

# Spring Graal Native 0.6.0.RELEASE reference guide

Andy Clement, Sébastien Deleuze, Filip Hanik, Dave Syer, Esteban Ginez, Jay  
Bryant

# Table of Contents

1. What about Spring?	2
2. Samples	3
3. Guides	4
3.1. Using the feature	4
3.1.1. Download and install GraalVM	4
3.1.2. Set up the sample project	4
Update the pom.xml file	4
Add the Maven plugin	5
Add the repository for spring-graal-native	6
Add the appropriate dependencies	6
Set the start-class element	7
Update the source code	7
3.1.3. Build the application	8
3.1.4. Run the application	8
3.1.5. Summary	9
3.2. Using the GraalVM agent	9
3.2.1. Download and install GraalVM	9
3.2.2. Setup the sample project	10
Update the pom.xml file	10
Add the Maven plugin	10
Add the repository for spring-graal-native	11
Add the feature and appropriate configuration dependencies	12
Set the start-class element	12
Update the source code	13
3.2.3. Create a location for the generated configuration	13
3.2.4. Run the application with the agent	14
3.2.5. What about initialization configuration?	14
3.2.6. Build the application	15
Run the application	15
3.2.7. Summary	16
3.3. Using the hybrid feature + agent mode	17
3.3.1. Download and install GraalVM	17
3.3.2. Setup the sample project	17
3.3.3. Update the pom.xml file	17
Add the Maven plugin	18
Add the repository for spring-graal-native	20
Add the feature and appropriate configuration dependencies	20
Set the start-class element	21

Update the source code .....	21
Create a location for the generated configuration .....	21
Run the application with the agent .....	22
Run the application .....	23
Summary .....	23
4. Options .....	24
4.1. GraalVM options .....	24
4.1.1. Options enabled by default .....	24
4.1.2. Options recommended by default .....	24
4.1.3. Other options .....	24
4.2. Feature options .....	24
4.3. Optional options .....	25
5. Troubleshooting .....	26
5.1. native-image is failing .....	26
5.1.1. Out of memory error when building the native image .....	26
5.1.2. You get a weird crash .....	26
5.2. The built image does not run .....	27
5.2.1. Missing resource bundles .....	27
5.2.2. Getting ClassNotFoundException .....	28
5.3. Where has my logging gone? .....	29
5.3.1. No access hint found for import selector: XXX .....	29
5.4. Diagnosing issues with the feature .....	29
6. Extension guide .....	33
6.1. Extending the feature .....	33
6.2. Hints .....	33
6.3. Triggered .....	33
6.4. What do hints look like? .....	34
6.5. Optimizing which hints are acted on .....	34
6.6. Structure of the spring-boot-graal-configuration module .....	34
6.7. Contributing new hints .....	35
6.8. Is this the way? .....	35
7. Contact us .....	37

This guide walks you through the various options for building GraalVM native images for your Spring Boot applications.

This involves compiling your application as normal and then running an extra native-image step that builds a self-contained platform-specific executable for that application. The executable should have a more predictable memory profile as well as fast startup.

The key complexity in building a native image is that the image building process needs to be informed about what your application is going to do at runtime. The build process attempts to construct the most optimal image possible and therefore tries to exclude anything not strictly required. However, it sometimes needs a few hints so that it can ensure it includes in the image all the necessary code and data to satisfy the application runtime needs.

In particular, the following concerns apply:

- If the application is going to reflect at runtime on some type, field, method or constructor, the native image must include the data to satisfy those reflection requests.
- If the application is going to load resources (such as `application.properties`), they must be included in the image.
- If the application is going to try to use proxies at runtime, these must be generated up front during the image-build process and included in the image.
- There can be a benefit to initializing classes when the image is built, as opposed to when the resultant executable starts up. The image build process needs to be told which classes to initialize at build time versus at run time.

There are two primary ways to communicate this information to the build process:

- Through JSON files on the classpath.
- Through a GraalVM feature that participates in the image-build process and, after analyzing the application being compiled, calls APIs to register information about reflection, resources, proxies, and so on.

Some projects are starting to include the necessary JSON files in their default distributions (for example, Netty).

Crafting these JSON files can be tricky, but they can actually be computed by using an agent that comes with GraalVM. Run the application and exercise all relevant code paths (possibly through tests) with the agent attached, and it produces most of the configuration. The only part it does not compute is the initialization hints.

It is worth mentioning that there are limitations on what GraalVM supports with respect to native images. If your application is doing any of these, it may not be possible to build a native image. The limitations are discussed [here](#).

# Chapter 1. What about Spring?

The `spring-graal-native` project represents a work-in-progress attempt to implement a GraalVM feature. This feature examines the Spring Boot application being built and tells the image-build process about it. The feature understands how Spring is implemented and, so, from examining what auto-configuration is defined on the classpath (which components are defined and so on), it knows what to grant reflective access to and what to generate proxies for.

The feature is a work in progress: There are numerous sample projects for which it works, as can be seen in the samples folder within the project ([you can read more about the samples here](#)), but does that mean it will work on the first thing you try it on? That is not guaranteed. When things do not work, do not panic, there are multiple ways to proceed, and this reference guide covers those.

See the [feature guide](#) for more information on getting started with the Spring feature, including a full walkthrough with an example.

It is also possible to use the GraalVM supplied agent to determine the configuration data for your Spring application. See [the agent guide](#) for that approach. Be aware that there are still bugs in the agent, though, and it can “miss things”. Resolving these misses can be tricky. We are working with the GraalVM team to ensure issues are fixed.

There is even a hybrid model where the feature and the agent are used together. Neither the feature nor the agent are perfect, but sometimes they can nicely complement each other’s deficiencies. See [the hybrid guide](#).

# Chapter 2. Samples

There are numerous samples in the `spring-graal-native-samples` subfolder of the root project. These show the feature in use with different technologies.

The samples do not all currently use Maven. Instead, they are built for us to play around with a little more easily. Each includes a `build.sh` script which (Java) builds the app, calls the `compile.sh` script to run the `native-image` process, and runs a `verify.sh` script to check whether the built image works.

Our build process for the feature currently runs `./build.sh` in the root of the project and then `./build-samples.sh` to build all the samples and verify everything is working. Running `native-image` instances takes a long time and eats RAM.

The samples show the wide variety of tech that is working fine: Tomcat, Netty, Thymeleaf, JPA, and others. The pet clinic sample brings multiple technologies together in one application.

`jafu` and `jafu-webmvc` samples are based on the [Spring Fu experimental project](#) and are designed to demonstrate how leveraging Spring infrastructure in a functional way ends up to smaller native images that consume less memory.

If you are starting to build your first Spring Boot application, we recommend you follow one of the guides.

# Chapter 3. Guides

## 3.1. Using the feature

This section walks through how to try building a native image for a Spring Boot project. This is a practical guide, so we perform this for real on the [REST service getting started guide](#).

