Everything You Never Wanted to Know About Spring Boot 3 AOT

Table of Contents

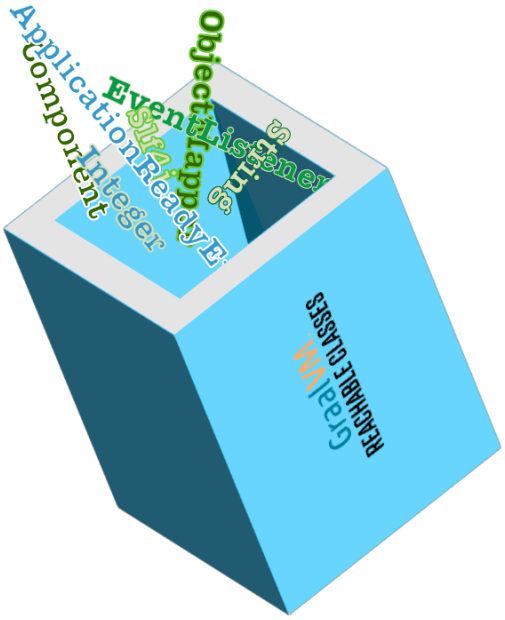
Hi, Spring fans! Spring Boot 3 is here, and it's amazing! If you want to know the broad strokes of Spring Framework 6 includes, check out this Spring Tips video for a quick rundown. In this text, however, I want to look at the details of the brand-new ahead-of-time (AOT) compilation engine in Spring Boot 3.

Spring Boot 3's AOT engine introduces a new component model designed to provide extra information about a Spring application to support optimizing the application both on the JRE and in a GraalVM native image context. For the 80% case, you shouldn't worry about a thing. But,

sometimes, you may need to intervene to get the best results. This text aims to introduce the engine, its happy-path usage, and the corner-case use.

Why would you need to intervene? What could go wrong? GraalVM native image compilation, that's what. GraalVM's native image compilation produces lightning-quick and super memory-efficient binaries. The caveat is that to achieve this magic trick, GraalVM's native image compiler does a compile time scan to determine what types are used, and what types those types use, and what types those types use, etc., all the way down the line until it *thinks* it knows what you're going to need to completely run the program. Then, it throws *everything* else away! This scan is sometimes called the reachability analysis, since only things that are transitively *reachable* from the `main()` method of the program are preserved. The problem is that this compile time scan can't deterministically predict every runtime use of types because Java has so many dynamic behaviors, like reflection, proxies, JNI, serialization, etc. Somebody or something has to tell GraalVM about these particular usecases.

```
record Customer(Integer id, String name) {}

@Slf4j
@Component
class CustomerService {

  private final ObjectMapper objectMapper ;

  // ...

  @EventListener(ApplicationReadyEvent.class)
  public void ready() throws Exception {
    java.util.List.of(new Customer(...), new Customer(...))

  // ...


  }
}
```



The GraalVM native image compiler needs this configuration specified either as command line switches or as `.json` configuration files present on the classpath - `META-INF/native-image/{reflect,jni,resource,proxy,etc.}-config.json`. Ideally, each library will ship its configuration. For example, Apache Tomcat, Netty, etc., need to support their use by furnishing this configuration. There's also an Oracle led effort called the GraalVM reachability repository that is a central repository of configuration provided for common libraries that don't ship their own `.json` configuration files (yet). Spring Boot 3 is the first framework to take advantage of this reachability repository out of the box.

But what about your code, written with Spring Framework? Spring (and Java, writ large) do many dynamic things with user-provided types that they need to account for with configuration, or the GraalVM compilation will fail.

In this release, we've introduced a whole new engine - the Spring AOT engine - to make it possible to take your existing Spring Boot applications, upgrade them to Spring Boot 3, and then enjoy the incredible benefits of GraalVM native image compilation. That's the dream, anyway, and we're essentially there already. This text will look at the new AOT engine, GraalVM, and the increasingly

rarer corner cases requiring some intervention.

Introduction

I find it helpful to understand where we were to appreciate where we are now. Traditional Spring applications have *phases* that they go through when they run. Here's a simplified understanding of what's happening in your standard Spring application:

Ingest

The Spring application starts up and reads in all the configuration sources. Remember: Spring Framework is a dependency management framework. It needs to know how your various objects are constructed, their lifecycles (constructors, `InitializingBean#afterPropertiesSet`,`@PostConstruct`, `DisposableBean#destroy`, `@PreDestroy`), etc. So it reads the various configuration files from all the `@Configuration`-annotated classes in your application through component scanning, where Spring discovers classes annotated with `@Component`. This component scanning also discovers classes annotated with annotations that have the `@Component` annotation on them, like `@Service`, `@Repository`, @Controller`, and `@RestController`. It also reads in configuration from Spring's classic XML configuration format. (I don't use the XML format and haven't seen it in the wild in over half a decade, but *still*…).

`BeanFactory` creation of `BeanDefinition` instances

At this point, Spring turns all the various inputs into a metamodel representation of your objects, a `BeanFactory` full of `BeanDefinition` instances. These `BeanDefinition` instances describe the objects, wiring, and more. They represent the constructors, annotations, properties to be injected, setters, etc. The `BeanDefinition` instances describe everything required to describe an object and get to a valid state so that it may be given to other objects and generally start doing work.

Notably, at this phase, there are no live-fire beans. Nothing is opening ports and sockets, doing disk IO, etc. This phase in the `BeanFactory` does not involve any business logic.

Bean Creation

Spring will take all the `BeanDefinition` instances and create actual, live-fire beans. Spring will call constructors, invoke lifecycle methods, etc. After this phase, you'll have an application ready to serve production traffic. Being production-worthy is *good* ™. We like production, don't we?

Spring Boot 3 introduces a new phase, at compile time, to support AOT.

Enter GraalVM

GraalVM is a drop-in replacement for OpenJDK. It *is* OpenJDK, largely. It has a few extra utilities: notably, a polyglot VM, a native image compiler, and a replacement HotSpot replacement for the JIT (just-in-time) compiler. We could spend all day on these extra features, but let's focus on the `native-image` compiler. Henceforth, when I say "GraalVM," I refer to the `native-image` compiler.

You Gotta Start Somewhere

To start, as always, we'll go to the Spring Initialzr (start.spring.io). Choose Spring Boot 3.0 or later from the Spring Initializr. I'm using Spring Boot 3, Apache Maven, and Java 17. Java 17 is the new baseline for Spring Framework 6 and Spring Boot 3. I'm also using GraalVM 22.3. If you're using the fabulous SDKMAN utility, then you can do the following to get the same version of GraalVM that I have on my computer as of this writing (late 2022):

```
sdk install java 22.3.r17-grl
```

I've also added a few dependencies. We won't touch on these dependencies in any great length, but they'll help us to demonstrate a few concepts, so I've added them.

- Web (`org.springframework.boot` : `spring-boot-starter-web`)
- Actuator (`org.springframework.boot` : `spring-boot-starter-actuator`)
- Spring Data JDBC (`org.springframework.boot` : `spring-boot-starter-data-jdbc`)
- H2 (`com.h2database` : `h2`)

And, of course, I've got a standard Spring Boot entry class.

```
package com.example.aot;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class AotApplication {

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

}
```

I won't add anything to this class. Instead, as we introduce new concepts, I'll create new `@Configuration`-annotated classes in sub-packages. We won't revisit this class.

In practice, let's look at our first example of the new AOT engine. This trivial and fairly typical application works with a database and surfaces an HTTP endpoint like any other trivial Spring Boot application. But, of course, you have seen this before.

```
package com.example.aot.basics;

import org.springframework.boot.context.event.ApplicationReadyEvent;
import org.springframework.context.ApplicationListener;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.annotation.Id;
import org.springframework.data.repository.CrudRepository;

import java.util.stream.Stream;

@Configuration
class BasicsConfiguration {

    @Bean
    ApplicationListener<ApplicationReadyEvent>
basicsApplicationListener(CustomerRepository repository) {
        return event -> repository //
                .saveAll(Stream.of("A", "B", "C").map(name -> new Customer(null,
name)).toList()) //
                .forEach(System.out::println);
    }

}

①
record Customer(@Id Integer id, String name) {
}

②
interface CustomerRepository extends CrudRepository<Customer, Integer> {

}
```

① look ma, no Lombok!

② this is using Spring Data JDBC, and you can too!

You'll need some SQL schema, so add `src/main/resources/schema.sql` for our SQL interactions; so let's have Spring Boot create it for us on startup:

```
create table customer
(
    id   serial primary key,
    name varchar(255) not null
);
```

Run the main class in your IDE or on the command line in the usual ways:

```
mvn -DskipTests spring-boot:run
```

You should see some output on the console. It works. Hurray. Moving on, let's turn it into a native image, thusly:

```
mvn -Pnative -DskipTests native:compile
```

That'll take a minute to finish, so now's a good time to pour a cup of coffee or tea. Maybe do a crossword puzzle. Reflect on your poor life choices to get to this point where your JVM applications take around a minute to compile. It's a bit dispiriting. But always remember: good things come to those who wait!

You'll find the compiled binary in the `target` directory, named `aot`. Run it, and you'll see it starts up in no time at all. Like a lightbulb! And, the best part is that it takes very little memory. Of course, there are different ways to measure memory, but looking at *resident set size* is informative. Here's a script I use to measure this stuff called `rss.sh`:

```
#!/usr/bin/env bash
PID=${1}
RSS=`ps -o rss ${PID} | tail -n1`
RSS=`bc <<< "scale=1; ${RSS}/1024"`
echo "${PID}: ${RSS}M"
```

It captures the RSS for a given process identifier, scales it to make it easier to parse, then prints it out. So, for example, you can find the process identifier (PID) for the Spring Boot application in the console towards the top of the application's output.

