

Everything You Never Wanted to Know About Spring Boot 3 AOT

By Josh Long

Table of contents

Introduction	3
Understanding Spring	3
Ingest	3
BeanFactory creation of BeanDefinition instances	4
Bean creation	4
Enter GraalVM.	4
Where to start	4
The Spring component model.	8
How they run	8
Event listeners	8
FactoryBeans: A reusable recipe for object creation	10
XML configuration	13
Scopes	15
Qualifiers	18
Configuration properties	21
Detecting code is in a native image.	22
Application migration	24
Run the AOT code on the JRE	24
Run the Java agent	27
Soft-touch refactorings to enable AOT	28
Processing beans	36
Proxies	37
Code generation	41
Testing.	46
Conclusion.	48
About the author.	48

Introduction

Spring Boot® 3 is here. For a quick rundown of what Spring Framework® 6 includes, [watch this Spring Tips video](#). This white paper takes a look at the details of the brand-new ahead-of-time (AOT) compilation engine in Spring Boot 3.

The Spring Boot 3 AOT engine introduces a new component model designed to provide extra information about a Spring® application to support optimizing the application both on the Java Runtime Environment (JRE) and in a GraalVM native image context. For the 80 percent case, you shouldn't worry about a thing. But, sometimes, you might need to intervene to get the best results.

You might need to intervene because of GraalVM's native image compilation, which produces lightning-quick and super memory-efficient binaries. But to achieve this magic trick, GraalVM's native image compiler does a compile time scan to determine what types are used, what types those types use, and so on, 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 because 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, such as reflection, proxies, Java Native Interface (JNI), and serialization. Somebody or something has to tell GraalVM about these particular use cases.

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 and Netty need to support their use by furnishing this configuration. There's also an Oracle-led effort called the GraalVM reachability repository, which is a central repository of configuration provided for common libraries that don't yet ship their own .json configuration files. 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.

The Spring Boot 3 release introduced a 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 benefits of GraalVM native image compilation. This white paper looks at the new AOT engine, GraalVM, and the increasingly rarer corner cases that require some intervention.

Understanding Spring

To appreciate where Spring Boot 3 is now, it's helpful to understand where it was. Traditional Spring applications have phases they go through when they run. The following provide 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`), and the like. So it reads the various configuration files from all the classes annotated with `@Configuration` 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, such as `@Service`, `@Repository`, `@Controller` and `@RestController`. It also reads in configuration from the Spring classic XML configuration format.

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, and the link. The **BeanDefinition** instances describe everything required to describe an object and get to a valid state so that it can 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, or doing disk I/O. 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, and so on. After this phase, you will have an application ready to serve production traffic. 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 notable extra utilities: a polyglot VM, a native image compiler, and a HotSpot replacement for the just-in-time (JIT) compiler. For the purpose of this white paper, GraalVM refers to the native image compiler.

Where to start

To start, go to the [Spring Initializr](#) and choose Spring Boot 3.0 or later.

Examples in this white paper use Spring Boot 3, Apache Maven, Java 17, and GraalVM 22.3. Java 17 is the new baseline for Spring Framework 6 and Spring Boot 3. If you're using the [SDKMAN utility](#), then you can do the following to get the same version of GraalVM used for this white paper.

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

This white paper won't touch on the following dependencies in any great length, but they will help to demonstrate a few concepts:

- **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

The following is 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);
    }

}
```

Nothing will be added to this class. Instead, as new concepts get introduced, new `@Configuration`-annotated classes will be created in sub-packages.

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

}
```

As you can see in Label 1, this example does not use Lombok.

In Label 2, you can see that this example uses Spring Data JDBC.

You will need some SQL schema to use Spring Data JDBC, so add `src/main/resources/schema.sql` for our SQL interactions. The following code shows how you can have Spring Boot create it 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 to indicate it works. You can then turn it into a native image.