### 3.1.1. Download and install GraalVM

Although the `native-image` build is invoked through Maven, the agent is shipped with GraalVM and we need to install it by following [these instructions](#). Notice Java 8 and Java 11 versions are available. Although either should work, we have had more issues with the Java 11 one, so playing it safe would be selecting the Java 8 version.

### 3.1.2. Set up the sample project

Like the instructions for using the feature, here we use the getting started REST service guide. This is the sample project we trace with the agent and then build into a native image. The following commands install the REST service guide:

```
git clone https://github.com/spring-guides/gs-rest-service
cd gs-rest-service/complete
```

You may already be ready to go with your own project.

#### Update the `pom.xml` file



Ensure that the project uses a supported version of Spring Boot.

If needed, upgrade the project to Spring Boot 2.3.0.M3 (it may work with 2.2.X versions of Boot if upgrading is tricky) and add the following repositories to the `pom.xml` file:

```
<pluginRepositories>
  <!-- ... -->
  <pluginRepository>
    <id>spring-milestone</id>
    <name>Spring milestone</name>
    <url>https://repo.spring.io/milestone</url>
  </pluginRepository>
</pluginRepositories>
```

And

```

<repositories>
  <!-- ... -->
  <repository>
    <id>spring-milestone</id>
    <name>Spring milestone</name>
    <url>https://repo.spring.io/milestone</url>
  </repository>
</repositories>

```

## Add the Maven plugin

GraalVM provides a [Maven plugin](#). Paste the following XML into the `pom.xml` file (we use it later to invoke the native image build):

```

<profiles>
  <profile>
    <id>graal</id>
    <build>
      <plugins>
        <plugin>
          <groupId>org.graalvm.nativeimage</groupId>
          <artifactId>native-image-maven-plugin</artifactId>
          <version>20.0.0</version>
          <configuration>
            <buildArgs>-Dspring.graal.remove-unused-autoconfig=true --no-fallback
--allow-incomplete-classpath --report-unsupported-elements-at-runtime
-H:+ReportExceptionStackTraces --no-server</buildArgs>
          </configuration>
          <executions>
            <execution>
              <goals>
                <goal>native-image</goal>
              </goals>
              <phase>package</phase>
            </execution>
          </executions>
        </plugin>
        <plugin>
          <groupId>org.springframework.boot</groupId>
          <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
      </plugins>
    </build>
  </profile>
</profiles>

```



The important part is the `<buildArgs>..</buildArgs>` block, which shows the options we pass to the `native-image` operation and the `spring-graal-native` feature. Those prefixed `-D` are aimed at the feature.



Notice the `--no-server` option. Like a Gradle daemon, the server here is supposed to help accelerate subsequent builds. Unfortunately, there is an [issue](#) where the server causes different results to come out of the compilation. In particular, we have seen logging disappear if the server is used to aid compilation. Hence, using the server to aid compilation is turned off here.

The `-Dspring.graal.remove-unused-autoconfig=true` is an option we can use to evaluate some of the Spring Boot conditions at image-build time. One example of such an option is `@ConditionalOnClass`. Because `native-image` runs at a point when the full classpath is known, we can know for certain if a class is present. If it is not present, the auto configuration conditional on that class can be discarded and not have an impact on the image startup. More details on options are discussed [here](#).

### Add the repository for `spring-graal-native`

If necessary, add the repository for the `spring-graal-native` dependency, as follows:

```
<repositories>
  <!-- ... -->
  <repository>
    <id>spring-milestone</id>
    <name>Spring milestone</name>
    <url>https://repo.spring.io/milestone</url>
  </repository>
</repositories>
```

### Add the appropriate dependencies

The following listing shows the dependencies to add:

```

<dependencies>
  <!-- ... -->
  <dependency>
    <groupId>org.springframework.experimental</groupId>
    <artifactId>spring-graal-native</artifactId>
    <version>0.6.0.RELEASE</version>
  </dependency>

  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context-indexer</artifactId>
  </dependency>
</dependencies>

```

- What is **spring-graal-native**? The **spring-graal-native** dependency brings together several components. It includes the GraalVM feature implementation. It includes the temporary substitutions (a GraalVM term) for patching around issues with some classes whilst we wait for more correct fixes in these classes. The feature behaviour is actually driven by a set of annotations that encapsulate boot knowledge that is non obvious from a high level static analysis of the code, for example a particular ImportSelector may required reflective access to a type. This knowledge is also included in the **spring-graal-native** dependency.
- The **spring-context-indexer** has been in Spring for a while. In a native image, all notion of classpath is lost, so it is not possible to explore the classpath to find components at runtime. The indexer actually produces a list of components at Java compile time and captures it in a **spring.components** file in the built application. If Spring starts and finds this file, it uses it instead of attempting to explore the classpath. The indexer can be used for this whether building a native image or just running your application as a standard Java application.

### Set the **start-class** element

The native image build needs to know the entry point to your application. It does consult a few places to find it, but, in our sample, we set it in the properties section of the **pom.xml**, as follows:

```

<start-class>com.example.restservice.RestServiceApplication</start-class>

```

### Update the source code

In this sample, are no changes need to be made. However, in some Boot applications, it may be necessary to make some tweaks to ensure they are not doing anything that is not supported by GraalVM native images.

### Proxies

The only kind of proxy allowed with native images is a JDK proxy. It is not possible to use CGLIB or

some other kind of generated proxy. Boot 2.2, added the option to avoid creating these kinds of native image incompatible proxies for configuration class contents and this happens to suit native image compilation. The enhancement in question is discussed [here](#). Basically, applications need to switch to using `proxyBeanMethods=false` in their configuration annotations. The framework code has already all moved to this model. The following example comes from the `webflux-netty` sample:

```
@SpringBootApplication(proxyBeanMethods = false)
public class DemoApplication {

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

    @RestController
    class Foo {

        @GetMapping("/")
        public String greet() {
            return "hi!";
        }
    }
}
```

### 3.1.3. Build the application

```
mvn -Pgraal clean package
```

Did it build cleanly? If so, the resultant executable is in the target folder named after the `start-class` (in this case, `com.example.restservice.RestServiceApplication`).

Did it fail? See the [Troubleshooting](#) section.

### 3.1.4. Run the application

To run your application, you need to run the following executable:

```
./target/com.example.restservice.RestServiceApplication

...
Mar 18, 2020 3:26:16 PM
org.springframework.boot.web.embedded.tomcat.TomcatWebServer start
INFO: Tomcat started on port(s): 8080 (http) with context path ''
Mar 18, 2020 3:26:16 PM org.springframework.boot.StartupInfoLogger logStarted
INFO: Started RestServiceApplication in 0.084 seconds (JVM running for 0.087)
```

The startup time is <100ms, compared ~1500ms when starting the fat jar.

Did your application run successfully? If so, good. If not, see the [Troubleshooting](#) page.