Take it and then pass it as the argument for `rss.sh`, like this:

```
./rss.sh <PID>
```

On my 2021 M1 MacBook Pro, I tend to get numbers just south of 100MB. Not bad! How much RAM is your current JVM application taking? I'd be pleasantly surprised if it were a gigabyte or less! Imagine being able to deploy the same application for 1/10th of the memory footprint!

The Spring component model

So, we know the basic stuff's working. Spring's got a rich, dynamic, multifaceted component model that can do amazing things. So, let's look at some examples demonstrating a few interesting aspects of the Spring programming model, old and new(-ish). I hope, but don't know, that a lot of this will rehash old stuff for many of you. So, feel free to skim this section and see if anything new in the headers strikes you as interesting. It's here because I wanted to be thorough and view the rich Spring component model through the prism of the new Spring AOT engine.

See How They Run

You can compile and run all the code in the usual way as a GraalVM native image:

```
mvn -DskipTests native:compile && ./target/aot
```

Run the application, and you should see all the output from the previous examples, as expected. The best part? It'll have started in no time and be far smaller than a binary and a process occupying RAM. You can use the `rss.sh` script we introduced to measure the process' RSS.

Event Listeners

Did you know that Spring has an event bus that you can use to publish and receive events in one component or another? Any component can fire an event (or more) and listen to and consume these events. There are two ways to consume events: with a bean of type `ApplicationListener<ApplicationEvent>`, or with the `@EventListener` annotation.

Here's a simple example.

```
package com.example.aot.events;

import org.springframework.boot.context.event.ApplicationReadyEvent;
import org.springframework.boot.web.context.WebServerInitializedEvent;
import org.springframework.context.ApplicationEvent;
import org.springframework.context.ApplicationListener;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.event.EventListener;

@Configuration
class EventsConfiguration {

    ①
    @Bean
    ApplicationListener<WebServerInitializedEvent>
webServerInitializedEventApplicationListener() {
        return event -> run("ApplicationListener<WebServerInitializedEvent>", event);
    }

    ②
    @EventListener
    public void eventListener(ApplicationReadyEvent event) {
        run("@EventListener", event);
    }

    private void run(String where, ApplicationEvent are) {
        System.out.println(where + " : " + are);
    }

}
```

① this is the first more traditional approach: a bean of type `ApplicationListener<T extends ApplicationEvent>`.

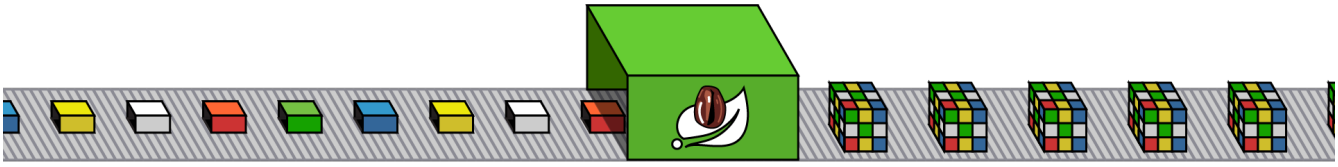
② this is the newer style, which frees your code of any explicit dependencies on the Spring framework

This example listens for two unrelated events. Although there is no significance to these events for our example, they demonstrate different ways of consuming Spring application events.

- `WebServerInitializedEvent` tells when the embedded web server has finished initializing.
- The `ApplicationReadyEvent` gets called as late as possible, right before the application handles traffic.

These are just a few of the events that Spring and Spring Boot emit as part of the lifecycle of an application. There are other events for all sorts of stuff, including Spring Security authentication, Actuator, Spring lifecycle, and more.

FactoryBeans: A Reusable Recipe for Object Creation



Sometimes objects require finessing and customization. Sometimes, creating an object becomes more complicated than just a simple constructor. Therefore, isolating this construction logic in a single place is helpful because it is reusable. There are at least two patterns that describe this sort of parameterized construction beside a constructor:

- the fluid builder pattern
- the factory pattern

There's nothing Spring can do to support the first pattern: that's very much up to each implementor how their types reflect the object creation patterns particular to their API. However, the Spring Framework `FactoryBean<T>` supports the second pattern. When you register a class of type `FactoryBean<T>` in the Spring context, it is the *product* (an instance of type `T`) of that `FactoryBean<T>`, not the `FactoryBean<T>` itself, that is registered in the application context and made available for injection. In other words, you can't inject a reference to `FactoryBean<Foo>`, only `Foo`.

Let's look at its use in a trivial example:

```
package com.example.aot.factorybeans;

import org.springframework.beans.factory.FactoryBean;
import org.springframework.boot.context.event.ApplicationReadyEvent;
import org.springframework.context.ApplicationListener;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
class FactoryBeansConfiguration {

    ①
    @Bean
    AnimalFactoryBean animalFactoryBean() {
        return new AnimalFactoryBean(true, false);
    }

    ②
    @Bean
    ApplicationListener<ApplicationReadyEvent> factoryBeanListener(Animal animal) {
        return event -> animal.speak();
    }
```

```
}

class AnimalFactoryBean implements FactoryBean<Animal> {

    private final boolean likesYarn, likesFrisbees;

    AnimalFactoryBean(boolean likesYarn, boolean likesFrisbees) {
        this.likesYarn = likesYarn;
        this.likesFrisbees = likesFrisbees;
    }

    @Override
    public Animal getObject() {
        return (this.likesYarn && !this.likesFrisbees) ? new Cat() : new Dog();
    }

    @Override
    public Class<?> getObjectType() {
        return Animal.class;
    }

}

interface Animal {

    void speak();

}

class Dog implements Animal {

    @Override
    public void speak() {
        System.out.println("woof");
    }

}

class Cat implements Animal {

    @Override
    public void speak() {
        System.out.println("meow");
    }

}
```

① the `AnimalFactoryBean` produces an object of type `Animal`. But which? It depends on the parameters fed into the `FactoryBean`.

② the client code injects the `Animal`, ignorant of the construction logic.

XML Configuration

Now, I know you didn't ask, but just in case you were wondering: Spring's new AOT engine works well enough with classic XML application configuration. We talked about XML configuration when we introduced the idea that Spring *ingests*, or reads in, configuration from various sources, one of which is XML files.

Let's look, ever so quickly, at an example. First, the XML configuration file:

```
<?xml version="1.0" encoding="UTF-8" ?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
        http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

    ①
    <bean class="com.example.aot.xml.MessageProducer" id ="messageProducer"/>

    ②
    <bean class="com.example.aot.xml.XmlLoggingApplicationListener" >
        <property name="producer" ref="messageProducer"/>
    </bean>


</beans>
```

① this `<bean/>` element defines a bean of type `MessageProducer`

② this `<bean/>` element defines a bean of type `XmlLoggingApplicationListener`, which in turn injects an instance of type `MessageProducer`

Now, let's look at the Java code:

```
package com.example.aot.xml;

import org.springframework.boot.context.event.ApplicationReadyEvent;
import org.springframework.context.ApplicationListener;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.ImportResource;
import org.springframework.lang.Nullable;

import java.util.function.Supplier;

①
@Configuration
@ImportResource("app.xml")
class XmlConfiguration {

}

②
class MessageProducer implements Supplier<String> {

    @Override
    public String get() {
        return "Hello, world!";
    }

}

③
class XmlLoggingApplicationListener implements
ApplicationListener<ApplicationReadyEvent> {

    @Nullable
    private MessageProducer producer;

    public void setProducer(@Nullable MessageProducer producer) {
        this.producer = producer;
    }

    @Override
    public void onApplicationEvent(ApplicationReadyEvent event) {
        var message = this.producer.get();
        System.out.println("the message is " + message);
    }

}
```

① the `XmlConfiguration` is just another `@Configuration`-annotated class as before, but this time with a very import directive: the `@ImportResource("app.xml")` tells Spring to load the beans defined in

the XML configuration file as beans in the `BeanFactory`

② The `MessageProducer` supplies our message

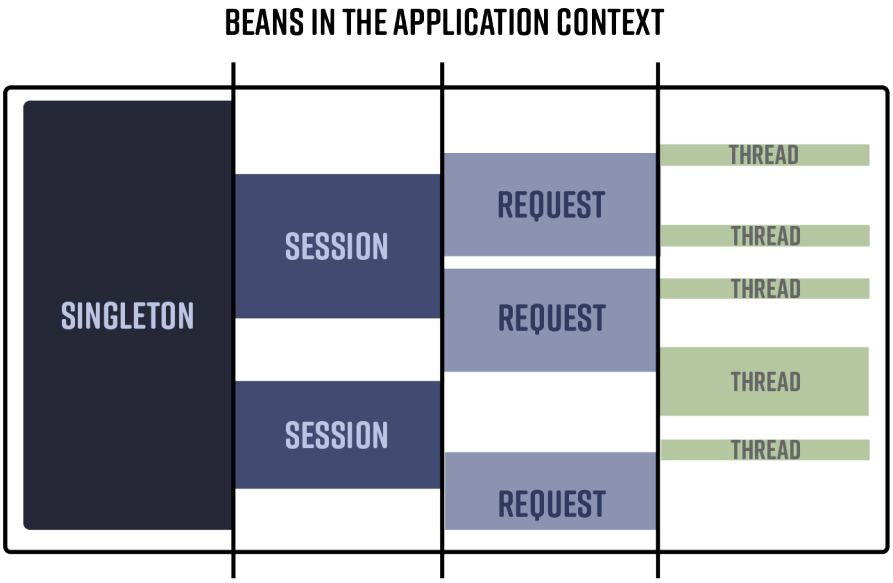
③ The `XmlLoggingApplicationListener` uses the `MessageProducer` to print out a message. Nothing particularly special.

Compile and run the application as a native image and you'll see the output: `the message is ⋯`, which will only work if Spring's able to correctly read and run the XML configuration file.

