Spring Native documentation

Version 0.12.2 - Andy Clement, Sébastien Deleuze, Filip Hanik, Dave Syer, Esteban Ginez, Jay Bryant, Brian Clozel, Stéphane Nicoll, Josh Long

Table of Contents

Spring Native is now superseded by Spring Boot 3 official native support, see the related reference documentation for more details.

Chapter 1. Overview

Spring Native provides support for compiling Spring applications to native executables using the GraalVM native-image compiler.

Compared to the Java Virtual Machine, native images can enable cheaper and more sustainable hosting for many types of workloads. These include microservices, function workloads, well suited to containers, and Kubernetes

Using native image provides key advantages, such as instant startup, instant peak performance, and reduced memory consumption.

There are also some drawbacks and trade-offs that the GraalVM native project expect to improve on over time. Building a native image is a heavy process that is slower than a regular application. A native image has fewer runtime optimizations after warmup. Finally, it is less mature than the JVM with some different behaviors.

The key differences between a regular JVM and this native image platform are:

- A static analysis of your application from the main entry point is performed at build time.
- The unused parts are removed at build time.
- Configuration is required for reflection, resources, and dynamic proxies.
- Classpath is fixed at build time.
- No class lazy loading: everything shipped in the executables will be loaded in memory on startup.
- Some code will run at build time.
- There are some limitations around some aspects of Java applications that are not fully supported.

The goal of this project is to incubate the support for Spring Native, an alternative to Spring JVM, and provide a native deployment option designed to be packaged in lightweight containers. In practice, the target is to support your Spring applications, almost unmodified, on this new platform.

This is work in progress, see the list of supported features for more details.

1.1. Modules

Spring Native is composed of the following modules:

- `spring-native`: runtime dependency required for running Spring Native, provides also Native hints API.
- `spring-native-configuration`: configuration hints for Spring classes used by Spring AOT plugins, including various Spring Boot auto-configurations.
- `spring-native-docs`: reference guide, in asciidoc format.
- `spring-native-tools`: tools used for reviewing image building configuration and output.

- `spring-aot`: AOT generation infrastructure common to Maven and Gradle plugins.
- `spring-aot-test`: Test-specific AOT generation infrastructure.
- `spring-aot-gradle-plugin`: Gradle plugin that invokes AOT generation.
- `spring-aot-maven-plugin`: Maven plugin that invokes AOT generation.
- `samples`: contains various samples that demonstrate features usage and are used as integration tests.

Chapter 2. Getting started

There are two main ways to build a Spring Boot native application:

- Using Spring Boot Buildpacks support to generate a lightweight container containing a native executable.
- Using the Native Build Tools to generate a native executable.

The easiest way to start a new native Spring Boot project is to go to start.spring.io, add the "Spring Native" dependency and generate the project.

2.1. Getting started with Buildpacks

This section gives you a practical overview of building a Spring Boot native application using Cloud Native Buildpacks. This is a practical guide that uses the RESTful Web Service getting started guide.

2.1.1. System Requirements

Docker should be installed, see Get Docker for more details. Configure it to allow non-root user if you are on Linux.

You can run `docker run hello-world` (without `sudo`) to check the Docker daemon is reachable as expected. Check the Maven or Gradle Spring Boot plugin documentation for more details.

On MacOS, it is recommended to increase the memory allocated to Docker to at least 8GB, and potentially add more CPUs as well. See this Stackoverflow answer for more details. On Microsoft Windows, make sure to enable the Docker WSL 2 backend for better performance.

2.1.2. Sample Project Setup

The completed "RESTful Web Service" guide can be retrieved using the following commands:

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

Validate Spring Boot version

Spring Native 0.12.2 only supports Spring Boot 2.7.7, so change the version if necessary.

Maven

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.7.7</version>
    <relativePath/>
</parent>
```

Gradle Groovy

```
plugins {
    // ...
    id 'org.springframework.boot' version '2.7.7'
}
```

Gradle Kotlin

```
plugins {
    // ...
    id("org.springframework.boot") version "2.7.7"
}
```

Add the Spring Native dependency

`org.springframework.experimental:spring-native` provides native configuration APIs like `@NativeHint` as well as other mandatory classes required to run a Spring application as a native image. You need to specify it explicitly only with Maven.

Maven

```
<dependencies>
    <!-- ... -->
    <dependency>
        <groupId>org.springframework.experimental</groupId>
        <artifactId>spring-native</artifactId>
        <version>0.12.2</version>
    </dependency>
</dependencies>
```

Gradle Groovy

```
// No need to add the spring-native dependency explicitly with Gradle, the Spring AOT
plugin will add it automatically.
```

Gradle Kotlin

```
// No need to add the spring-native dependency explicitly with Gradle, the Spring AOT
plugin will add it automatically.
```

Add the Spring AOT plugin

The Spring AOT plugin performs ahead-of-time transformations required to improve native image compatibility and footprint.

The transformations also apply to the JVM so this can be applied regardless.

Maven

```
<build>
    <plugins>
        <!-- ... -->
        <plugin>
            <groupId>org.springframework.experimental</groupId>
            <artifactId>spring-aot-maven-plugin</artifactId>
            <version>0.12.2</version>
            <executions>
                <execution>
                    <id>generate</id>
                    <goals>
                        <goal>generate</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

Gradle Groovy

```
plugins {
    // ...
    id 'org.springframework.experimental.aot' version '0.12.2'
}
```

Gradle Kotlin

```
plugins {
    // ...
    id("org.springframework.experimental.aot") version "0.12.2"
}
```

The plugin provides a number of options to customize the transformations, see AOT generation for

more details.