### 3.1.5. Summary

Hopefully, this section has given you a taste of the process of building native images. There is much more coming to optimize Spring in all areas: smaller images, reduced memory usage, faster native image compilation, and more. We are also working with the Graal team in all the pitfall areas mentioned earlier. Across the board, things should only get better. If you apply these techniques to your own application and have problems, see [Troubleshooting](#).

## 3.2. Using the GraalVM agent

The feature is a work in progress, it doesn't understand all of Spring yet, but perhaps more importantly it doesn't understand what else your app might be doing with various other non-Spring related dependencies. In situations like this it can be useful to try the agent which may catch 'other things' your application is doing. You can attach the GraalVM agent to an application and, during the course of the application's execution, it produce the necessary configuration files for a subsequent **native-image** build. As the application runs, it observes the types being reflectively accessed, the resources being loaded, and the proxies being requested, and it captures this information in **.json** files.

This section walks through using the agent to compute the configuration data and then build a **native-image** using it. A real application is used to show the process, but the same techniques should apply to any application.



Currently, there are agent bugs, and we cover those, including how they manifest and how to do deal with them, as we go.

### 3.2.1. Download and install GraalVM

Although the **native-image** build is going to be invoked through Maven, the agent is shipped with GraalVM and we need to install it by following [these instructions](#). Notice that Java 8 and Java 11 versions are available. Although either should work, we have had more issues with the Java 11 one, so playing it safe would be selecting the Java 8 version.

### 3.2.2. Setup the sample project

Like the instructions for using the feature, here we use the getting started REST service guide. This is the sample project we trace with the agent and then build into a **native-image**. The following commands install the REST service:

```
git clone https://github.com/spring-guides/gs-rest-service
cd gs-rest-service/complete
```

#### Update the **pom.xml** file



Ensure that the project uses a supported version of Spring Boot.

If needed, upgrade the project to Spring Boot 2.3.0.M3 and add the following repositories to the **pom.xml** file:

```
<pluginRepositories>
  <!-- ... -->
  <pluginRepository>
    <id>spring-milestone</id>
    <name>Spring milestone</name>
    <url>https://repo.spring.io/milestone</url>
  </pluginRepository>
</pluginRepositories>
```

And

```
<repositories>
  <!-- ... -->
  <repository>
    <id>spring-milestone</id>
    <name>Spring milestone</name>
    <url>https://repo.spring.io/milestone</url>
  </repository>
</repositories>
```

#### Add the Maven plugin

GraalVM provides a **Maven plugin** that we are going to bring in here. Paste the following XML into the **pom.xml** file (we use it later to invoke the native image build):

```

<profiles>
  <profile>
    <id>graal</id>
    <build>
      <plugins>
        <plugin>
          <groupId>org.graalvm.nativeimage</groupId>
          <artifactId>native-image-maven-plugin</artifactId>
          <version>20.0.0</version>
          <configuration>
            <buildArgs>-Dspring.graal.mode=initialization-only --no-fallback
--allow-incomplete-classpath --report-unsupported-elements-at-runtime
-H:+ReportExceptionStackTraces --no-server</buildArgs>
          </configuration>
          <executions>
            <execution>
              <goals>
                <goal>native-image</goal>
              </goals>
              <phase>package</phase>
            </execution>
          </executions>
        </plugin>
        <plugin>
          <groupId>org.springframework.boot</groupId>
          <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
      </plugins>
    </build>
  </profile>
</profiles>

```

The important part is the `<buildArgs>..</buildArgs>` block that shows the options we pass to the `native-image` operation and the `spring-graal-native` feature. Those prefixed `-D` are aimed at the feature.



Notice the `--no-server` option. Like a Gradle daemon, the server here is supposed to help accelerate subsequent builds. Unfortunately, there is an [issue](#) where the server causes different results to come out of the compilation. In particular, we have seen logging disappear the server is used to aid compilation. Hence, using the server to aid compilation is turned off here.

### Add the repository for `spring-graal-native`

If necessary, add the repository for the `spring-graal-native` dependency, as follows:

```

<repositories>
  <!-- ... -->
  <repository>
    <id>spring-milestone</id>
    <name>Spring milestone</name>
    <url>https://repo.spring.io/milestone</url>
  </repository>
</repositories>

```

## Add the feature and appropriate configuration dependencies

The following listing adds the needed dependencies:

```

<dependencies>
  <!-- ... -->
  <dependency>
    <groupId>org.springframework.experimental</groupId>
    <artifactId>spring-graal-native</artifactId>
    <version>0.6.0.RELEASE</version>
  </dependency>

  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context-indexer</artifactId>
  </dependency>
</dependencies>

```

- Why do we need `spring-graal-native`? This is discussed in more detail later in this section. Basically, the feature is being used in a lightweight mode here where it is not providing all the configuration. Rather, it provides only the initialization configuration. That is because the agent does not compute this information.
- The `spring-context-indexer` has been in Spring for a while. In a `native-image`, all notion of classpath is lost, so it is not possible to explore the classpath to find components at runtime. The indexer actually produces a list of components at Java compile time and captures it in a `spring.components` file in the built application. If Spring starts and finds this file, it uses it instead of attempting to explore the classpath. The indexer can be used for this whether building a native image or running your application as a standard Java application.

## Set the `start-class` element

The native image build needs to know the entry point to your application. It consults a few places to find it. However, in our sample, we set it in the properties section of the `pom.xml` file:

```
<start-class>com.example.restservice.RestServiceApplication</start-class>
```

## Update the source code

In the case of this sample, no changes need to be made. However, in some Boot applications, you may need to make some tweaks to ensure they are not doing anything that is not supported by GraalVM native images.

### Proxies

The only kind of proxy allowed with native images is a JDK proxy. It is not possible to use CGLIB or some other kind of generated proxy. Boot 2.2 added the option to avoid creating these kinds of **native-image** incompatible proxies for configuration class contents, and this happens to suit **native-image** compilation. The enhancement in question is discussed [here](#). Basically, applications need to switch to using **proxyBeanMethods=false** in their configuration annotations. The framework code has already all moved to this model. The following example is from the **webflux-netty** sample:

```
@SpringBootApplication(proxyBeanMethods = false)
public class DemoApplication {

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

    @RestController
    class Foo {

        @GetMapping("/")
        public String greet() {
            return "hi!";
        }
    }
}
```

### 3.2.3. Create a location for the generated configuration

This can be anywhere, but that location needs to be under a location of **META-INF/native-image** and on the classpath so that the native image operation automatically picks it up. If we want to keep this configuration around, we can generate it straight into the project (and perhaps store it in version control), as follows:



```
mkdir -p src/main/resources/META-INF/native-image
```



The “proper” location is perhaps a `<groupid>/<artifactid>` location below native-image, but we keep it simple here for now.