Scopes

Beans in Spring have a lifecycle governing a given object's lifetime. Unless you specify something precisely, the default scope is `singleton`. Here are some of the more commonly used scopes.

- `singleton` - Spring creates a bean when the application starts up and destroys it when it shuts down. A bean defined in this way is *global* - all clients of the bean will see the same state. So take care to handle concurrent access to this state in the same way you would any multithreaded access.
- `session` - a bean is created anew for each new HTTP Session. Each user with an HTTP session will have their instance of the bean. Changes won't be visible to other clients.
- `web` - each new HTTP request gets a new bean instance.
- `thread` - beans are unique to each thread. Sort of like a `ThreadLocal`.



This mechanism is pluggable, so implementors may also provide their scopes. Spring does this across the portfolio in projects like Spring Web Flow and Spring Batch. You'll also see it in third-party projects like the Flowable workflow engine.

Let's look at an example where a Spring controller uses a request-scoped bean. The bean's state should change from one HTTP request to another, and the client - the HTTP controller - doesn't

need to do anything. It's as though the instance is swapped out from under neath it, in a way that's completely transparent to the client.

```
package com.example.aot.scopes;

import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Lazy;
import org.springframework.stereotype.Component;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.context.annotation.RequestScope;

import java.util.UUID;

@Configuration
class ScopesConfiguration {

}

①
@Component
@RequestScope
class RequestContext {

    private final String uuid = UUID.randomUUID().toString(); ②

    public String getUuid() {
        return uuid;
    }

}

@RestController
class ContextHttpController {

    private final RequestContext context;

    ③
    ContextHttpController(RequestContext context) {
        this.context = context;
    }

    @GetMapping("/scopes/context")
    String uuid() {
        return this.context.getUuid();
    }

}
```

① this bean is a `request` scoped, so it'll be created anew for each incoming HTTP request.

② this request-scoped bean should be different across different HTTP requests but the same for successive accesses during the same HTTP request.

③ Spring is giving us a proxy, which won't result in an actual instance until the bean starts its lifecycle, bound to whatever externalities govern them.

Compile the application and visit `http://localhost:8080/scopes/context` in your browser a few times. You should see that the `UUID` differs in each HTTP request.

Qualifiers

Qualifiers are conceptually very simple: given two types with the same interface, how does Spring choose which to inject in a given place? The answer is that we *qualify* the bean we'd like.

Suppose we're trying to build two applications with competing mobile phone marketplace implementations, like the Apple AppStore and Android's Play store.

We might model them with an interface, which we're calling `MobileMarketplace`. In this example, we have two implementations of that interface, and we've used the `@Qualifier` annotation on both the bean itself and the place where the Spring injects it. As long as the `String value` in the annotation matches, Spring will inject the correct instance. This mechanism goes even further: you can put `@Qualifier` on your custom annotation and then use that annotation instead of `@Qualifier` directly on the various implementations. This practice helps you enforce a ubiquitous language, making your code's domain more approachable.

Let's look at an example:

```
package com.example.aot.qualifiers;

import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.boot.context.event.ApplicationReadyEvent;
import org.springframework.context.ApplicationListener;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.stereotype.Service;

import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.util.Map;

@Configuration
class QualifiersConfiguration {

    ①
    @Bean
    ApplicationListener<ApplicationReadyEvent> android(@Android MobileMarketplace
mobileMarketplace) {
        return event -> System.out.println(mobileMarketplace.getClass().toString());
    }
```

```
    ②
    @Bean
    ApplicationListener<ApplicationReadyEvent> ios(@Apple MobileMarketplace
mobileMarketplace) {
        return event -> System.out.println(mobileMarketplace.getClass().toString());
    }

    ③
    @Bean
    ApplicationListener<ApplicationReadyEvent> mobileMarketplaceListener(
            Map<String, MobileMarketplace> mobileMarketplaces) {
        return event -> mobileMarketplaces
                .forEach((key, bean) -> System.out.println(key + '=' +
bean.getClass().getName()));
    }

}

④
@Retention(RetentionPolicy.RUNTIME)
@Qualifier("ios")
@interface Apple {

}

@Retention(RetentionPolicy.RUNTIME)
@Qualifier("android")
@interface Android {

}

interface MobileMarketplace {

}

⑤
@Service
@Qualifier("ios")
class AppStore implements MobileMarketplace {

}

@Service
@Qualifier("android")
class PlayStore implements MobileMarketplace {

}
```

① this bean uses the `android` implementation

② this bean uses the `apple` implementation

③ can't decide? Just inject a `Map<String,T>`, where `T` is the type you're looking for. Spring will provide a map of bean names to bean instances.

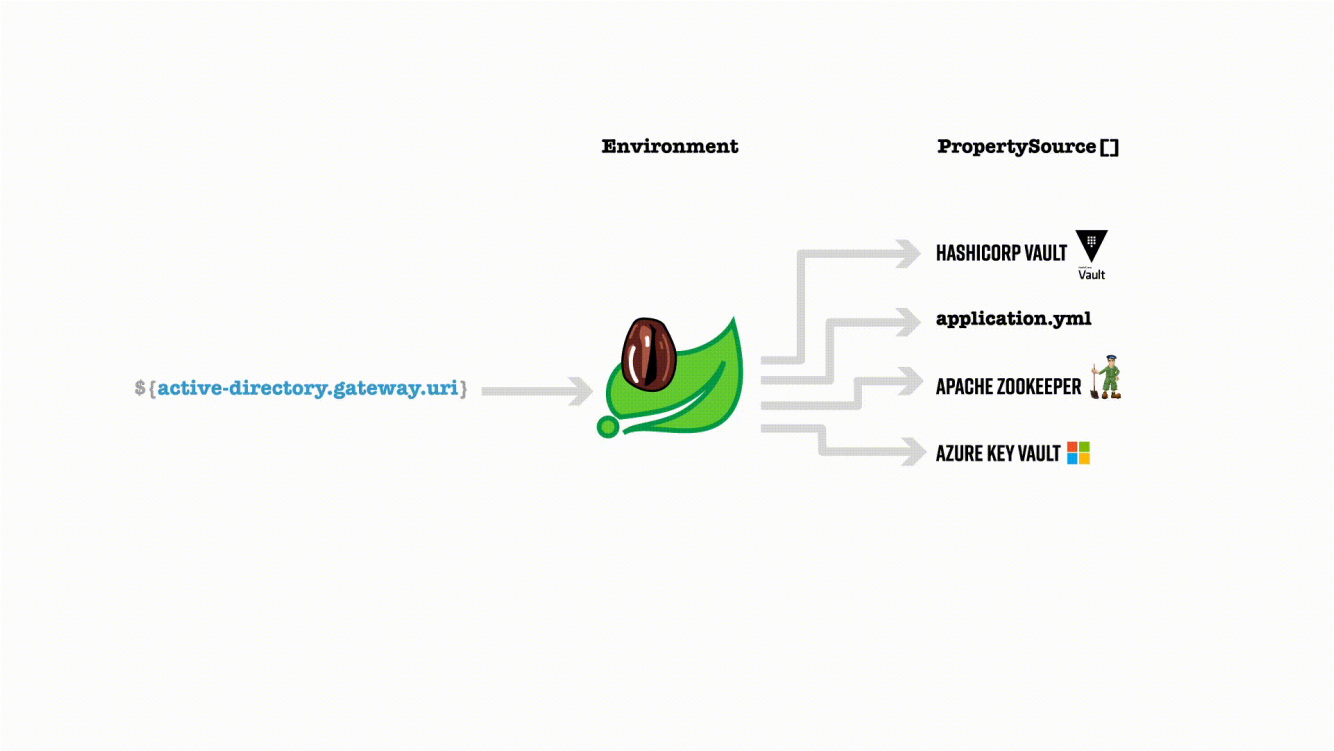
④ we create a meta-annotation.

⑤ we implement the interface

I love that I can directly define a bean's qualifier using the `@Qualifier` annotation and inject it into a particular site using the meta-annotation. Or vice versa, or both. Spring doesn't care. *It just works.*

Compile and run the code to the correct instances printed out.

Configuration Properties



Spring Framework provides the `Environment` abstraction, mapping between a `String` key and a `String` value. In addition, there is a strategy interface (`PropertyResolver`) for resolving these properties. Spring Boot can then take values in the `Environment` and map them to objects via setters or their constructors. Here's an example:

```
package com.example.aot.properties;

import org.springframework.boot.context.event.ApplicationReadyEvent;
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.boot.context.properties.EnableConfigurationProperties;
import org.springframework.context.ApplicationListener;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
@EnableConfigurationProperties(DemoProperties.class)
class PropertiesConfiguration {

    @Bean
    ApplicationListener<ApplicationReadyEvent> propertiesApplicationListener( ①
            DemoProperties properties) {
        return args -> System.out.println("the name is " + properties.name());
    }

}

②
@ConfigurationProperties(prefix = "bootiful")
record DemoProperties(String name) {
}
```

① Here, we're injecting a Java object called `DemoProperties`, to which properties starting with `bootiful` are bound

② the `@ConfigurationProperties` annotation wires Spring to inject properties onto an instance of `DemoProperties`.

Compile and run the code to see the values reflected in the output.

Detecting that you're in a Native Image

Sometimes hopefully not often! - you'll want to know when your code runs in a native image context. Knowing when your application is running is helpful because, as good as Spring's AOT engine is, we can't make it work perfectly in every situation, short of reviewing every line of code written. Some oddities arise from working in a GraalVM native image, and it's essential to be aware of those.

There is one helpful System property that you can use here: `org.graalvm.native image.imagecode`. We've encapsulated that check-in a method in Spring Framework: `NativeDetector.inNativeImage()`. So here it is in action.

```
package com.example.aot.detection;

import org.springframework.boot.context.event.ApplicationReadyEvent;
import org.springframework.context.ApplicationListener;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.NativeDetector;

@Configuration
class DetectionConfiguration {

    @Bean
    ApplicationListener<ApplicationReadyEvent> detectionListener() {
        return args -> System.out.println("is native image? " +
NativeDetector.inNativeImage());
    }

}
```

Run it on the JRE, and it'll return `false`. Run it in a GraalVM native image, and it'll return `true`.