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

Once complete, you will find the compiled binary in the target directory, named `aot`. Run it, and you will see that it starts up in no time at all, and that it takes very little memory. Of course, there are different ways to measure memory, but looking at [resident set size](#) is informative. You can use the following script to measure `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"
```

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



```

:: Spring Boot ::                (v3.0.0)

2022-12-06T11:28:22.503-08:00 INFO 16819 --- [main] com.example.aot
: Starting AotApplication using Java 17.0.1 with PID 16819
(/Users/jlong/Desktop/spring-boot-3-aot/target/classes started by jlong in
/Users/jlong/Desktop/spring-boot-3-aot)

```

Figure 1: PID location.

You can then pass the PID as the argument for `rss.sh`.

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

On a 2021 M1 MacBook Pro, this uses just less than 100MB. So if your current JVM application uses 1 gigabyte or less of RAM, you should be able to deploy it for one-tenth of the memory footprint after using this script.

The Spring component model

Spring has 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.

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 will have started in no time and be far smaller than a binary and a process occupying RAM. You can use the `rss.sh` script introduced previously to measure the process's RSS.

Event listeners

Spring has an event bus 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);
    }
}
```

In this example, Label 1 shows the more traditional approach: a bean of type `ApplicationListener<T extends ApplicationEvent>`.

Label 2 shows 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 this example, they demonstrate different ways of consuming Spring application events:

- `WebServerInitializedEvent` tells when the embedded web server has finished initializing.
- `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, and 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 up to each implementor on 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 the following 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");
}

}
```

In this example, Label 1 shows that **AnimalFactoryBean** produces an object of type **Animal**. But which? That depends on the parameters fed into the **FactoryBean**.

Label 2 shows that the client code injects the **Animal**, ignorant of the construction logic.

XML configuration

The new Spring AOT engine works well with the classic XML application configuration. Spring ingests, or reads in, configuration from various sources, one of which is XML files.

Let's take a quick look at an example, starting with 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>
```

In this example, Label 1 shows that this `<bean/>` element defines a bean of type `MessageProducer`.

Label 2 shows that 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);
}

}
```

In this example, Label 1 shows the `XmlConfiguration` is just another `@Configuration`-annotated class as before, but this time with an import directive. `@ImportResource("app.xml")` tells Spring to load the beans defined in the XML configuration file as beans in the `BeanFactory`.

Label 2 shows that the `MessageProducer` supplies the message.

Label 3 shows that the `XmlLoggingApplicationListener` uses the `MessageProducer` to print out a message.

Compile and run the application as a native image and you'll see the output—the message is ...—which will only work if Spring can 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`. The following 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`.

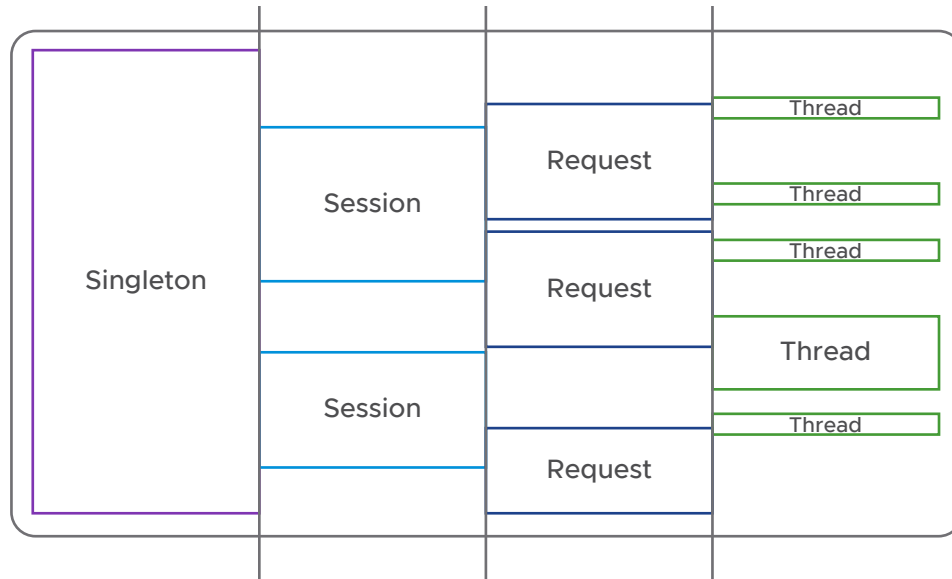


Figure 2: Beans in the application context.

This mechanism is pluggable, so implementors can also provide their scopes. Spring does this across the portfolio in projects, such as Spring Web Flow and Spring Batch. You also see it in third-party projects, such as 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 underneath 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();  
    }  
  
}
```

In this example, Label 1 shows that this bean is a request scoped, so it'll be created anew for each incoming HTTP request.

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

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

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 you qualify the bean you would like.

Suppose you are trying to build two applications with competing mobile phone marketplace implementations, such as the Apple App Store and Google Play.