### 3.2.4. Run the application with the agent

The following commands run the application with the agent:

```
mvn clean package
java -agentlib:native-image-agent=config-output-dir=src/main/resources/META-
-INF/native-image \
  -jar target/rest-service-0.0.1-SNAPSHOT.jar
```

It runs as normal.



While it is up to you to make sure you exercise any codepaths you want to ensure are covered by the native image that will be built, exercising those paths may cause extra reflection access or resource loading and in other situations.

Shutdown the app.

Notice the files that now exist in the folder:

```
ls -l src/main/resources/META-INF/native-image
total 256
-rw-r--r--  1 foo  bar      4 18 Mar 18:59 jni-config.json
-rw-r--r--  1 foo  bar  1057 18 Mar 18:59 proxy-config.json
-rw-r--r--  1 foo  bar 98309 18 Mar 18:59 reflect-config.json
-rw-r--r--  1 foo  bar 17830 18 Mar 18:59 resource-config.json
```

### 3.2.5. What about initialization configuration?

The agent does not compute which types need build-time versus run-time initialization. For this reason, the `spring-graal-native` feature is still going to be used, but only to provide that initialization information. All the reflection, resource, and proxy configuration is going to be what we generated.



The feature is also providing a couple of substitutions. These are kind of “patches” for classes in the framework or dependent libraries that do not properly support `native-image`. These should be temporary, and the proper solution should be pushed back into the framework or library concerned. You might have to develop substitutions of your own if your dependent libraries are slow to fix themselves for GraalVM `native-image` interoperation.

### 3.2.6. Build the application

The following command builds the application:

```
mvn -Pgraal clean package
```

Did it build cleanly? If so, the resultant executable is in the target folder named after the start-class (in this case, `com.example.restservice.RestServiceApplication`).

Did it fail? See the [Troubleshooting](#) section. As of this writing, this step works.

#### Run the application

Run the following executable to run the application:

```
./target/com.example.restservice.RestServiceApplication
```

It failed. This is a realistic situation right now. You have to work a little harder while the agent is missing things. We do that now to troubleshoot this problem.

You should get the following exceptions when you launch it:

```
Caused by: java.util.MissingResourceException:  
Resource bundle not found javax.servlet.http.LocalStrings.  
Register the resource bundle using the option  
-H:IncludeResourceBundles=javax.servlet.http.LocalStrings.
```

You can tweak the `pom.xml` file to add `-H:IncludeResourceBundles=javax.servlet.http.LocalStrings` to the `<buildArgs>` section as another option.

Recompile. Now it fails with the following error:

```
Caused by: java.util.MissingResourceException:
  Resource bundle not found javax.servlet.LocalStrings.
  Register the resource bundle using the option
    -H:IncludeResourceBundles=javax.servlet.LocalStrings
```

You can add that `-H:IncludeResourceBundles=javax.servlet.LocalStrings` to `pom.xml` `<buildArgs>` and recompile again.

Now it might launch. However, on curling to the endpoint (`curl localhost:8080/greeting`) it shows the following error:

```
Caused by: java.lang.ClassNotFoundException:
  org.apache.catalina.authenticator.jaspic.AuthConfigFactoryImpl
  at
  com.oracle.svm.core.hub.ClassForNameSupport.forName(ClassForNameSupport.java:60)
  ~[na:na]
```

This is what happens when the agent misses a reflective usage. This particular one is [issue 2198](#). It has been fixed but after Graal 20.0. In this situation, we can manually add this entry. To do so, open `src/main/resources/META-INF/native-image/reflect-config.json` and insert the following on line 2 (after the `[` on line 1):

```
{
  "name": "org.apache.catalina.authenticator.jaspic.AuthConfigFactoryImpl",
  "allDeclaredConstructors": true,
  "allDeclaredMethods": true
},
```

It should now work after a rebuild. The startup time is `<100ms`, compared to `~1500ms` when starting the fat jar.

### 3.2.7. Summary

Hopefully, that has given you a taste of the process of building native images. There is much more coming to optimize Spring in all areas: smaller images, reduced memory usage, faster native image compilation, and more. We are also working with the Graal team in all the pitfall areas described earlier. Things across the board should only get better. If you apply these techniques to your own application and have problems, see the [Troubleshooting](#) section.

## 3.3. Using the hybrid feature + agent mode

This section shows the `spring-graal-native` feature and agent being used together.

This section walks through how to build a native image for a Spring Boot project. This is a practical guide, so we perform this for real on the [REST service getting started guide](#). Unlike the [feature](#) or [agent](#) sections, which show how to use those capabilities standalone, this will use them together in a single project. With the feature not being able to easily know everything about your application, and the agent missing things and not understanding Spring applications, it can sometimes be useful to use them both together.

### 3.3.1. Download and install GraalVM

Although the native image build is going to be invoked through Maven, the agent is shipped with GraalVM, and we need to install it by following [these instructions](#). Notice that Java 8 and Java 11 versions are available. Although either should work, we have had more issues with the Java 11 one, so playing it safe would be selecting the Java 8 version.

### 3.3.2. Setup the sample project

Just to get ready, clone the sample project, as follows:

```
git clone https://github.com/spring-guides/gs-rest-service
cd gs-rest-service/complete
```

You may already be ready to go with your own project.

### 3.3.3. Update the `pom.xml` file



Ensure that the project uses a supported version of Spring Boot.

If needed, upgrade the project to Spring Boot 2.3.0.M3 and add the following repositories to the `pom.xml` file:

```
<pluginRepositories>
  <!-- ... -->
  <pluginRepository>
    <id>spring-milestone</id>
    <name>Spring milestone</name>
    <url>https://repo.spring.io/milestone</url>
  </pluginRepository>
</pluginRepositories>
```

And

```
<repositories>
  <!-- ... -->
  <repository>
    <id>spring-milestone</id>
    <name>Spring milestone</name>
    <url>https://repo.spring.io/milestone</url>
  </repository>
</repositories>
```

### Add the Maven plugin

GraalVM provides a Maven plugin that we are going to bring in here. Paste the following XML into the `pom.xml` file (we use it later to invoke the native image build):

```

<profiles>
  <profile>
    <id>graal</id>
    <build>
      <plugins>
        <plugin>
          <groupId>org.graalvm.nativeimage</groupId>
          <artifactId>native-image-maven-plugin</artifactId>
          <version>20.0.0</version>
          <configuration>
            <buildArgs>-Dspring.graal.remove-unused-autoconfig=true --no-fallback
--allow-incomplete-classpath --report-unsupported-elements-at-runtime
-H:+ReportExceptionStackTraces --no-server</buildArgs>
          </configuration>
          <executions>
            <execution>
              <goals>
                <goal>native-image</goal>
              </goals>
              <phase>package</phase>
            </execution>
          </executions>
        </plugin>
        <plugin>
          <groupId>org.springframework.boot</groupId>
          <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
      </plugins>
    </build>
  </profile>
</profiles>

```

The important part is the `<buildArgs>..</buildArgs>` block that shows the options we are passing to the `native-image` operation and the `spring-graal-native` feature. Those prefixed `-D` are aimed at the feature.



