. Things that change should be external. So far, we've seen that Spring Boot can pull in configuration from the command line arguments (program arguments), and environment variables. It can also read configuration coming from JOpt. It can come even from a JNDI context if you happen to be running in an application server with one of those around!

Spring Boots's ability o pull in any environment variable is beneficial here. It's also more secure than using program arguments because the program arguments will show up in the output of operating system tools. Environment variables are a better fit.

So far, we've seent hat Spring Boot can pull in configuration from a lot of different places. It knows about profiles, it knows about `.yml.` and `.properties`. It's pretty flexible! But what if it doesn't know how to do what you want it to do? You can easily reach its new tricks using a custom `PropertySource<T>`. You might want to do something like this if you wish to, for example, to integrate your application with the configuration you're storing in an external database or a directory or some other things about which Spring Boot doesn't automatically know.

COPY

```
package com.example.configuration.propertysource;

import lombok.extern.log4j.Log4j2;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.ApplicationRunner;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.builder.SpringApplicationBuilder;
import org.springframework.context.annotation.Bean;
import org.springframework.core.env.PropertySource;

@Log4j2
@SpringBootApplication
public class ConfigurationApplication {

    public static void main(String[] args) {
        new SpringApplicationBuilder()
            .sources(ConfigurationApplication.class)
            .initializers(context -> context
                .getEnvironment()
                .getPropertySources()
                .addLast(new BootifulPropertySource())
            )
            .run(args);
    }

    @Bean
    ApplicationRunner applicationRunner(@Value("${bootiful-message}") String bootifulMessage) {
        return args -> {
            log.info("message from custom PropertySource: " + bootifulMessage);
        };
    }
}

class BootifulPropertySource extends PropertySource<String> {

    BootifulPropertySource() {
        super("bootiful");
    }

    @Override
    public Object getProperty(String name) {

        if (name.equalsIgnoreCase("bootiful-message")) {
            return "Hello from " + BootifulPropertySource.class.getSimpleName() + "!";
        }

        return null;
    }
}
```

The example above is the safest way to register a `PropertySource` early enough on that everything that needs it will be able to find it. You can also do it at runtime when Spring has started wiring objects together, and you have access to configured objects, but I wouldn't be sure that this will work in every situation. Heres how that might look.

COPY

```
package com.example.configuration.propertysource;

import lombok.extern.log4j.Log4j2;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.ApplicationRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.core.env.ConfigurableEnvironment;
import org.springframework.core.env.PropertySource;

@Log4j2
@SpringBootApplication
public class ConfigurationApplication {

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

```
        SpringApplication.run(ConfigurationApplication.class, args);
    }

    @Bean
    ApplicationRunner applicationRunner(@Value("${bootiful-message}") String bootifulMessage) {
        return args -> {
            log.info("message from custom PropertySource: " + bootifulMessage);
        };
    }

    @Autowired
    void contributeToTheEnvironment(ConfigurableEnvironment environment) {
        environment.getPropertySources().addLast(new BootifulPropertySource());
    }
}

class BootifulPropertySource extends PropertySource<String> {

    BootifulPropertySource() {
        super("bootiful");
    }

    @Override
    public Object getProperty(String name) {

        if (name.equalsIgnoreCase("bootiful-message")) {
            return "Hello from " + BootifulPropertySource.class.getSimpleName() + "!";
        }

        return null;
    }
}
```

Thus far, we've looked almost entirely at how to source property values from elsewhere. Still, we haven't talked about what becomes of the Strings once they're in our working memory and available for use in the application. Most of the time, they're just strings, and we can use them as-is. Sometimes, however, it's useful to turn them into other types of values - ints, Dates, doubles, etc. this work - turning strings into things - could be the topic of a whole other Spring Tips video and perhaps one ill do soon. Suffice it to say that there are a lot of interrelated pieces there - the `ConversionService` , `Converter<T>` s, Spring Boot's `Binder` s, and so much more. For common cases, this will just work. You can, for example, specify a property `server.port = 8080` and then inject it into your application as an int:

```
@Value("${server.port}") int port
```
COPY

It might be helpful to have these values bound to an object automatically. This is precisely what Spring Boots `ConfigutationProperties` do for you. Let's see this in action.

Ley's say that ou ave an application.properties file with the following property:

```
bootiful.message = Hello from a @ConfiguratinoProperties
```
COPY

Then you can run the application and see that the configuration value has been bound to the object for us:

```
package com.example.configuration.cp;

import lombok.Data;
import lombok.RequiredArgsConstructor;
import lombok.extern.log4j.Log4j2;
import org.springframework.boot.ApplicationRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.boot.context.properties.ConstructorBinding;
import org.springframework.boot.context.properties.EnableConfigurationProperties;
import org.springframework.context.annotation.Bean;

@Log4j2
@SpringBootApplication
@EnableConfigurationProperties(BootifulProperties.class)
public class ConfigurationApplication {

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

    @Bean
    ApplicationRunner applicationRunner(BootifulProperties bootifulProperties) {
        return args -> {
            log.info("message from @ConfigurationProperties " + bootifulProperties.getMessage());
        };
    }

}

@Data
@RequiredArgsConstructor
@ConstructorBinding
@ConfigurationProperties("bootiful")
class BootifulProperties {
    private final String message;
}
```
COPY

The `@Data` and `@RequiredArgsConstructor` annotations on the `BootifulProperties` object come from Lombok. `@Data` synthesizes getters for final fields and getters and setters for non-final fields. `@RequiredArgsConstructor` synthesizes a constructor for all the final fields int he class. The result is an object that's immutable once constructed through constructor initialization. Spring boot's ConfigurationProperties mechanism doesn't know about immutable objects by default; you need to use the `@ConstructorBinding` annotation, a reasonably new addition to Spring Boot, to make it do the right thing here. This is even more useful in other programming languages like Kotlin ( `data class ...` ) and Scala ( `case class ...` ), which have syntax sugar for creating immutable objects.

We've seen that Spring can load configuration adjacent to the application `.jar` , and that it can load the configuration from environment variables and program arguments. It's not hard o get information into a Spring Boot application, but its sort of piecemeal. It's hard to version control environment variables or to secure program arguments.

To solve some of these problems, the Spring Cloud team built the spring CLou COnfigu Server. The Spring Cloud Config Server is an HTTP API that fronts a backend storage engine. The storage s pluggable, with the most common being a Git repository, though there is support for others as well. These include SUbversion, a local file system, and even MongDB.

We're going to set up a new Spring Cloud Config Server. Go to the Spring Initializr and choose `Config Server` and then click `Generate` . Open it in your favorite IDE.

We're going to need to do two things to make it work: first, we must use an annotation and then provide a configuration value to point it to the Git repository with our configuration file. Here are the `application.properties` .

```
spring.cloud.config.server.git.uri=https://github.com/joshlong/greetings-config-repository.git
server.port=8888
```

And here's what your main class should look like.

```
package com.example.configserver;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.config.server.EnableConfigServer;

@EnableConfigServer
@SpringBootApplication
public class ConfigServerApplication {

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

Run the application - `mvn spring-boot:run` or just run the application in your favorite IDe. It's now available. It'll act as a proxy to the Git configuration in the Github repository. Other clients can then use the Spring Cloud Config Client to pull their configuration in from the Spring Cloud Config Server, which will, in turn, pull it in from the Gi repository. Note: I'm making this as insecure as possible for ease of the demo, ut you can and should secure both links in the chain - from the config client to the config server, and from the config server to the git repository. Spring Cloud Config Server, the Spring Cloud Config Client, and Github all work well together, and securely.

Now, go back to the build for our configuration app and makes rue to uncomment the Spring Cloud Config Client dependency. To start the Spring Cloud Config Server, it'll need to have some - you guessed it! - configuration. A classic chicken and egg problem. This configuration needs to be evaluated earlier, before the rest of the configuration. You can put this configuration in a file called `bootstrap.properties` .