You might model them with an interface; in this example, it is called **MobileMarketplace**. In this example, there are two implementations of that interface, with the `@Qualifier` annotation on both the bean itself and the place where 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 {

}
```

In this example, Label 1 shows that this bean uses the Android implementation.

Label 2 shows that this bean uses the Apple implementation.

Label 3 shows that you can inject **Map<String,T>**, where T is the type you're looking for. Spring will provide a map of bean names to bean instances.

Label 4 shows the creation of meta-annotation.

Label 5 shows the implementation of the interface.

You can directly define a bean's qualifier using the **@Qualifier** annotation and inject it into a particular site using meta-annotation. Or vice versa, or both. It just works in Spring.

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

Configuration properties

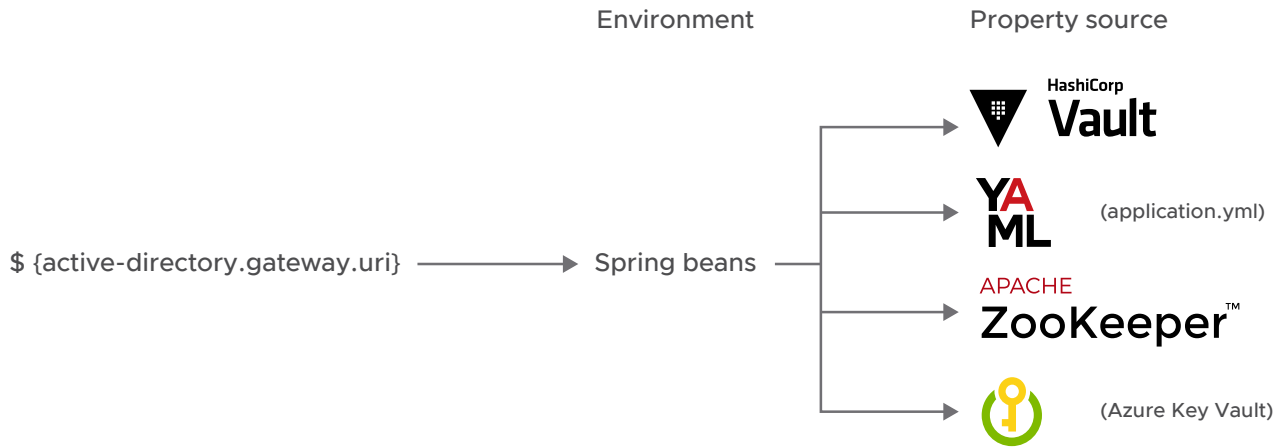


Figure 3: Environment configurations in Spring.

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) {
    }
```

In this example, Label 1 shows the injection of a Java object called **DemoProperties** to which properties starting with “bootiful” are bound.

Label 2 shows that 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 code is in a native image

Sometimes, you will want to know when your code runs in a native image context. Knowing this is helpful because, as good as the Spring AOT engine is, it can't be made to 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 you can use: `org.graalvm.native image.imagecode`. This check-in method is encapsulated in Spring Framework: `NativeDetector.inNativeImage()`.

Here's an example.

```
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 this script on the JRE, and it will return **false**. Run it in a GraalVM native image, and it will return **true**.

On first instinct, you might want to wrap this property in a `Condition` object and then use `@Conditional` to call that condition to make beans available in the Spring `BeanFactory` conditionally. But that doesn't work.

Recall the previous discussion on the phases of a Spring Boot application. It starts up, ingests all the configuration, and 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 plug-in 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.

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, you should avoid using profiles if you intend to create a GraalVM native image. There are some `@Conditional` annotations you should avoid, too.

The Spring AOT engine evaluates the `@Conditional` annotations at compile time. Some invariant conditions are the same at compilation time and runtime, such as `@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, such as 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. The 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 previously, where you will 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 compile times of 30 seconds or longer 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 figuring 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. However, there are some 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 is the default mode and works exactly as it always has
- 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. 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 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 you run the normal code.