Notice the `--no-server` option. Like a Gradle daemon, the server here is supposed to help accelerate subsequent builds. Unfortunately, there is an [issue](#) where the server causes different results to come out of the compilation. In particular, we have seen logging disappear if the server is used to aid compilation. Hence using the server to aid compilation is turned off here.

We can use the `-Dspring.graal.remove-unused-autoconfig=true` option to evaluate some of the Spring Boot conditions at image-build time. For example, it applies to `@ConditionalOnClass`. Because `native-image` runs at a point when the full classpath is known, we can know for certain if a class is around. If it is not around, the auto-configuration conditional on that class can be discarded and not have an impact on the image startup. More details on options are discussed [here](#).

## Add the repository for `spring-graal-native`

If necessary, add the repository for the `spring-graal-native` dependency, as follows:

```
<repositories>
  <!-- ... -->
  <repository>
    <id>spring-milestone</id>
    <name>Spring milestone</name>
    <url>https://repo.spring.io/milestone</url>
  </repository>
</repositories>
```

## Add the feature and appropriate configuration dependencies

```
<dependencies>
  <!-- ... -->
  <dependency>
    <groupId>org.springframework.experimental</groupId>
    <artifactId>spring-graal-native</artifactId>
    <version>0.6.0.RELEASE</version>
  </dependency>

  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context-indexer</artifactId>
  </dependency>
</dependencies>
```

- What is `spring-graal-native`? The `spring-graal-native` dependency brings together several components. It includes the GraalVM feature implementation. It includes the temporary substitutions (a GraalVM term) for patching around issues with some classes whilst we wait for more correct fixes in these classes. The feature behaviour is actually driven by a set of annotations that encapsulate boot knowledge that is non obvious from a high level static analysis of the code, for example a particular `ImportSelector` may required reflective access to a type. This knowledge is also included in the `spring-graal-native` dependency.
- The `spring-context-indexer` has been in Spring for a while. In a native image, all notion of classpath is lost, so it is not possible to explore the classpath to find components at runtime. The indexer actually produces a list of components at Java compile time and captures it in a `spring.components` file in the built application. If Spring starts and finds this file, it uses it instead of attempting to explore the classpath. The indexer can be used for this whether building a native image or just running your application as a standard Java application.

## Set the `start-class` element

The native image build needs to know the entry point to your application. It does consult a few places to find it. However, in our sample we should set it in the properties section of the `pom.xml` file, as follows:

```
<start-class>com.example.restservice.RestServiceApplication</start-class>
```

## Update the source code

In the case of this sample, there are no changes to be made. However, in some Boot applications, you may need to make some tweaks to ensure that they are not doing anything that is not supported by GraalVM native images.

### Proxies

The only kind of proxy allowed with native images is a JDK proxy. It is not possible to use CGLIB or some other kind of generated proxy. Boot 2.2 added the option to avoid creating these kinds of native image incompatible proxies for configuration class contents, and this happens to suit native image compilation. The enhancement in question is discussed [here](#). Basically, applications need to switch to using `proxyBeanMethods=false` in their configuration annotations. The framework code has already all moved to this model. The following example is from the `webflux-netty` sample:

```
@SpringBootApplication(proxyBeanMethods = false)
public class DemoApplication {

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

    @RestController
    class Foo {

        @GetMapping("/")
        public String greet() {
            return "hi!";
        }
    }
}
```

## Create a location for the generated configuration

When run with the agent, it needs somewhere to store the `.json` files it computes. This can be anywhere, but that location needs to be under a location of `META-INF/native-image` and on the



classpath so that the native image operation automatically pick it up. If we want to keep this configuration around, we can generate it straight into the project (and perhaps store it in version control), as follows:

```
mkdir -p src/main/resources/META-INF/native-image
```



The “proper” location is perhaps a `<groupid>/<artifactid>` location below `native-image` but we keep it simple here for now.

## Run the application with the agent

The following commands run the application with the agent:

```
mvn clean package
java -agentlib:native-image-agent=config-output-dir=src/main/resources/META-
-INF/native-image \
-jar target/rest-service-0.0.1-SNAPSHOT.jar
```

It should run as normal.



While it is up to you to make sure you exercise any codepaths you want to ensure are covered by the native image that will be built, exercising those paths may cause extra reflection access or resource loading or other issues.

Shutdown the app.

Notice that the files that now exist in the folder:

```
ls -l src/main/resources/META-INF/native-image
total 256
-rw-r--r--  1 foo  bar      4 18 Mar 18:59 jni-config.json
-rw-r--r--  1 foo  bar  1057 18 Mar 18:59 proxy-config.json
-rw-r--r--  1 foo  bar 98309 18 Mar 18:59 reflect-config.json
-rw-r--r--  1 foo  bar 17830 18 Mar 18:59 resource-config.json
```

Build a native image for the application, as follows:

```
mvn -Pgraal clean package
```

This uses the feature to do some computation, but it also uses the input generated by the agent.

Did it build cleanly? If so, the resultant executable is in the target folder named after the `start-class` (in this case, `com.example.restservice.RestServiceApplication`).

Did it fail? See the [Troubleshooting](#) page. As of this writing, this step works.

## Run the application

To run the application, run the following executable:

```
./target/com.example.restservice.RestServiceApplication

...
Mar 18, 2020 3:26:16 PM
org.springframework.boot.web.embedded.tomcat.TomcatWebServer start
INFO: Tomcat started on port(s): 8080 (http) with context path ''
Mar 18, 2020 3:26:16 PM org.springframework.boot.StartupInfoLogger logStarted
INFO: Started RestServiceApplication in 0.084 seconds (JVM running for 0.087)
```

The startup time is <100ms, compared ~1500ms when starting the fat jar.

Did your application run successfully? If so, good. If not, see the [Troubleshooting](#) page.

## Summary

Hopefully, this section has given you a taste of the process of building native images. There is much more coming to optimize Spring in all areas: smaller images, reduced memory usage, faster native image compilation, and more. We are also working with the Graal team in all the pitfall areas described earlier. Things across the board should only get better. If you apply these techniques to your own application and have problems, see [Troubleshooting](#).

# Chapter 4. Options

You can tweak the behavior of GraalVM native by setting options. These are passed to `native-image` as command line parameters, in a `META-INF/native-image/native-image.properties` file with an `Arg` key or through the `<buildArgs>..</buildArgs>` block when invoking with Maven.

## 4.1. GraalVM options

GraalVM offers options that are enabled by default and others that are not enabled by default (some of which are recommended, though).

### 4.1.1. Options enabled by default

These options are enabled by default when using `spring-graal-native`, since they are mandatory to make a Spring application work when compiled as GraalVM native images.