You'll need to identify your application to give it a name so that when it connects to the Spring Cloud Config Server, it will know which configuration to provide us. The name we specify here will be matched to a property file in the Git repository. Here's what you should put in the file.

```
spring.cloud.config.uri=http://localhost:8888
spring.application.name=bootiful
```

now we can read any value we want in the git repository in the `bootiful.properties` file whose contents are:

```
message-from-config-server = Hello, Spring Cloud Config Server
```

We can pull that configuration file in like this:

```
package com.example.configuration.configclient;

import lombok.extern.log4j.Log4j2;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.ApplicationRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;

@Log4j2
@SpringBootApplication
public class ConfigurationApplication {

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

    @Bean
```

```
    ApplicationRunner applicationRunner(@Value("${message-from-config-server}") String configServer) {
        return args -> {
            log.info("message from the Spring Cloud Config Server: " + configServer);
        };
    }
}
```

You should see the value in the output. Not bad! The Spring Cloud Config Server does a lot of cool stuff for us. It can encrypt values for us. It can help version out properties. One of my favorite things is that you can change the configuration independent of the change to the codebase. You can use that in conjunction with the Spring Cloud `@RefreshScope` to dynamically reconfigure an application after it started running. (I should do a video on the refresh scope and its many myriad uses...) The Spring Cloud Config Server is among the most popular Spring Cloud modules for a reason - it can be used with monoliths and microservices alike.

The Spring Cloud Config Server can encrypt values in the property files if you configure it appropriately. It works. A lot of folks also use Hashicorp's excellent Vault product, which is a much more fully-featured offering for security. Vault can secure, store, and tightly control access to tokens, passwords, certificates, encryption keys for protecting secrets, and other sensitive data using a UI, CLI, or HTTP API. You can also use this easily as a property source using the Spring Cloud Vault project. Uncomment the Sring Cloud Vault dependency from the build, and let us look at setting up Hashicorp Vault.

Download the latest version and then run the following commands. I'm assuming a Linux or Unix-like environment. It should be fairly straightforward to translate to Windows, though. I won't try to explain everything about Vault; I'd refer you to the excellent Getting Statted guides for Hashicorp Vault, instead. Here's the least-secure, but quickest, the way I know to get this all set up and working. First, run the Vault server. I'm providing a root token here, but you would typically use the token provided by Vault on startup.

```
export VAULT_ADDR="https://localhost:8200"
export VAULT_SKIP_VERIFY=true
export VAULT_TOKEN=00000000-0000-0000-0000-000000000000
vault server --dev --dev-root-token-id="00000000-0000-0000-0000-000000000000"
```

Once that's up, in another shell, install some values into the Vault server, like this.

```
export VAULT_ADDR="http://localhost:8200"
export VAULT_SKIP_VERIFY=true
export VAULT_TOKEN=00000000-0000-0000-0000-000000000000
vault kv put secret/bootiful message-from-vault-server="Hello Spring Cloud Vault"
```

That puts the key `message-from-vault-server` with a value `Hello Spring Cloud Vault` into the Vault service. Now, let's change our application to connect to that Vault instance to read the secure values. We'll need a bootstrap.properties, just as with the Spring Cloud Config Client.

```
spring.application.name=bootiful
spring.cloud.vault.token=${VAULT_TOKEN}
spring.cloud.vault.scheme=http
```

Then, you can use the property just like any other configuration values.

```
package com.example.configuration.vault;

import lombok.extern.log4j.Log4j2;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.ApplicationRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;

@Log4j2
@SpringBootApplication
public class ConfigurationApplication {

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

    @Bean
    ApplicationRunner applicationRunner(@Value("${message-from-vault-server:}") String valueFromVaultServer) {
        return args -> {
            log.info("message from the Spring Cloud Vault Server : " + valueFromVaultServer);
        };
    }
}
```