```
...

import org.springframework.boot.context.properties.ConfigurationPropertiesBindingPostProcessor__
BeanDefinitions;

import org.springframework.boot.sql.init.dependency.DatabaseInitializationDependencyConfigurer__
BeanDefinitions;

import org.springframework.boot.validation.beanvalidation.MethodValidationExcludeFilter__
BeanDefinitions;

import org.springframework.boot.web.server.ErrorPageRegistrarBeanPostProcessor__BeanDefinitions;
import org.springframework.boot.web.server.WebServerFactoryCustomizerBeanPostProcessor__
BeanDefinitions;

import org.springframework.context.event.DefaultEventListenerFactory__BeanDefinitions;
import org.springframework.context.event.EventListenerMethodProcessor__BeanDefinitions;

import org.springframework.data.repository.core.support.PropertiesBasedNamedQueries__
BeanDefinitions;

import org.springframework.data.repository.core.support.RepositoryComposition__BeanDefinitions;
import org.springframework.data.web.config.ProjectingArgumentResolverRegistrar__BeanDefinitions;
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__
BeanDefinitions;

import org.springframework.web.servlet.config.annotation.WebMvcConfigurationSupport__
BeanDefinitions;

/**
 * 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.  
internalEventListenerProcessor", EventListenerMethodProcessor__BeanDefinitions.  
getInternalEventListenerProcessorBeanDefinition());  
  
beanFactory.registerBeanDefinition("org.springframework.context.event.  
internalEventListenerFactory", DefaultEventListenerFactory__BeanDefinitions.  
getInternalEventListenerFactoryBeanDefinition());  
  
beanFactory.registerBeanDefinition("aotApplication", AotApplication__BeanDefinitions.  
getAotApplicationBeanDefinition());  
  
beanFactory.registerBeanDefinition("basicsConfiguration", BasicsConfiguration__  
BeanDefinitions.getBasicsConfigurationBeanDefinition());  
  
beanFactory.registerBeanDefinition("bfppConfiguration", BfppConfiguration__BeanDefinitions.  
getBfppConfigurationBeanDefinition());  
  
beanFactory.registerBeanDefinition("compilationEndpointConfiguration",  
CompilationEndpointConfiguration__BeanDefinitions.  
getCompilationEndpointConfigurationBeanDefinition());  
  
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());  
  
...
```

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

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

You can also configure the Spring Boot build plug-in to run with that switch active by default.

```
<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 will 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 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, and the like 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 them to tell you what you will need to contribute hints for. The Java agent dumps out everything, including some things that the Spring AOT engine will do automatically for you, so you don't need to reproduce. 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 have 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.

Configure the Spring Boot Maven build plug-in 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.

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

Inspect the `target/native-image/` directory. You will find that the Java agent has written a `reflect-config.json` file, which tells you something will reflect upon the `Person` type at runtime. You 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. You could perhaps run it while running your tests. You want to ensure that you capture every scenario that might require configuration. Be sure to go through all the `.json` configuration files and note every instance where one of the types created, such as `Person`, is present.

At this point, disable the Java agent, commenting it out in the Maven `pom.xml`. It has 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 without interventions.

Here's the completed program, with code to support the Spring AOT engine's work turning your 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());
        }
    }
}

```

Label 1 shows that this program simply reads in CSV data (in the least production-worthy way possible) and maps the rows into Person instances.

The Jackson JSON marshaling library will need to reflect on those Person instances. Label 2 shows the use of the Spring `@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 a `@Configuration` class.

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

As an alternative to using the `@RegisterReflectionForBinding` annotation, Label 4 shows you can contribute a hint for reflective access for that type.

You will need to create a file with sample data called `src/main/resources/data.csv`. Here's an example.

```
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.

For this example to work, or not work, in a GraalVM native image, you will need to comment out the lines that 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 is related to being unable to load the `.csv` file. So, restore the `@ImportRuntimeHints` annotation. Compile and run that as a GraalVM native image. The next error is 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 only used Jackson to print out some objects? 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 such as this where Spring will just work. Spring will still need a custom hint for the `.csv` file.

Processing the BeanFactory

Spring makes it simple 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 you access the **BeanFactory** and manipulate the **BeanDefinition** instances. You 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 a microservices framework, its sole function was to make it easy for Spring Boot applications deployed to a platform as a service (PaaS), such as Cloud Foundry or Heroku, to connect to managed infrastructure, such as a database or a message queue. Spring Cloud did this by analyzing the **BeanFactory**, identifying infrastructure-related beans (such as **javax.sql.DataSource**), and replacing them with a new bean of an identical interface but with a connection that points 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 **BeanFactoryPostProcessor** implementations.