- `--allow-incomplete-classpath` allows image building with an incomplete class path and reports type resolution errors at run time when they are accessed the first time, instead of during image building.
- `--report-unsupported-elements-at-runtime` reports usage of unsupported methods and fields at run time when they are accessed the first time, instead of as an error during image building.

### 4.1.2. Options recommended by default

- `--no-server` means do not use the image-build server which can be sometimes unreliable, see [graal#1952](#) for more details.
- `--verbose` makes image building output more verbose.
- `--no-fallback` enforces native image only runtime and disable fallback on regular JVM
- `-H:+ReportExceptionStackTraces` provides more detail should something go wrong.

### 4.1.3. Other options

- `--initialize-at-build-time` initializes classes by default at build time without any class or package being specified. This option is currently (hopefully, temporarily) required for Netty-based applications but is not recommended for other applications, since it can trigger compatibility issues, especially regarding logging and static fields. See [this issue](#) for more details. You can use it with specific classes or package specified if needed.
- `-H:+PrintAnalysisCallTree` helps to find what classes, methods, and fields are used and why. You can find more details in GraalVM [reports documentation](#).
- `-H:+TraceClassInitialization` provides useful information to debug class initialization issues.

## 4.2. Feature options

The current feature options are as follows:

- `-Dspring.graal.remove-unused-autoconfig=true` removes unused configurations, enabling faster compilation and smaller executables.
- `-Dspring.graal.mode=initialization-only` flips the feature into a lightweight mode that only supplies substitutions and initialization configuration. It no longer sets reflection, resource, or proxy data, relying on the project to supply that. This is a good mode for the feature when you use the GraalVM agent to compute the project configuration, as discussed [here](#).
- `-Dspring.graal.verbose=true` outputs lots of information about the feature behavior as it processes auto-configuration and chooses which to include.
- `-Dspring.graal.skip-logback=true` skips Logback configuration.
- `-Dspring.graal.remove-yaml-support=true` removes Yaml support from Spring Boot, enabling faster compilation and smaller executables.
- `-Dspring.graal.dump-config=/tmp/dump.txt` dumps the configuration to the specified file.
- `-Dspring.graal.missing-selector-hints=warning` switches the feature from a hard error for missing hints to a warning. See the [Troubleshooting](#) section for more details on this.

## 4.3. Optional options

- `--enable-all-security-services` required for HTTPS and crypto.
- `--static` builds a statically linked executable, useful to deploy on a `FROM scratch` Docker image.

# Chapter 5. Troubleshooting

While trying to build native images, various things can go wrong, either at image-build time or at runtime when you try to launch the built image. Usually, the problem is a lack of configuration. GraalVM `native-image` probably was not told about the application intending to reflectively call something or load a resource. However, there can also be more serious problems, the `native-image` crashing or the application using a third party library that includes code not compatible with `native-image`.

This section explores some of the errors that can be encountered and possible fixes or workarounds. The two main sections of this page are problems at image-build time and problems at image runtime.

The Spring team is actively working with the Graal team on issues. You can see the currently open issues [here](#). This section tracks Spring-labelled issues on the Graal issue tracker.

## 5.1. `native-image` is failing

The image can fail for a number of reasons. We have described the most common causes and their solutions here.

### 5.1.1. Out of memory error when building the native image

`native-image` consumes a lot of RAM. We have most success on 32G RAM desktop machines. 16G is possible for smaller samples but 8G machines are likely to hit problems more often.

### 5.1.2. You get a weird crash

If you get a “weird crash”, check to see if you get the following error:

```
Exception during JVMCI compiler initialization:
Exception in thread "main": java.lang.ExceptionInInitializerError
java.lang.ExceptionInInitializerError
    at
com.oracle.svm.core.hub.ClassInitializationInfo.initialize(ClassInitializationInfo
.java:290)
    at java.lang.Class.ensureInitialized(DynamicHub.java:496)
    at java.lang.System.getenv(System.java:897)
    at org.graalvm.libgraal.jni.JNIUtil.traceLevel(JNIUtil.java:268)
    at org.graalvm.libgraal.jni.JNIUtil.trace(JNIUtil.java:292)
    at
org.graalvm.libgraal.jni.HotSpotToSVMScope.<init>(HotSpotToSVMScope.java:114)
    at
org.graalvm.compiler.hotspot.management.libgraal.HotSpotGraalManagement.defineClas
sesInHotSpot(HotSpotGraalManagement.java:170)
    at
org.graalvm.compiler.hotspot.management.libgraal.HotSpotGraalManagement.initialize
```

```

(HotSpotGraalManagement.java:115)
    at
org.graalvm.compiler.hotspot.HotSpotGraalRuntime.<init>(HotSpotGraalRuntime.java:178)
    at
org.graalvm.compiler.hotspot.HotSpotGraalCompilerFactory.createCompiler(HotSpotGraalCompilerFactory.java:156)
    at
org.graalvm.compiler.hotspot.HotSpotGraalCompilerFactory.createCompiler(HotSpotGraalCompilerFactory.java:134)
    at
org.graalvm.compiler.hotspot.HotSpotGraalCompilerFactory.createCompiler(HotSpotGraalCompilerFactory.java:52)
    at
jdk.vm.ci.hotspot.HotSpotJVMCIRuntime.getCompiler(HotSpotJVMCIRuntime.java:599)
    at
jdk.vm.ci.hotspot.HotSpotJVMCIRuntime.compileMethod(HotSpotJVMCIRuntime.java:667)
    at
com.oracle.svm.jni.JNIJavaCallWrappers.jniInvoke_VA_LIST_Nonvirtual:Ljdk_vm_ci_hotspot_HotSpotJVMCIRuntime_2_0002ecompileMethod_00028Ljdk_vm_ci_hotspot_HotSpotResolvedJavaMethod_2IJI_00029Ljdk_vm_ci_hotspot_HotSpotCompilationRequestResult_2(JNIJavaCallWrappers.java:0)
Caused by: java.lang.ArrayIndexOutOfBoundsException: Index 83 out of bounds for length 82
    at
com.oracle.svm.jni.functions.JNIFunctions.SetObjectArrayElement(JNIFunctions.java:683)
    at java.lang.ProcessEnvironment.environ(ProcessEnvironment.java)
    at java.lang.ProcessEnvironment.<clinit>(ProcessEnvironment.java:70)
    at
com.oracle.svm.core.hub.ClassInitializationInfo.invokeClassInitializer(ClassInitializationInfo.java:350)
    at
com.oracle.svm.core.hub.ClassInitializationInfo.initialize(ClassInitializationInfo.java:270)
    ... 14 more
Error: Image build request failed with exit status 255

```

This has been seen with Graal 20.0. The current workaround is to re-run your application, because the problem is intermittent.

## 5.2. The built image does not run

If your built image does not run, you can try a number of fixes. This section describes those possible fixes.