My first instinct when I learned about that property was to wrap it in a `Condition` object and then use `@Conditional` to call that `Condition`, and thus make beans available in the Spring `BeanFactory` *conditionally*.

I tried it, and it didn't work.

Recall our discussion around the *phases* of a Spring Boot application. It starts up, *ingests* all the configuration, then creates a metamodel representation of the beans (`BeanDefinition` instances). Finally, Spring creates all the beans out of those `BeanDefinition` instances.

In an AOT application intended for GraalVM native image compilation, Spring Framework introduces a new phase during the compilation of the code. In this new phase, the Spring Boot build plugin creates a `BeanFactory` with populated `BeanDefinition` instances and stops there. Spring has a few new interfaces that are created and given a chance to inspect this `BeanFactory`, contributing whatever extra metadata is required to compile the beans in a GraalVM native image properly.

These interfaces also generate *new*, optimized code to recreate the state of the `BeanFactory` and skip the ingest phase. This new code and the contributed metadata are ultimately sent into the GraalVM native image compiler for compilation into a native image.

Spring only includes the beans in the final build present in the `BeanFactory` at compile time. So, if you have a bean that wasn't created at compile time because some `@Conditional` test evaluated to `false`, or because some profile wasn't active, it won't be there in the native image. To that end, I'd strongly urge you to avoid using profiles if you intend to create a GraalVM native image. There are some `@Conditional` annotations you should avoid, too.

Spring's AOT engine evaluates the `@Conditional` annotations at compile time. Some invariant conditions are the same at compilation time and runtime - like `@ConditionalOnClass` and

`@ConditionalOnProperty`. They work just fine in the native world: the classes present at compile time are, by definition, the classes present at runtime. However, some conditions depend on ambient state, like whether you're running in a Kubernetes cluster (`@ConditionalOnCloudPlatform(platform=CloudPlatform.Kubernetes)`). Avoid these conditions unless you plan on compiling your code in a Kubernetes cluster.

Application Migration

So far, everything works well out of the box. Our goal is that the vast majority of Spring applications have a path to upgrade to Spring Boot 3 and then enjoy the benefits of native image compilation. But, sometimes, things don't work as expected because some code has done something to run afoul of the cases, as mentioned earlier, where you'll need to furnish configuration `.json` files.

The error messages are pretty helpful, but the cycle time of compiling and running the application, getting a compiler error, making a change, then repeating can be tedious, especially when 30 seconds or longer compile times and long reset cycles are involved!

Once you know what configuration your application will need, it's pretty straightforward to craft the hints using the Java API. The trouble is, how do you figure out what hints are required? It can be tortuous to have an idea, make the change, then wait a minute (or more!) to find out if that worked and - if not - what the next error is. The reset cycle is what scares people. Mercifully, there are some pretty good ways to get a lot of this work done for you: running in AOT mode on the JRE and the GraalVM Java agent.

Run the AOT Code on the JRE

The first thing you might do is run the code in AOT mode. If you think about it, Spring Boot applications now have three different runtime destinations:

- traditional Spring Boot running on the JRE. This mode is the default and works exactly as it always has. Do nothing different, and you'll get this mode.
- code generated during the AOT phase of compilation, also run on the JRE
- code generated during the AOT phase of compilation, running in a GraalVM native image.

You get slightly more optimized performance when running AOT code on the JRE. Your application will perform considerably better when running the AOT code in a GraalVM native image. That second option - running AOT code on the JRE - is also interesting because it lets you preview what will get fed into the GraalVM native image compiler, which is more performant than the traditional behavior.

You can generate the AOT code without doing a full GraalVM reset:

```
mvn clean compile spring-boot:process-aot package
```

This command transforms your Spring application and generates GraalVM native image hints in

the `target/spring-aot` directory of your application. You can inspect that directory to see which GraalVM native image hints were generated (under the `target` directory) and what code was generated (under the `sources` directory).

There's a lot of interesting stuff happening here. The most important thing, from my perspective, is `AotApplication__BeanFactoryRegistrations.java`, in which all the `@Configuration`-annotated classes from the codebase appear transpiled into functional bean registrations. These functional bean registrations skip all the ingest and yield a `BeanFactory` that looks like it would have had we run the normal code.

```
...
import
org.springframework.boot.context.properties.ConfigurationPropertiesBindingPostProcesso
r__BeanDefinitions;
import
org.springframework.boot.sql.init.dependency.DatabaseInitializationDependencyConfigure
r__BeanDefinitions;
import
org.springframework.boot.validation.beanvalidation.MethodValidationExcludeFilter__Bean
Definitions;
import
org.springframework.boot.web.server.ErrorPageRegistrarBeanPostProcessor__BeanDefinitio
ns;
import
org.springframework.boot.web.server.WebServerFactoryCustomizerBeanPostProcessor__BeanD
efinitions;
import org.springframework.context.event.DefaultEventListenerFactory__BeanDefinitions;
import
org.springframework.context.event.EventListenerMethodProcessor__BeanDefinitions;
import
org.springframework.data.repository.core.support.PropertiesBasedNamedQueries__BeanDefi
nitions;
import
org.springframework.data.repository.core.support.RepositoryComposition__BeanDefinition
s;
import
org.springframework.data.web.config.ProjectingArgumentResolverRegistrar__BeanDefinitio
ns;
import
org.springframework.data.web.config.SpringDataJacksonConfiguration__BeanDefinitions;
import
org.springframework.data.web.config.SpringDataWebConfiguration__BeanDefinitions;
import
org.springframework.transaction.annotation.AbstractTransactionManagementConfiguration_
_BeanDefinitions;
import
org.springframework.transaction.annotation.ProxyTransactionManagementConfiguration__Be
anDefinitions;
import
org.springframework.web.servlet.config.annotation.WebMvcConfigurationSupport__BeanDefi
```

```
nitions;

/**
 * Register bean definitions for the bean factory.
 */
public class AotApplication__BeanFactoryRegistrations {
  /**
   * Register the bean definitions.
   */
  public void registerBeanDefinitions(DefaultListableBeanFactory beanFactory) {

beanFactory.registerBeanDefinition("org.springframework.context.event.internalEventLis
tenerProcessor",
EventListenerMethodProcessor__BeanDefinitions.getInternalEventListenerProcessorBeanDef
inition());

beanFactory.registerBeanDefinition("org.springframework.context.event.internalEventLis
tenerFactory",
DefaultEventListenerFactory__BeanDefinitions.getInternalEventListenerFactoryBeanDefini
tion());
    beanFactory.registerBeanDefinition("aotApplication",
AotApplication__BeanDefinitions.getAotApplicationBeanDefinition());
    beanFactory.registerBeanDefinition("basicsConfiguration",
BasicsConfiguration__BeanDefinitions.getBasicsConfigurationBeanDefinition());
    beanFactory.registerBeanDefinition("bfppConfiguration",
BfppConfiguration__BeanDefinitions.getBfppConfigurationBeanDefinition());
    beanFactory.registerBeanDefinition("compilationEndpointConfiguration",
CompilationEndpointConfiguration__BeanDefinitions.getCompilationEndpointConfigurationB
eanDefinition());
    beanFactory.registerBeanDefinition("orderService",
OrderService__BeanDefinitions.getOrderServiceBeanDefinition());
    beanFactory.registerBeanDefinition("proxiesConfiguration",
ProxiesConfiguration__BeanDefinitions.getProxiesConfigurationBeanDefinition());
    beanFactory.registerBeanDefinition("componentsConfiguration",
ComponentsConfiguration__BeanDefinitions.getComponentsConfigurationBeanDefinition());
    beanFactory.registerBeanDefinition("greetingsController",
GreetingsController__BeanDefinitions.getGreetingsControllerBeanDefinition());
    beanFactory.registerBeanDefinition("detectionConfiguration",
DetectionConfiguration__BeanDefinitions.getDetectionConfigurationBeanDefinition());
    beanFactory.registerBeanDefinition("eventsConfiguration",
EventsConfiguration__BeanDefinitions.getEventsConfigurationBeanDefinition());
    beanFactory.registerBeanDefinition("factoryBeansConfiguration",
FactoryBeansConfiguration__BeanDefinitions.getFactoryBeansConfigurationBeanDefinition(
));
    beanFactory.registerBeanDefinition("migrationsConfiguration",
MigrationsConfiguration__BeanDefinitions.getMigrationsConfigurationBeanDefinition());
    beanFactory.registerBeanDefinition("propertiesConfiguration",
PropertiesConfiguration__BeanDefinitions.getPropertiesConfigurationBeanDefinition());
    beanFactory.registerBeanDefinition("appStore",
AppStore__BeanDefinitions.getAppStoreBeanDefinition());
    beanFactory.registerBeanDefinition("playStore",
```

```
PlayStore__BeanDefinitions.getPlayStoreBeanDefinition());
...
```

There is more to discuss here, but that'll have to wait.

You can run the AOT-optimized and generated code on the JRE using the `-DspringAot=true` switch. Thus:

```
java -DspringAot=true -jar aot-0.0.1-SNAPSHOT.jar
```

You can configure the Spring Boot build plugin to run with that switch active by default, too:

```
<plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
    <configuration>
        <systemPropertyVariables>
            <springAot>true</springAot>
        </systemPropertyVariables>
    </configuration>
</plugin>
```

Run it in the usual way:

```
mvn spring-boot:run
```

The AOT mode is great because it's much faster than a complete GraalVM native image compilation and run. In addition, you'll find that because the AOT mode generates more efficient code that cuts out a lot of the extra `BeanDefinition` instances that might otherwise hang around, the resulting application can be slightly more efficient even just running on the JRE.