The **BeanFactoryPostProcessor** enables you to see everything in the **BeanFactory** 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 interfaces. Remember, Spring hasn't created any of the beans at this point. So, while you will be able to call **BeanFactory#getBean(String)**, it will force Spring to initialize the object, calling the constructor, the methods, and so on before the bean is ready. Don't do this; it will cause mistakes.

Let's look at an example of writing a **BeanFactoryPostProcessor** that analyzes and programmatically registers a new **BeanDefinition** in the application context. (You will probably never need to do this, but it is hopefully illustrative.) Use a subclass of **BeanFactoryPostProcessor** called **BeanDefinitionRegistryPostProcessor**, which gives you a **BeanFactory** downcast to a specific subtype that supports programmatically registering new beans. The trouble is that this bean will require some reflection using the Jackson JSON API, so you will need to 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").getBeanDefinition());
    }

}

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;
    }
}

```

In this example, you work with the same bean at compile time and runtime. Label 1 shows the creation of a variable with the name of the bean for subsequent access from both interface implementations at both phases (AOT and runtime).

As shown with Label 2, note the use of the static keyword with both `BeanDefinition` 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 any lifecycle issues. Avoid depending on anything from the `BeanFactory`. Also, don't force Spring to construct the `@Configuration` class containing the `@Bean` registration methods.

Label 3 shows that the `Listener` class is an `ApplicationListener` that, when run, will iterate over a collection of mock data transfer objects (DTOs) and print them out using Jackson for JSON serialization. This serialization involves reflection for which you will need to furnish hints.

Label 4 shows that the first callback, the `ListenerBeanFactoryPostProcessor`, programmatically registers a new `BeanDefinition` of type `Listener` if a bean of the name you have 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, you mutate the `BeanFactory`.

Label 5 shows that the Spring AOT engine will involve the `ListenerBeanFactoryInitializationAotProcessor` during the compilation phase so it can 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 to write code when manipulating the Spring application context. Don't write code in terms of the `BeanFactory` if you 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 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 and after initialization.

Working on a bean-by-bean basis can be very powerful. If you want to work with actual, live-fire beans, you can use the Spring `BeanPostProcessor`. This interface puts you in a position to act on and transform objects before they're 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. Objects can be of any shape, such as objects that have a certain marker interface or a certain annotation. They can be whatever you want and whatever you could discover.

Let's look at an example that creates proxies for beans with an annotation, `@Logged`, logging out any method invocations.

You can implement this proxy with the Spring `ProxyFactory`, which simplifies using 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 and the like. There is a way to decorate an existing object with cross-cutting concerns, such as starting and committing a transaction before and after a method invocation. Spring uses them for things such as `@Transactional`, `@Scheduled`, `@Async`, `@Authorized` and countless more.

There are two types of proxies: CGLIB and JDK. Proxies make it simple 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, interface means 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: This is probably 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 such as `FooService`, `DefaultFooService`, `FooService` and `FooServiceImpl`. Now, 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](#) to dynamically generate the code to subclass an existing type. The constraint is that the type is subclassable. So, beware of things such as `final` and `sealed`.

The Spring 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 actions that you will need to account for at compile time in a GraalVM native image context.

Proxies

Spring can do a lot of this for you. Let's take a look at a simple proxy example of the logs information whenever somebody invokes a method annotated called `@Logged`, which you will create. The annotation is decorative. You will layer 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);
    }

}
```

Label 1 shows the usual trying things out when the application starts up.

Label 2 shows the creation of this proxy in a subclass of `BeanPostProcessor` called `SmartInstantiationAwareBeanPostProcessor`. The Spring AOT engine invokes the `SmartInstantiationAwareBeanPostProcessor#determineBeanType` to determine what the bean should be. The Spring 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.

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

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

Label 5 shows that `BeanPostProcessor` implementations visit every bean in the `BeanFactory`, so you 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.

Label 6 shows that the `SmartInstantiationAwareBeanPostProcessor#determineBeanType` method does almost all the work of creating a proxy but stops short of actually creating the proxy. For this example, you only need to know what the class would be if you did create a proxy. This information then gets fed into the AOT engine.

Label 7 shows that 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.

This example uses the cohesive design of the Spring AOT engine to transparently create proxies that just work.