### 5.2.1. Missing resource bundles

In some cases, when there is a problem, the error message tries to tell you exactly what to do, as

follows:

```
Caused by: java.util.MissingResourceException:  
Resource bundle not found javax.servlet.http.LocalStrings.  
Register the resource bundle using the option  
-H:IncludeResourceBundles=javax.servlet.http.LocalStrings.
```

Here, it is clear we should add `-H:IncludeResourceBundles=javax.servlet.http.LocalStrings` to the `<buildArgs>` section in the `pom.xml` file.

### 5.2.2. Getting `ClassNotFoundException`

The following listing shows a typical `ClassNotFoundException`:

```
Caused by: java.lang.ClassNotFoundException:  
org.apache.catalina.authenticator.jaspic.AuthConfigFactoryImpl  
at  
com.oracle.svm.core.hub.ClassForNameSupport.forName(ClassForNameSupport.java:60)  
~[na:na]
```

This exception indicates the `native-image` build was not told about the reflective need for a type and, so, did not include the data to satisfy a request to access it at runtime.

If you use the agent, it can be a sign that the agent is missing something (this particular message was collected while debugging [issue 2198](#)).

You have a number of options to address it:

- Download a new version of GraalVM that includes a fixed agent.
- Raise a bug against the `spring-graal-native` project, as a key aim of the feature is to make sure these things get registered. If it is not “obvious” that it should be registered, it may be necessary for a new or expanded hint to be added to the `spring-graal-native-configuration` project (see the [extension guide](#) if you want to explore that).
- Manually add it. The `native-image` run picks up any configuration it finds. In this case, you can create a `reflect-config.json` under a `META-INF/native-image` folder structure and ensure that is on the classpath. This is sometimes most easily accomplished by creating it in your project `src/main/resources` folder (so, `src/main/resources/META-INF/native-image/reflect-config.json`). Note that there are various access options to specify. To fix it, we set `allDeclaredConstructors` and `allDeclaredMethods`. We might need `allDeclaredFields` if the code intends to reflect on those, too. The following entry satisfies the preceding error:

```
[
{
  "name":"org.apache.catalina.authenticator.jaspic.AuthConfigFactoryImpl",
  "allDeclaredConstructors":true,
  "allDeclaredMethods":true
}
]
```

Recompiling should pick up this extra configuration, and the resultant image include metadata to satisfy reflection of `AuthConfigFactoryImpl`.

## 5.3. Where has my logging gone?

In standard operation, `native-image` uses a server to optimize compilation speeds. Using the server has been known to produce native images that are missing logging. In this case, add `--no-server` to the arguments being passed to `native-image`.

### 5.3.1. No access hint found for import selector: XXX

The feature chases down configuration references to other configurations (`@Import` usages). However if you use an import selector, that means code is deciding what the next imported configuration should be, which is harder to follow. The feature does not do that level of analysis (it could get very complicated). This means that, although the feature can tell it has encountered a selector, it does not know what types that selector needs reflective access to or what further configurations it references. Now, the feature could continue. Maybe it would work, and maybe it would crash at runtime. Typically, the error you get can when there is a missing hint can be very cryptic. If the selector is doing a “if this type is around, return this configuration to include”, it may be not finding some type (when it is really there but is not exposed in the image) and not including some critical configuration. For this reason, the feature fails early and fast, indicating that it does not know what a particular selector is doing. To fix it, take a look in the selector in question and craft a quick hint for it. See [this commit](#) that was fixing this kind of problem for a Spring Security [\(issue\)](#).

you can temporarily turn this hard error into a warning. It is possible that, in your case, you do not need what the selector is doing. To do so, specify the `-Dspring.graal.missing-selector-hints=warning` option to cause log messages about the problem but not a hard fail. Note that using warnings rather than errors can cause serious problems for your application.

## 5.4. Diagnosing issues with the feature

Sometimes, you want to use the feature but cannot. Maybe you like that the feature offers that more optimal mode of discarding unnecessary configuration at image-build time, which the agent mode does not. When you use the feature, you either get an error about some missing piece of configuration or, worse, you get no error and it does not work (implying there is probably missing configuration that is not critical for the app to start but is just critical for it to actually work). If the



error is clear, you can follow the guidelines in the [extension guide](#) and perhaps contribute it back. But in the case where you have no idea, what do you do?

The first step to take here is try and run it with the agent, as follows:

```
mkdir -p native-image-config
mvn clean package
java -agentlib:native-image-agent=config-output-dir=native-image-config \
-jar target/myapp-0.0.1-SNAPSHOT.jar
```

After hitting the application through whatever endpoints you want to exercise and shutting it down, there should be config files in the output folder, as follows:

```
ls -l native-image-config
-rw-r--r--  1 foo bar   135 26 Mar 11:25 jni-config.json
-rw-r--r--  1 foo bar   277 26 Mar 11:25 proxy-config.json
-rw-r--r--  1 foo bar 32132 26 Mar 11:25 reflect-config.json
-rw-r--r--  1 foo bar   461 26 Mar 11:25 resource-config.json
```

Now, we want to compare `native-image-config/reflect-config.json` with the configuration being produced by the feature. Luckily, the feature supports a dump mode, where it puts it out on disk for us to see. Add the following to the maven `<buildArgs>...</buildArgs>` section or as a parameter in the direct call to `native-image`:

```
-DdumpConfig=/a/b/c/feature-reflect-config.json
```

Then, after running the native image build again, that file should exist. It is now possible to diff the computed one with the agent one. The scripts folder in `spring-graal-native` contains a compare script, which you can invoke as follows:

```
~/spring-graal-native/scripts/reflectCompare.sh feature-reflect-config.json
native-image-config/reflect-config.json > diff.txt
```

This script produces a summary of the differences. It understands the format a little better than doing a plain `diff`:

```
$ tail diff.txt
...

Summary:
In first but not second: 395
In second but not first: 69
In both files but configured differently: 51
In both files and configured the same: 67
```

We might search that for entries are in the agent file that are not in the computed file for Spring, as follows:

```
grep "^> org.spring" diff.txt
```

This shows data similar to the following:

```
> org.springframework.context.ApplicationEventPublisherAware
setFlags:[allPublicMethods]
> org.springframework.context.ApplicationListener setFlags:[allPublicMethods]
> org.springframework.context.EnvironmentAware setFlags:[allPublicMethods]
> org.springframework.context.SmartLifecycle setFlags:[allPublicMethods]
> org.springframework.core.annotation.AliasFor setFlags:[allDeclaredMethods]
> org.springframework.core.annotation.SynthesizedAnnotation
```

You can craft these into a config file for the project, as follows:

```
mkdir -p src/main/resources/META-INF/native-image
```

Now create `src/main/resources/META-INF/native-image/reflect-config.json` with content similar to the following (including the first one from the diff in this example):

```
[
  {"name":"org.springframework.context.ApplicationEventPublisherAware","allPublicMet
hods":true}
]
```

As we add the details found in the diff, we can rebuild the `native-image` each time and see which

bits help. Once computed, we can create a hint in the feature configuration project that captures this knowledge (see the [extension guide](#) for more info on that) or, if it is more related to this specific application than the infrastructure, we might leave that `reflect-config.json` in the project and commit it to our repository alongside the source for future use.