That said, running in the AOT mode is only useful if you're trying to sift through the generated files to confirm something is as it should be. It doesn't help you if you don't yet know what should be in the first place. This part can be tricky because it can be difficult to know what configuration is required.

Run the Java Agent

You can run the program and the agent alongside it and get the agent to tell you what things were reflected on, serialized, proxied, etc., by running the app on the JVM with `-Dagent=true` when running the app with maven (try `mvn spring-boot:run`). The agent dumps config files in the `target/native/agent-output` directory. You can inspect those and then use those to tell you what you'll need to contribute hints for. The Java agent dumps out *everything*, including some things that Spring's AOT engine will do automatically for you, and thus, you don't need to reproduce. I'd start with all the mentions of types across the various configuration files in the same packages as the code in your application. That way, you won't specify hints that Spring might've already done for

you.

Importantly: the program also prints out all the random stuff and changes from one run to another. YOu can usually identify these as the classes with dollar signs in them or UUID-like names. Ignore those until the very end. They're probably not interesting.

Configure the Spring Boot Maven build plugin to contribute a JVM argument:

```
<plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
    <configuration>
        <jvmArguments>
            -agentlib:native-image-agent=config-output-dir=target/native-image
        </jvmArguments>
        <!--....-->
    </configuration>
</plugin>
```

Then run it thusly:

```
./mvnw -DskipTests clean package spring-boot:run
```

Inspect the `target/native-image/` directory. You'll find that the Java agent has written out a `reflect-config.json` file that tells us something will reflect upon the `Person` type at runtime. We can also see an entry for the `data.csv` file in the `resource-config.json` file.

It would be best to exhaust as many code paths as possible while running the Java agent. Perhaps you could run it while running your tests? We want to ensure that you capture every scenario that may require configuration. Be sure to go through all the `.json` configuration files and note every instance where one of the types we've created - like `Person` - is present.

At this point, I'd disable the Java agent, commenting it out in the Maven `pom.xml`. It's served its purpose for now. Move the `native-image` directory aside, so it doesn't get deleted later. If you try to compile and run the application as a GraalVM native image and fail, you will want to consult these files.

Soft-touch Refactorings to Enable AOT

Let's look at a bare-bones example that could run afoul of GraalVM's native image compilation but for our interventions. Here's the completed program, complete with code to support the Spring AOT engine's work turning our application into a GraalVM native image:

```
package com.example.aot.migrations;

import com.fasterxml.jackson.databind.ObjectMapper;
import lombok.SneakyThrows;
```

```
import org.springframework.aot.hint.RuntimeHints;
import org.springframework.aot.hint.RuntimeHintsRegistrar;
import org.springframework.aot.hint.annotation.RegisterReflectionForBinding;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.context.event.ApplicationReadyEvent;
import org.springframework.context.ApplicationListener;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.ImportRuntimeHints;
import org.springframework.core.io.Resource;
import org.springframework.util.FileCopyUtils;

import java.io.InputStreamReader;
import java.util.stream.Stream;

@Configuration
class MigrationsConfiguration {

    ①
    record Person(String id, String name) {
    }

    @Bean
    @RegisterReflectionForBinding(Person.class) ②
    @ImportRuntimeHints(MigrationsRuntimeHintsRegistrar.class)
    ApplicationListener<ApplicationReadyEvent> peopleListener(ObjectMapper
objectMapper,
            @Value("classpath:/data.csv") Resource csv) { ③
        return new ApplicationListener<ApplicationReadyEvent>() {
            @SneakyThrows
            @Override
            public void onApplicationEvent(ApplicationReadyEvent event) {
                try (var in = new InputStreamReader(csv.getInputStream())) {
                    var csvData = FileCopyUtils.copyToString(in);
                    Stream.of(csvData.split(System.lineSeparator())).map(line ->
line.split(","))
                            .map(row -> new Person(row[0], row[1])).map(person ->
json(person, objectMapper))
                            .forEach(System.out::println);
                }
            }
        };
    }

    @SneakyThrows
    private static String json(Person person, ObjectMapper objectMapper) {
        return objectMapper.writeValueAsString(person);
    }

    static class MigrationsRuntimeHintsRegistrar implements RuntimeHintsRegistrar {
```

```
        @Override
        public void registerHints(RuntimeHints hints, ClassLoader classLoader) {
            hints.resources().registerPattern("data.csv");

            ④
            // hints.reflection().registerType(Person.class, MemberCategory.values());
        }

    }

}
```

① this program simply reads in CSV data (in the least production-worthy way possible!) and maps the rows into `Person` instances

② we know that the Jackson JSON marshaling library will need to reflect on those `Person` instances, so we've used Spring's convenient `@RegisterReflectionForBinding` annotation to tell Spring to proactively register the GraalVM hints required to support that. This annotation must be on or in a Spring component or `@Configuration` class.

③ we know that this program will need to read the `.csv` file from the `.jar`, a resource, which will be a problem for GraalVM unless we contribute some hints. We use the `@ImportRuntimeHints` annotation to tell Spring to involve a class of type `RuntimeHintsRegistrar` in the compilation process so that we can programmatically contribute those hints.

④ we *could*, as an alternative to using the `@RegisterReflectionForBinding` annotation, have also contributed a hint for reflective access for that type.

You'll need to create a file with sample data called `src/main/resources/data.csv`. Here's the one I put in the code:

```
1,Josh
2,Stéphane
3,Andy
4,Madhura
5,Olga
6,Yuxin
7,Violetta
8,Spencer
9,Chloe
10,Dr. Syer
```

Run this on the JRE, and it should just work. As expected.

Now, for our example to work - or, instead, *not* work, in a GraalVM native image, you'll need to comment out the lines where we use the `@RegisterReflectionForBinding` and `@ImportRuntimeHints` annotations. Then, run and compile the program. You should see failures when you start it up as a GraalVM native image.

The first error I got was related to being unable to load the `.csv` file. So, restore the `@ImportRuntimeHints` annotation. Compile and run that as a GraalVM native image. Next, you'll get

an error about serializing the `Person`. Restore the line with `@RegisterReflectionForBinding`, and everything should work.

Compile and run the program; you should see the `Person` records printed on the console.

This program uses the Jackson JSON marshaling library directly to print out JSON representations of some objects. This example is a bit contrived: when was the last time you used Jackson to print out some objects, and *that's all*? It seldom happens. If this program had a controller that returned the `Person`, Spring would automatically know to register hints for it. There are a lot of common cases like this where Spring will *just work*.

Spring will still need a custom hint for the `.csv` file.

Processing the Bean Factory

Spring makes it trivial to act on a collection of `BeanDefinition` instances through callback interfaces evaluated at compile time (AOT) and runtime (JRE). For example, the `BeanFactoryPostProcessor` is a callback interface that lets us access the `BeanFactory` and manipulate the `BeanDefinition` instances. We can register new ones, update existing ones, or even remove them in this interface and other, more specific subclasses of this interface.

The `BeanFactory` is mutable at this point. Various Spring projects use this fact to great effect. Before the Spring team turned the Spring Cloud project into the microservices sensation it is today; its sole function was to make it easy for Spring Boot applications deployed to a Platform-as-a-Service (PaaS) (like Cloud Foundry or Heroku) to connect to managed infrastructure like a database or a message queue. It did this by analyzing the `BeanFactory`, identifying infrastructure-related beans like `javax.sql.DataSource`, and then replacing them with a new bean of an identical interface, but with a connection that is pointing to the managed infrastructure identified by environment variables in the process space of the running application. So, you write an application using a `DataSource` talking to an embedded H2 instance on your local machine. Still, when you deploy it, this `BeanFactoryPostProcessor` identifies connection strings in environment variables in the process space for the application and uses that to create a proper `DataSource` pointing to managed infrastructure, presumably on another host and port. This process was transparent to the user, thanks to the magic of `BeanFactoryPostProcessor` implementations.

The `BeanFactoryPostProcessor` is a great place to see *everything* in the `BeanFactory` all in one place when the application starts up.

The `BeanFactoryInitializationAotProcessor` is a new interface that is a peer to the `BeanFactoryPostProcessor`. It runs at compile time, and you have the same contract: you can contribute hints (or even do code-generation) while analyzing the `BeanDefinition` instances in the application.

You must work only in terms of the `BeanDefinition` instances and bean names in both these interfaces. Remember, Spring hasn't created any of the beans at this point. So, while you'll be able to call `BeanFactory#getBean(String)`, it'll force Spring to eagerly initialize the object, calling the constructor and the methods and so on before the bean is ready. Don't do this - it'll screw things up!