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

Code generation

Before going much further, let's 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 the purview of Spring. You will need to furnish the appropriate hints for GraalVM to know 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 the Spring AOT engine. 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 of code generation that goes even further. Register a new bean during compilation, overwriting one that is already present, so that you can capture compile-time information and put it in the endpoint. This Actuator endpoint, the **CompilationEndpoint**, will capture information about the ambient state of the build at compile-time, such as 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("postProcessCompilationEn
dpoint", 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,
                    Instant.class, System.currentTimeMillis() + "L",
                    File.class,
                    new File(".").getAbsolutePath()

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

```

Label 1 shows the registration of an empty, no-op version of the `CompilationEndpoint`.

Label 2 shows the post-process of the bean at compile time using an implementation of a `BeanRegistrationAotProcessor`.

Label 3 shows testing things out when the application starts.

Label 4 shows how to register a Spring Boot Actuator endpoint. 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 there are two constructors: one for the no-op case and another for information about the build, such as the time and directory of the compilation.

Label 5 shows that, like the `BeanPostProcessor`, `BeanRegistrationAotProcessor` implementations work on a bean-by-bean basis, getting a chance to inspect and change them. 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 you to inspect the reflective metadata associated with the bean, and the `CompilationEndpoint` is the bean created earlier. If you don't use the Spring AOT engine, then the bean looked at earlier is the only bean you will ever get. However, if you use the AOT engine, this method will return a new `CompilationEndpoint`, initialized with values during compilation.

Label 6 shows the use of Java Poet to create a new instance of `CompilationEndpoint` with real values initialized and captured during the AOT phase itself.

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 you have 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;
    }
}
```

Testing

Fixing an issue is critical to success with technology. But how do you ensure it stays fixed going forward? Here is where the Spring rich testing support comes in. It, too, has been updated to work in a GraalVM native image context.

Testing is incredibly important. The Spring team has done a lot 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 will get two output artifacts: one for tests and another for the production code. Please note that producing the test code as a native image doubles the interminable compile time, which could be a few minutes. When the compilation finishes, there will 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);
    }
}

```

Label 1 shows the creation of a Spring ApplicationContext and then using it to inject references to beans, such as the CustomerRepository created previously when discussing AOT fundamentals.

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

It's good that the basics work, too. So does it mean that you can run the usual test harness against business logic and confirm to whatever extent you would like that things work as they did before? And 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 work 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")).accepts(hints);

    }

}
```

Label 1 shows the creation of an instance of the implementation on the `RuntimeHintsRegistrar` interface.

Label 2 shows calling `RuntimeHintsRegistrar#registerHints` on the implementation, passing in mock instances of `RuntimeHints` and the current `ClassLoader`.

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

Conclusion

Build times have come a long way in Spring. In 2020, a build in Spring Native (the predecessor to the Spring Framework 6 AOT engine) took 10 minutes and only worked for the smallest and most specific scenarios. Now, you can get a Spring app working in a GraalVM native image context in less than 1 minute.

The performance and memory usage implications are hard to ignore. As the community embraces the newfound AOT engine support in Spring, the move to the AOT engine will become easier to obtain.

The goal of this white paper is to make it easier for application and framework developers to leverage Spring Boot 3 and Spring Framework 6. For guidance on implementing into your code, the Spring Boot codebase provides many examples for types, such as `RuntimeHintsRegistrar`, `BeanRegistrationAotProcessor`, and `BeanFactoryInitializationAotProcessor`.

The Spring AOT engine already has many implementations, such as Spring Security, Spring Data, Spring Cloud, Spring Shell, Spring Integration, and Spring Batch. And things are moving quickly. There is work already or nearly completed for various open source projects, such as Axon, Vaadin, Hilla Framework, MyBatis, JHipster, the official Kubernetes Java client, JobRunr, and many others.

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, the Spring AOT engine helps you introduce new possibilities, such as serverless, embedded and infrastructure, for which Spring might not have been considered ideal in the past.

To get started, go to the [Spring Initializr](#) and generate your next value-producing, production-bound Spring Boot application with GraalVM native image support.

About the author

Josh Long ([@starbuxman](#)) has been a Spring Developer Advocate since 2010. He is a Java Champion, author of six books (including [Reactive Spring](#)), creator of numerous bestselling video trainings (including [Building Microservices with Spring Boot Livelessons](#) with Spring Boot co-founder Phil Webb), an open source contributor (Spring Boot, Spring Integration, Spring Cloud, Activiti and Vaadin, etc.), a YouTuber ([Coffee + Software with Josh Long](#) and his [Spring Tips series](#)), and a podcaster ([A Bootiful Podcast](#)).