# Chapter 6. Extension guide

This section describes how to extend Spring Graal Native.

## 6.1. Extending the feature

We are experimenting with extension models, as you will see if you look in the `spring-graal-native-configuration` module within the project. Giant `.json` files are a little unwieldy, and the structure of the Spring Boot autoconfigure module means many types would need to be mentioned in a `.json` file for that module, even though only a fraction of the autoconfigurations are likely to be active in a single application.

What the configuration module is trying to achieve is to tie the access to certain types to the Spring configuration that needs them. Then, for any given application, if we know the likely active configurations, we can expose only the types most likely to be needed.

This section walks through the structure of this hint data and finishes with how to add more configuration for areas we have not yet looked at. Perhaps you want to work on one and submit it as a PR.

## 6.2. Hints

So the giant `.json` file the feature project used to include has been decomposed into lots of `@NativeImageHint` annotations. Each of these hints specifies a set of types that need accessibility, what kind of accessibility is needed, and, optionally, a trigger.

These `@NativeImageHint` should be hosted in one of two places.

First, in the `spring-graal-native-configuration` module, you can see that they are hosted on types that implement the `org.springframework.graal.extension.NativeImageConfiguration` interface (defined by the feature). Implementations of this interface should be listed in a `src/main/resources/META-INF/services/org.springframework.graal.extension.NativeImageConfiguration` file, which the feature loads through regular Java service loading.

Second, they can be put directly onto Spring configuration classes, and they are picked up by the feature.

In this way, if you are experimenting with making your application into a native image, you can keep the hint configuration separate from the application code (the first case above). Once you are happy with it, you can keep it directly with your configuration. The first option also lets hints be provided for code you do not know or otherwise have access to change or rebuild.

## 6.3. Triggered

A trigger in a `NativeImageHint` is optional. If there is a trigger, it is a Spring configuration class name. If that configuration is thought to be active in a particular application, the specified types in the hint are made accessible. If there is no trigger, there are two possibilities:

- The trigger is inferred if the hint is on a Spring configuration type. (It is inferred to be the name of that configuration).
- If purely being used on a type that implements `NativeImageConfiguration`, it is assumed that the specified types must always be accessible. This is useful for some common types to which any application, regardless of active configurations, needs access.

## 6.4. What do hints look like?

The following listing shows a hint:

```
@NativeImageHint(trigger = JacksonAutoConfiguration.class,
    typeInfos = {
        @TypeInfo(types = { JsonGenerator.class },
            access = AccessBits.CLASS | AccessBits.PUBLIC_METHODS
                | AccessBits.PUBLIC_CONSTRUCTORS)
    })
```

Basically, it is that optional trigger and a series of `@TypeInfo` annotations. Here, the trigger is `JacksonAutoConfiguration`, and the hint as a whole reads: *if it looks like `JacksonAutoConfiguration` is active in this application, the `JsonGenerator` type should be made reflectively accessible and the methods and constructors within it should be visible*. One `@TypeInfo` can list multiple types if they share a similar access need, or there may be multiple `@TypeInfo` entries in one hint if different groups of types need different kinds of access.

## 6.5. Optimizing which hints are acted on

With the feature operating during native image construction, it is in a closed world system. This means that the full classpath is known for this application and cannot be extended later. The feature can, therefore, perform configuration condition checks such as `@ConditionalOnClass` as the image is built and know whether configuration attached to that condition can be active. If configuration looks like it is active, the relevant hints are enacted. The feature also chases down references between configurations (`@Import`) and looks for hints on any that get pulled in.

This means that, for any application, there is some 'tree' of configurations active with hints scattered across them.

## 6.6. Structure of the `spring-boot-graal-configuration` module

In the `spring-boot-graal-configuration`, numerous package names look like Spring package names. That is deliberate. Notice the use of direct class references in the hints rather than strings. This type safety is a little more robust. If we upgrade a Spring version and the configuration module no longer compiles, we know something has changed that needs to be addressed. We may not have noticed if we use only string references. The reason these package names match Spring package

names is visibility. With this setup, the hint can refer to a type with only package visibility in the original code. What about private classes? There is a fallback, in that `@TypeInfo` can specify names as strings if it absolutely must, as follows:

```
@TypeInfo(  
    typeNames="com.foo.PrivateBar",  
    types= {PublicBar.class}  
)
```

Notice no access is specified here. There is a default access of everything. All methods, fields, constructors are reflectively accessible and the `.class` bytes are loadable as a resource.

## 6.7. Contributing new hints

The typical approach is:

1. Notice an error if your application when you try to build it or run it—a `classnotfound`, `methodnotfound`, or similar error. If you are using a piece of Spring we don't have a sample for, this is likely to happen.
2. Try to determine which configuration classes give rise to the need for that reflective access to occur. Usually, we do a few searches for references to the type that is missing, and those searches guide us to the configuration.
3. If there is already a `NativeImageConfiguration` implementation for that configuration, augment it with the extra type info. If there is not, create one and attach a `@NativeImageHint` to it, to identify the triggering configuration and the classes that need to be exposed. You also need to set the accessibility in the annotation (in the `@TypeInfo`). It is possible that more dependencies may need to be added to the configuration project to allow the direct class references. That is OK, so long as you ensure that they are provided scope. If you are adding a new `NativeImageConfiguration`, ensure that the `META-INF/services/*` file is updated to reference your new implementation.

## 6.8. Is this the way?

As we play around with this extension mechanism to determine suitability, we are thinking through a number of pros and cons:

Pros:

- The type safety of using direct class references is nice. Grouping types and knowing the trigger that causes their need to be accessible makes the system easier to reason about and debug.
- When looking at one entry in a giant json file, you may have no idea why that is needed. With the hint structure, you can know exactly which configuration causes it to be needed.

Cons:

- Currently, it only works to the granularity of 'all methods' or 'all constructors'. Ideally, it should

let individual methods and constructors be specified. How, the annotations may become unpleasant.

- We cannot specify proxies through this mechanism (or JNI config or the other flavors).
- Not being able to use direct class references for everything is not ideal. It looks like split packages, which is not nice.

So, it is an experiment. We are sure to refactor a few more times before we are done.

# Chapter 7. Contact us

Finally, before you post about it not working, please check the [troubleshooting guide](#), which is full of information on pitfalls, common problems, and how to deal with them (through fixes and workarounds).

We would love to hear about your successes and failures through the project issue tracker. Work has been started on an extension model that makes it easy to support areas of Spring the feature does not yet reach. If you want to make a contribution here, see the [extension guide](#). Please be aware this is pre-1.0 and, as such, some of these options and extension APIs are still evolving and may change before it is finally considered done.