Let's look at a trivial example. We'll write a `BeanFactoryPostProcessor` that analyzes and

programmatically registers a new `BeanDefinition` in the application context. Would you ever need to do this? No. But it's hopefully illustrative. We'll use a subclass of `BeanFactoryPostProcessor` called `BeanDefinitionRegistryPostProcessor`, which gives us a `BeanFactory` downcast to a specific subtype that supports programmatically registering new beans. The trouble is that this bean we'll register will require some reflection using the Jackson JSON API, so we'll need to also register a GraalVM hint for it using a `BeanFactoryInitializationAotProcessor` contribution.

```
package com.example.aot.bfpp;

import com.fasterxml.jackson.databind.ObjectMapper;
import lombok.SneakyThrows;
import org.springframework.aot.hint.MemberCategory;
import org.springframework.beans.BeansException;
import org.springframework.beans.factory.aot.BeanFactoryInitializationAotContribution;
import org.springframework.beans.factory.aot.BeanFactoryInitializationAotProcessor;
import org.springframework.beans.factory.config.ConfigurableListableBeanFactory;
import org.springframework.beans.factory.support.BeanDefinitionBuilder;
import org.springframework.beans.factory.support.BeanDefinitionRegistry;
import org.springframework.beans.factory.support.BeanDefinitionRegistryPostProcessor;
import org.springframework.boot.context.event.ApplicationReadyEvent;
import org.springframework.context.ApplicationListener;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import java.util.List;
import java.util.UUID;

import static com.example.aot.bfpp.BfppConfiguration.BEAN_NAME;

@Configuration
class BfppConfiguration {

    ①
    static String BEAN_NAME = "myBfppListener";

    ②
    @Bean
    static ListenerBeanFactoryPostProcessor listenerBeanFactoryPostProcessor() {
        return new ListenerBeanFactoryPostProcessor();
    }

    @Bean
    static ListenerBeanFactoryInitializationAotProcessor
listenerBeanFactoryInitializationAotProcessor() {
        return new ListenerBeanFactoryInitializationAotProcessor();
    }

}

③
```

```
class Listener implements ApplicationListener<ApplicationReadyEvent> {

    private final ObjectMapper objectMapper;

    Listener(ObjectMapper objectMapper) {
        this.objectMapper = objectMapper;
    }

    @Override
    @SneakyThrows
    public void onApplicationEvent(ApplicationReadyEvent event) {
        var products = List.of(new Product(UUID.randomUUID().toString()), new
Product(UUID.randomUUID().toString()));
        for (var p : products)
            System.out.println(objectMapper.writeValueAsString(p));
    }

}

record Product(String sku) {
}

④
class ListenerBeanFactoryPostProcessor implements BeanDefinitionRegistryPostProcessor
{

    @Override
    public void postProcessBeanFactory(ConfigurableListableBeanFactory bf) throws
BeansException {
    }

    @Override
    public void postProcessBeanDefinitionRegistry(BeanDefinitionRegistry registry)
throws BeansException {
        if (!registry.containsBeanDefinition(BEAN_NAME))
            registry.registerBeanDefinition(BEAN_NAME,

BeanDefinitionBuilder.rootBeanDefinition("com.example.aot.bfpp.Listener").getBeanDefin
ition());
    }

}

⑤
class ListenerBeanFactoryInitializationAotProcessor implements
BeanFactoryInitializationAotProcessor {

    @Override
    public BeanFactoryInitializationAotContribution
processAheadOfTime(ConfigurableListableBeanFactory bf) {
```

```
        if (bf.containsBeanDefinition(BEAN_NAME)) {
            return (ctx, code) -> {
                var hints = ctx.getRuntimeHints();
                hints.reflection().registerType(Product.class,
MemberCategory.values());
            };
        }
        return null;
    }

}
```

① we were going to work with the same bean at compile time and runtime. Here, we create a variable with the name of the bean for subsequent access from both interface implementations at both phases (AOT and runtime)

② Note that we use the `static` keyword with both bean definition registrations. Spring will involve these beans very early in the lifecycle of the `BeanFactory`. There will not be any live-fire beans. So, use `static` to avoid weird lifecycle issues. Avoid depending on anything from the `BeanFactory`. Don't force Spring to construct the `@Configuration` class containing the `@Bean` registration methods, too.

③ the `Listener` class is a trivia `ApplicationListener` that, when run, will iterate over a collection of dummy DTOs and print them out using Jackson for JSON serialization. This serialization involves reflection, for which we'll need to furnish hints.

④ the first callback, the `ListenerBeanFactoryPostProcessor`, programmatically registers a new `BeanDefinition` of type `Listener` if a bean of the name we've specified in the variable doesn't already exist in the context. This object runs at runtime in both the JRE and the AOT application. Here we mutate the `BeanFactory`.

⑤ Spring's AOT engine will involve the `ListenerBeanFactoryInitializationAotProcessor` during the compilation phase so that it may furnish hints to make the JSON serialization of the `Product` records work.

These two interfaces work well together. They complement each other. The `BeanFactory` is the lowest level against which I'd write code when manipulating the Spring application context. I don't want to write code in terms of the `BeanFactory` if I can avoid it. The `BeanFactory` suggests a valid working application. But it's not a working application. Spring must first turn the `BeanDefinition` instances into valid objects, or *beans*. That part comes next.

Processing Beans

Suppose the `BeanFactory` is too meta (no, not *that* Meta!) for you. In that case, you can still do interesting things at the next rung in the abstraction ladder, working with beans directly, both before initialization and after.

Working on a bean-by-bean basis can be very powerful. If you want to work with actual, live-fire beans, you can use Spring's `BeanPostProcessor`. This interface puts you in a position to act on and transform objects before they're finally live and handling logic. `BeanPostProcessor` instances are great for infrastructure code, such as frameworks, where you need to note, retain, or observe

references to objects of a given shape. What shape? Well, anything! Objects that have a certain marker interface. Objects that have a certain annotation. Whatever you want and whatever you could discover, given reflection. Let's look at an example that creates proxies for beans with an annotation, `@Logged`, logging out any method invocations.

I implement this proxy with Spring's `ProxyFactory`, which makes it trivial to use the *proxy* design pattern. In its most general form, a proxy is a class functioning as an interface to something else. Spring uses proxies to handle declarative transaction management, auditing, security, logging, concurrency, etc. there is a great way to *decorate* an existing object with cross-cutting concerns, like starting and committing a transaction before and after a method invocation. Spring uses them all over the place for things like `@Transactional`, `@Scheduled`, `@Async`, `@Authorized`, and countless more.

There are two types of proxies: CGLIB and JDK. Proxies make it trivial to create an object that implements an interface type of your choice and then forwards the actual work to a concrete instance of your choice. JDK proxies are built with Java's `InvocationHandler`. In the case of JDK proxies, the word `interface` used in the description above is quite literally a Java `interface` type.

But what if the contract of the surface area, the *interface* of your class, isn't a Java interface? What if it's a concrete class for which it makes little sense to extract a separate Java interface? JDK proxies require an interface, and they were the only thing supported in Spring until Spring Framework 1.1.

NOTE

I can't be sure, but I suspect this is one of the reasons so much of the early literature for Spring Framework suggests using interfaces with Spring beans and why you'd see things like `FooService` and `DefaultFooService` or `FooService` and `FooServiceImpl` or whatever. Nowadays, that's an anti-pattern and highly discouraged, *unless* you plan to have more than one implementation of `FooService`. Avoid the knee-jerk reaction to create an interface for an object that has the same shape as the object. It only complicates things.

Spring supports concrete proxies, too, using CGLIB. CGLIB is used to dynamically generate the code to subclass an existing type. The constraint, thus, is that the type is subclassable. So, beware of things like `final` and `sealed`.

Spring's concrete proxies are unique to Spring, but they're *everywhere*. You use them every time you use a `@Configuration` class!

Naturally, creating a dynamic subclass of a given type, registering it in the `ClassLoader`, and then swapping out your instance for the instance that delegates to yours imply a lot of funny business that we'll need to account for everything at compile time in a GraalVM native image context.

Proxies

Mercifully, Spring can do a lot of this for us. Let's take a look at an example.

Let's look at a simple proxy example of the logs information whenever somebody invokes a method annotated called `@Logged`, which we'll create. The annotation is nice because it's decorative. We're layering capabilities onto this method without complicating the business logic implementation.

```
package com.example.aot.bpp.proxies;

import org.aopalliance.intercept.MethodInterceptor;
import org.springframework.aop.framework.ProxyFactory;
import org.springframework.beans.BeansException;
import
org.springframework.beans.factory.config.SmartInstantiationAwareBeanPostProcessor;
import org.springframework.boot.context.event.ApplicationReadyEvent;
import org.springframework.context.ApplicationListener;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.stereotype.Service;
import org.springframework.util.ReflectionUtils;

import java.lang.annotation.Inherited;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

@Configuration
class ProxiesConfiguration {

    ①
    @Bean
    ApplicationListener<ApplicationReadyEvent> loggedListener(OrderService service) {
        return event -> service.addToPrices(7);
    }

    @Bean
    LoggedBeanPostProcessor loggedBeanPostProcessor() {
        return new LoggedBeanPostProcessor();
    }

}

②
class LoggedBeanPostProcessor implements SmartInstantiationAwareBeanPostProcessor {

    ③
    private static ProxyFactory proxyFactory(Object target, Class<?> targetClass) {
        var pf = new ProxyFactory();
        pf.setTargetClass(targetClass);
        pf.setInterfaces(targetClass.getInterfaces());
        pf.setProxyTargetClass(true); ④
        pf.addAdvice((MethodInterceptor) invocation -> {
            var methodName = invocation.getMethod().getName();
            System.out.println("before " + methodName);
            var result = invocation.getMethod().invoke(target,
```

```
invocation.getArguments());
            System.out.println("after " + methodName);
            return result;
        });
        if (null != target) {
            pf.setTarget(target);
        }
        return pf;
    }

    ⑤
    private static boolean matches(Class<?> clazzName) {
        return clazzName != null && (clazzName.getAnnotation(Logged.class) != null ||
ReflectionUtils
                .getUniqueDeclaredMethods(clazzName, method ->
method.getAnnotation(Logged.class) != null).length > 0);
    }

    ⑥
    @Override
    public Class<?> determineBeanType(Class<?> beanClass, String beanName) throws
BeansException {
        if (matches(beanClass)) {
            return proxyFactory(null,
beanClass).getProxyClass(beanClass.getClassLoader());
        }
        return beanClass;
    }

    ⑦
    @Override
    public Object postProcessAfterInitialization(Object bean, String beanName) throws
BeansException {
        var beanClass = bean.getClass();
        if (matches(beanClass)) {
            return proxyFactory(bean, beanClass).getProxy(beanClass.getClassLoader());
        }
        return bean;
    }

}

@Inherited
@Target({ TYPE, METHOD })
@Retention(RUNTIME)
@interface Logged {

}

@Logged
@Service
```

```
class OrderService {

    public void addToPrices(double amount) {
        System.out.println("adding $" + amount);
    }

}
```

① as usual, we'll try things out when the application starts up

② we're going to create this proxy in a subclass of `BeanPostProcessor` called `SmartInstantiationAwareBeanPostProcessor`. Spring's AOT engine invokes the `SmartInstantiationAwareBeanPostProcessor#determineBeanType` to determine what the bean should be. Spring's AOT engine uses that information to determine what proxies to build. When the application starts up completely, Spring will also invoke `SmartInstantiationAwareBeanPostProcessor#postProcessAfterInitialization(Object bean, String beanName)`, giving you a chance to inspect a given bean and -

③ we need to both build a proxy and determine the resulting class of that proxy at various phases in the lifecycle of the `SmartInstantiationAwareBeanPostProcessor`. So, we'll return the builder, an instance of `ProxyFactory`, and we'll either use it to determine the expected proxy's class type or create the final proxy object.