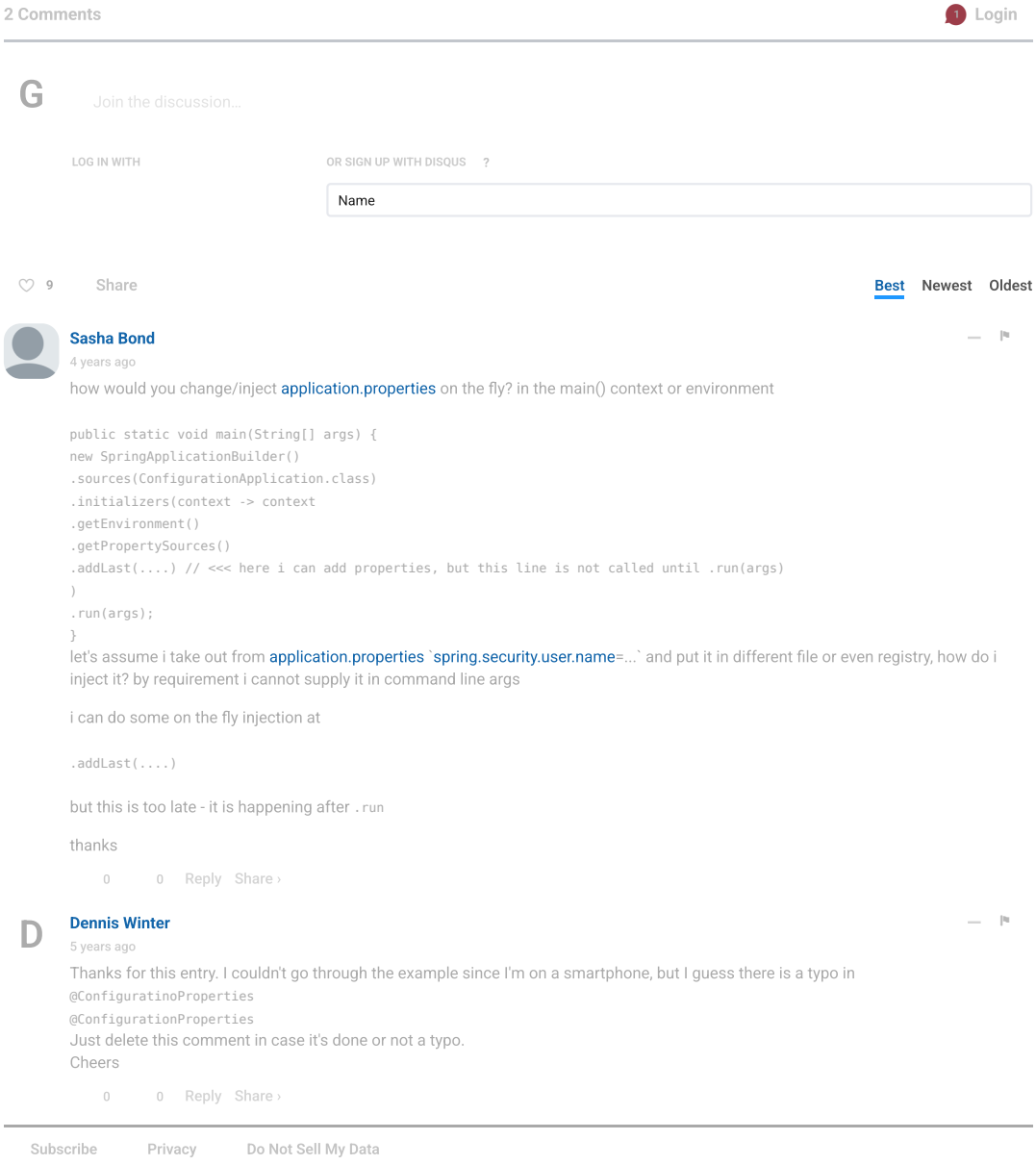
Now, before you run this, make sure also to have the same three environment variables we used in the tow interactions with the `vault` CLI configured: `VAULT_TOKEN`, `VAULT_SKIP_VERIFY`, and `VAULT_ADDR`. Then run it, and you should see reflected on the console the value that you write to Hashicorp Vault.

## Next Steps

Hopefully, you've learned something about the colorful and compelling world of configuration in SPring. With this information under your belt, you're now better prepared to use the other projects that support property resolution. Armed with this knowledge of how this works, you're ready to integrate configuration from different Spring integrations, of which there are a *ton*! You might use the Spring Cloud Netflix' Archaius integration, or the Configmaps integration with Spring Cloud Kubernetes, or the Spring Cloud GCP's Google Runtime Configuration API integration, or Spring Cloud Azure's Microsoft Azure Key Vault integration

etc.

I've only mentioned a few offerings here, but it doesn't matter if the list is exhaustive, their use will be the same if the integration is correct: the cloud's the limit!



Batch