④ setting `proxyTargetClass` to `true` and specifying a `targetClass` results in a CGLIB proxy. Otherwise, the expectation is that you're only proxying interfaces.

⑤ `BeanPostProcessor` implementations visit every bean in the `BeanFactory`, so we must only act on those with the annotation present. The `matches` method encodes this logic that looks for all types annotated with, or with methods annotated with, the `@Logged` annotation.

⑥ the `SmartInstantiationAwareBeanPostProcessor#determineBeanType` method does almost all the work of creating a proxy but stops short of actually creating the proxy. We're only interested in knowing what the class would be *if* we did create a proxy. This information then gets fed into the AOT engine.

⑦ here's where the rubber meets the road, and Spring creates a CGLIB proxy at runtime. The proxy creation works fine, as Spring will have already registered the requisite hints with GraalVM during compilation.

In this example, we're using the cohesive design of Spring's AOT engine to transparently create proxies that *just work*. If we on the Spring team have done our job right, and you've written your code to leverage the right interfaces, then you can trust Spring to do the right thing for CGLIB proxies.

Compile and run the application, and you should see information logged when we invoke the method in the `ApplicationListener`.

Code Generation

Now, before we go much further, let me just stress that this *isn't* the return of Spring Roo!

But code generation is a powerful part of the Spring AOT engine. It manifests in two ways: generating the requisite `.json` configuration files for GraalVM native images, and generating Java code, in `.java` files, from whole cloth.

Suppose you're using a library, and it is using `InvocationHandler`, outside of Spring's purview. You'll need to furnish the appropriate hints for GraalVM knows what to do there. You might also have stuff you want to do at runtime that requires upfront, compile-time processing. Code generation is one of the most powerful dimensions of Spring's AOT engine. We use it all over the place in Spring Framework 6! You can both furnish configuration and generate code at compile with implementations of the AOT equivalent to `BeanPostProcessor`, the `BeanRegistrationAotProcessor`.

Let's look at a simple example. This time we'll revisit the code generation but go even further. We're going to register a brand new bean during compilation, overwriting one that's already present, so that we can capture compile-time information and put it in the endpoint. This Actuator endpoint, the `CompilationEndpoint` is suitably trivial as to be understandable in a quick introduction like this one. This new `CompilationEndpoint` will capture information about the ambient state of the build *at compile-time*, like the time and directory in which the code is compiled.

```
package com.example.aot.bpp.code;

import org.springframework.beans.factory.aot.BeanRegistrationAotContribution;
import org.springframework.beans.factory.aot.BeanRegistrationAotProcessor;
import org.springframework.beans.factory.support.RegisteredBean;
```

```
import org.springframework.boot.actuate.endpoint.annotation.Endpoint;
import org.springframework.boot.actuate.endpoint.annotation.ReadOperation;
import org.springframework.boot.context.event.ApplicationReadyEvent;
import org.springframework.context.ApplicationListener;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.javapoet.CodeBlock;

import javax.lang.model.element.Modifier;
import java.io.File;
import java.time.Instant;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

@Configuration
class CompilationEndpointConfiguration {

    ①
    @Bean
    CompilationEndpoint compilationEndpoint() {
        return new CompilationEndpoint();
    }

    ②
    @Bean
    CompilationEndpointBeanRegistrationAotProcessor
compilationEndpointBeanRegistrationAotProcessor() {
        return new CompilationEndpointBeanRegistrationAotProcessor();
    }

    ③
    @Bean
    ApplicationListener<ApplicationReadyEvent>
compilationEndpointListener(CompilationEndpoint endpoint) {
        return event -> {
            var map = endpoint.compilation();
            for (var e : map.entrySet()) {
                System.out.println(e.getKey() + '=' + e.getValue());
            }
        };
    }

}

④
@Endpoint(id = "compilation")
class CompilationEndpoint {

    private final Map<String, Object> map = new ConcurrentHashMap<>();

    CompilationEndpoint() { // default runtime version
```

```
    }

    CompilationEndpoint(Instant instant, File directory) {
        map.putAll(Map.of("instant", instant, "directory", directory));
    }

    @ReadOperation
    public Map<String, Object> compilation() {
        return Map.of("compilation", this.map, "now", Instant.now());
    }

}

⑤
class CompilationEndpointBeanRegistrationAotProcessor implements
BeanRegistrationAotProcessor {

    @Override
    public BeanRegistrationAotContribution processAheadOfTime(RegisteredBean
registeredBean) {

        if
(!CompilationEndpoint.class.isAssignableFrom(registeredBean.getBeanClass()))
            return null;

        return (ctx, code) -> {

            var generatedClasses = ctx.getGeneratedClasses();

            var generatedClass = generatedClasses.getOrAddForFeatureComponent(
                    CompilationEndpoint.class.getSimpleName() + "Feature",
CompilationEndpoint.class,
                    b -> b.addModifiers(Modifier.PUBLIC));

            var generatedMethod =
generatedClass.getMethods().add("postProcessCompilationEndpoint", build -> {

                var outputBeanVariableName = "outputBean";
                build.addModifiers(Modifier.PUBLIC, Modifier.STATIC)
                        .addParameter(RegisteredBean.class, "registeredBean") //
                        .addParameter(CompilationEndpoint.class, "inputBean")//
                        .returns(CompilationEndpoint.class)
                        ⑥
                        .addCode(CodeBlock.builder()
                                .addStatement("$T $L = new $T( $T.ofEpochMilli($L),
new $T($S))",
                                        CompilationEndpoint.class,
outputBeanVariableName, CompilationEndpoint.class,
                                        Instant.class, System.currentTimeMillis() +
"L", File.class,
                                        new File(".").getAbsolutePath()
```

```
                                ).addStatement("return $L",
outputBeanVariableName).build());
            });
            var methodReference = generatedMethod.toMethodReference();
            code.addInstancePostProcessor(methodReference);
        };
    }

}
```

① we're going to register an empty, no-op version of the `CompilationEndpoint` here

② we will post-process the bean at compile time using an implementation of a `BeanRegistrationAotProcessor`

③ as usual, we'll test things out when the application starts.

④ Have you ever seen how to register a Spring Boot Actuator endpoint? Well, now you have. It's easy. Actuator Endpoints should be agnostic of whichever rendering mechanism - HTTP, JMX, etc. - you might use. This endpoint exports key-value pairs in a `Map<String, Object>` returned from the `compilation()` read operation. Note that we have *two* constructors: one for the no-op case and another for information about the build, like the time and directory of the compilation.

⑤ Like the `BeanPostProcessor`, `BeanRegistrationAotProcessor` implementations work on a bean-by-bean basis, getting a chance to inspect and change them. We again use the Java Poet abstraction to register new code, a method named `postProcessCompilationEndpoint`, which takes an instance of type `RegisteredBean` and another instance of type `CompilationEndpoint`. The `RegisteredBean` allows us to inspect the reflective metadata associated with the bean, and of course, the `CompilationEndpoint` is the bean we created earlier. If the user doesn't use Spring's AOT engine, then the bean we looked at earlier is the only bean they'll ever get. However, if they use the AOT engine, this method will return a *new* `CompilationEndpoint`, initialized with values during compilation.

⑥ the key bit starts here: we use Java Poet to create a new instance of `CompilationEndpoint` with real values initialized and captured during the AOT phase itself. Now *that's* meta!

Compile the application and then wait a minute. Then run the application. You should see that the information about the compilation, which will have occurred a minute or earlier, will be printed in the console.

You can see the output of what we've just done in the resulting generated Java code:

```
package com.example.aot.bpp.code;

import java.io.File;
import java.time.Instant;
import org.springframework.beans.factory.support.RegisteredBean;

public class CompilationEndpoint__CompilationEndpointFeature {
  public static CompilationEndpoint postProcessCompilationEndpoint(RegisteredBean
registeredBean,
      CompilationEndpoint inputBean) {
    CompilationEndpoint outputBean = new CompilationEndpoint(
Instant.ofEpochMilli(1675397823616L), new File("/Users/jlong/Desktop/spring-boot-3-
aot/."));
    return outputBean;
  }
}
```

Pretty tricky, eh? We've written code to write out more code and transform objects whose definitions we ultimately end into the GraalVM native image compiler. It is kind of mind-bending if you ask me.

Testing

Indeed, *all* of this GraalVM AOT native image machinery is kind of mind-bending to me, if I'm honest! But, in all that we've seen, I hope you feel that you know what to look at to identify and work around issues you discover. Of course, fixing an issue is critical to success with technology. But how do you ensure that it *stays* fixed going forward? Here is where Spring's rich testing support comes in. It, too, has been updated to work in a GraalVM native image context!

Testing is an incredibly important part of the story. The good news is we've done a ton of work to make testing **just work** in a native image context.

You can run Maven, or Gradle builds in native mode.

```
mvn -PnativeTest test
```

You'll get two output artifacts: one for tests and another for the production code. Fair warning tho! Producing the test code as a native image doubles the interminable compile time! That could be a few minutes. When the compilation finishes, there'll be a binary in the `target` directory called `native-tests`. Run that to execute the test code as a native image.

```
package com.example.aot.basics;

import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

①
@SpringBootTest(properties = "spring.main.web-application-type=none")
class CustomerRepositoryTest {

    private final CustomerRepository repository;

    CustomerRepositoryTest(@Autowired CustomerRepository repository) {
        this.repository = repository;
    }

    @Test
    void persist() {
        var saved = repository.save(new Customer(null, "Name"));
        Assertions.assertNotNull(saved.id());
        Assertions.assertNotNull(saved);
    }

}
```

① we are creating a Spring `ApplicationContext` and then using it to inject references to beans, like the `CustomerRepository` we created when we looked at AOT fundamentals earlier.

The basics work here. There are some exceptions; you may encounter issues with Mockito. But the basics work. Try it out!

It's good that the basics work, too. So does it mean that we can run the usual test harness against our business logic and confirm to whatever extent we'd like that things work as they did before. but what about the new code designed to support turning your application into a GraalVM native image? What about the hints you register for a given type using the Spring AOT engine? Spring has you covered. You can use the new `RuntimeHintsPredicates` type to confirm that your hints are working as expected.

```
package com.example.aot.migrations;

import org.assertj.core.api.Assertions;
import org.junit.jupiter.api.Test;
import org.springframework.aot.hint.RuntimeHints;
import org.springframework.aot.hint.predicate.RuntimeHintsPredicates;

class MigrationsConfigurationTest {

    @Test
    void hints() {
        var hints = new RuntimeHints();
        ①
        var registrar = new MigrationsConfiguration.MigrationsRuntimeHintsRegistrar();
        var classloader = getClass().getClassLoader();
        ②
        registrar.registerHints(hints, classloader);
        ③

Assertions.assertThat(RuntimeHintsPredicates.resource().forResource("data.csv")).accep
ts(hints);

    }

}
```

① create an instance of our implementation on the `RuntimeHintsRegistrar` interface

② call `RuntimeHintsRegistrar#registerHints` on our implementation, passing in mock instances of `RuntimeHints` and the current `ClassLoader`

③ use the `RuntimeHintsPredicates` type to make certain assertions about the state of the `RuntimeHints` afterward

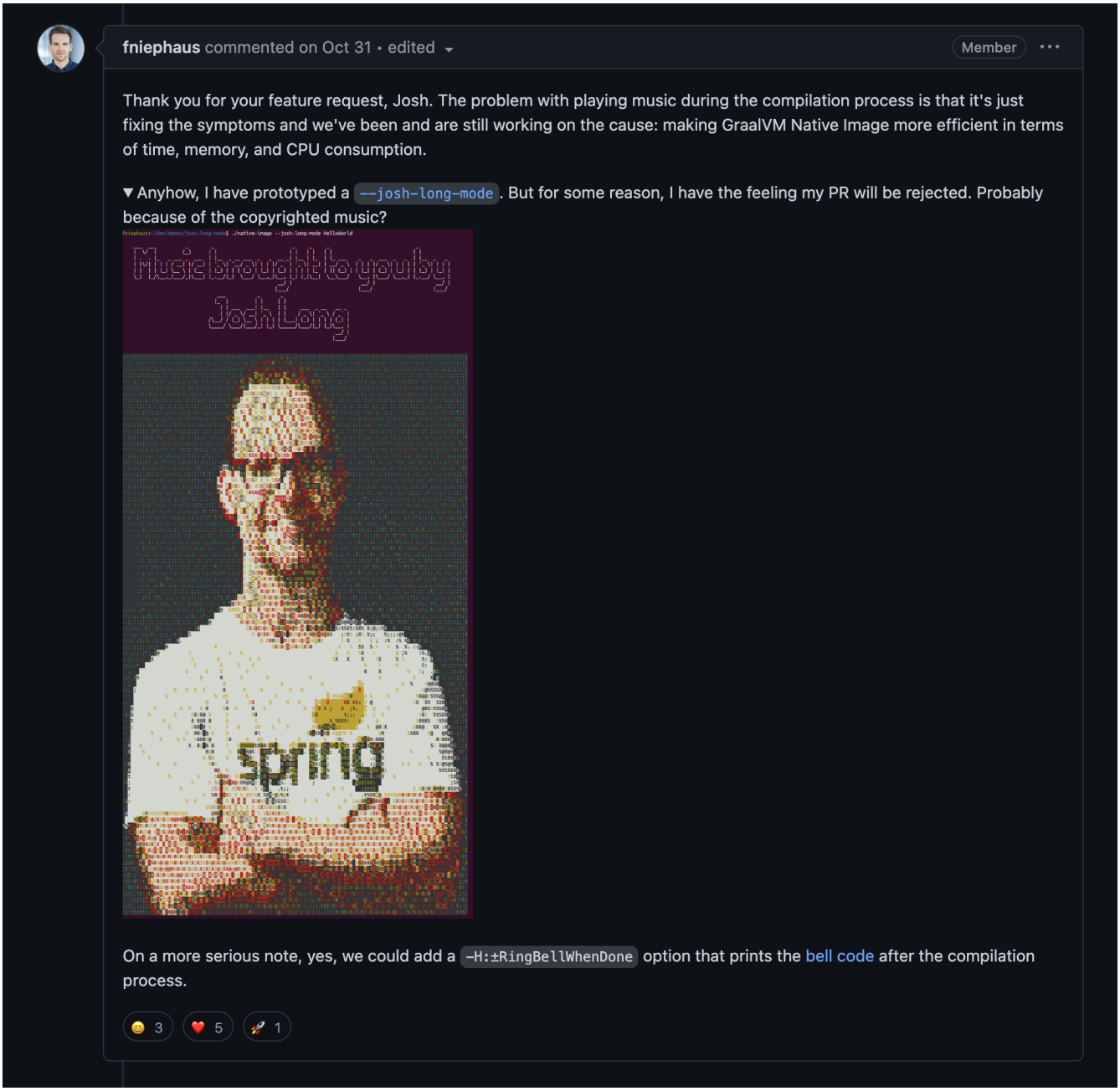
The testing support in Spring is exhaustive, and the Spring Boot even more so. It is not surprising, thus, that Spring's new AOT engine would further move the needle.

A Note on Compile Times

You know, I love GraalVM, but one thing that can be a real kick in the pants is the wait times. I get *why* it's happening. It's not hard to understand. GraalVM does a *lot* of work - doing a full analysis of our code and all the code on the classpath. So it's no wonder it takes as much memory and time as it does. And it has already improved *dramatically* even since I first started using it in earnest less than three years ago. It's a marvelous piece of software. But those compile times are tedious. They interrupt my sense of flow. These prolonged builds remind me of the famous XKCD "compiling" cartoon.

And it's almost a curse that they've improved as much as they have because the compilation times are now fast enough that you don't have time to go pour a cup of coffee or eat a cup of yogurt but slow enough that you feel like you're being interrupted. You're being kicked out of 'the zone. This

reminds me of that classic XKCD cartoon, which I suspect referred to large C and C++ applications of the day. Java, Go, and other more modern languages have quick compilation times. In this one sense, it feels like GraalVM has taken us a step backward. I kick off a compilation, and my mind wanders. I've been doing so many of these that I'm starting to hear elevator music when I compile! Don't you hear elevator music? So I filed an issue. I want everyone *else* to hear elevator music, too. So I asked. It doesn't hurt to ask. I went to the official GraalVM project, and I asked. it seemed to resonate with people. One person even suggested as to which elevator music should be played (one of the soundtracks from the Bond film *Goldeneye*, natch). I love it! And, Fabio Niephaus - of the official GraalVM team members at Oracle, a Ph.D. - a DOCTOR, like Drs. Seus, Syer, Strange, Who, and Subramaniam - responded with *a promising prototype*.



To say that I love this prototype would be an understatement! I was sad to see that he didn't think it would be merged, if for no other reason than because of the copyrighted music. Dang! Oh well. Hope *springs* eternal.

But, something good *did* come of this: there are some interesting ideas discussed around how to have a bell chime (or indeed any other sound played) when the compilation finishes. At least this

way, you'd never feel like you wasted precious time if you stepped away for a minute!

Conclusion

We've come a long way! When I started covering Spring Native (the predecessor to what would eventually become Spring Framework 6's AOT engine) in 2020, a build took *10 minutes* and only worked for the smallest and most specific scenarios. Now, here we are in 2022, and I can all but effortlessly get a Spring app working in a GraalVM native image context in less than a minute.

The performance and memory usage implications are hard to ignore. As the community embraces Spring's newfound AOT support, I think the inertia required to move to AOT will become easier and easier to obtain in the next days, weeks, months, and years. I hope this video makes it easier for application and framework developers to leverage Spring Boot 3 and Spring Framework 6. You won't be starved for examples if you need guidance in implementing this in your code. Look for pointers to types like `RuntimeHintsRegistrar`, `BeanRegistrationAotProcessor`, and `BeanFactoryInitializationAotProcessor` in the Spring Boot codebase! And those don't even begin to cover the projects' implementations, like Spring Security, Spring Data, Spring Cloud, Spring Shell, Spring Integration, and Spring Batch. We've only just begun to scratch the surface here, my friends. And so, too, has the community. That said, things are moving *quickly*: As I write this, there's already work nearly- or already- completed for various opensource projects like Axon, Vaadin, Hilla Framework, MyBatis, JHipster, the official Kubernetes Java client, JobRunr, so many others besides.

GraalVM native image technology could be a *very* powerful way to build applications better suited for production. GraalVM native images save costs and help you build more reliable systems. In addition, Spring's AOT engine helps you introduce new possibilities, like serverless, embedded, and infrastructure, for which Spring may not have been considered ideal in the past.

I hope this humble text has helped you get started and that you can go to the Spring Initialzr and generate your next value-producing, production-bound Spring Boot application with GraalVM native image support!