Enable native image support

Spring Boot's Cloud Native Buildpacks support lets you build a container for your Spring Boot application. The native image buildpack can be enabled using the `BP_NATIVE_IMAGE` environment variable as follows:

As of Spring Native 0.11, Liberica Native Image Kit (NIK) is the `native-image` compiler distribution used by default with Buildpacks.

Maven

```
<plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
    <configuration>
        <image>
            <builder>paketobuildpacks/builder:tiny</builder>
            <env>
                <BP_NATIVE_IMAGE>true</BP_NATIVE_IMAGE>
            </env>
        </image>
    </configuration>
</plugin>
```

Gradle Groovy

```
bootBuildImage {
    builder = "paketobuildpacks/builder:tiny"
    environment = [
        "BP_NATIVE_IMAGE" : "true"
    ]
}
```

Gradle Kotlin

```
tasks.getByName<BootBuildImage>("bootBuildImage") {
    builder = "paketobuildpacks/builder:tiny"
    environment = mapOf(
            "BP_NATIVE_IMAGE" to "true"
    )
}
```

`tiny` builder allows small footprint and reduced surface attack, you can also use `base` (the default) or `full` builders to have more tools available in the image for an improved developer experience.

Additional `native-image` arguments can be added using the `BP_NATIVE_IMAGE_BUILD_ARGUMENTS` environment variable.

Freeze native image version

By default, `native-image` versions will be upgraded automatically by Buildpacks to the latest release. You can explicitly configure Spring Boot Maven or Gradle plugins with a specific version of `java-native-image` buildpack which will freeze GraalVM version, see related versions mapping. For example, if you want to force using native image `22.1.0`, you can configure:

Maven

```
<plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
    <configuration>
        <!-- ... -->
        <image>
            <buildpacks>
                <buildpack>gcr.io/paketo-buildpacks/java-native-
image:7.19.0</buildpack>
            </buildpacks>
        </image>
    </configuration>
</plugin>
```

Gradle Groovy

```
bootBuildImage {
    // ...
    buildpacks = ["gcr.io/paketo-buildpacks/java-native-image:7.19.0"]
}
```

Gradle Kotlin

```
tasks.getByName<BootBuildImage>("bootBuildImage") {
    // ...
    buildpacks = listOf("gcr.io/paketo-buildpacks/java-native-image:7.19.0")
}
```

Use an alternative native image toolkit

If you want to change the default native image toolkit used by Buildpack (Liberica NIK) to an alternative one, you can explicitly configure Spring Boot Maven or Gradle plugins accordingly, see related Buildpack documentation. For example, if you want to use GraalVM CE instead of Liberica NIK, you can configure:

Maven

```
<plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
    <configuration>
        <!-- ... -->
        <image>
            <buildpacks>
                <buildpack>gcr.io/paketo-buildpacks/graalvm</buildpack>
                <buildpack>gcr.io/paketo-buildpacks/java-native-image</buildpack>
            </buildpacks>
        </image>
    </configuration>
</plugin>
```

Gradle Groovy

```
bootBuildImage {
    // ...
    buildpacks = ["gcr.io/paketo-buildpacks/graalvm", "gcr.io/paketo-buildpacks/java-
native-image"]
}
```

Gradle Kotlin

```
tasks.getByName<BootBuildImage>("bootBuildImage") {
    // ...
    buildpacks = listOf("gcr.io/paketo-buildpacks/graalvm", "gcr.io/paketo-
buildpacks/java-native-image")
}
```

Maven Repository

Configure your build to include the required repository for the `spring-native` dependency, as follows:

Maven

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

Gradle Groovy

```
repositories {
    // ...
    maven { url 'https://repo.spring.io/milestone' }
}
```

Gradle Kotlin

```
repositories {
    // ...
    maven { url = uri("https://repo.spring.io/milestone") }
}
```

The Spring AOT plugin also requires a dedicated plugin repository in the `pom.xml` file for Maven and in the in the `settings.gradle(.kts)` for Gradle.

Maven

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

Gradle Groovy

```
pluginManagement {
    repositories {
        // ...
        maven { url 'https://repo.spring.io/milestone' }
    }
}
```

Gradle Kotlin

```
pluginManagement {
    repositories {
        // ...
        maven { url = uri("https://repo.spring.io/milestone") }
    }
}
```

2.1.3. Build the native application

The native application can be built as follows:

Maven

```
$ mvn spring-boot:build-image
```

Gradle Groovy

```
$ gradle bootBuildImage
```

Gradle Kotlin

```
$ gradle bootBuildImage
```

During the native compilation, you will see a lot of `WARNING: Could not register reflection metadata` messages. They are expected and will be removed in a future version, see #502 for more details.

This creates a Linux container to build the native application using the GraalVM native image compiler. By default, the container image is installed locally.

2.1.4. Run the native application

To run the application, you can use `docker` the usual way as shown in the following example:

```
$ docker run --rm -p 8080:8080 rest-service-complete:0.0.1-SNAPSHOT
```

If you prefer `docker-compose`, you can write a `docker-compose.yml` at the root of the project with the following content:

```
version: '3.1'
services:
  rest-service:
    image: rest-service-complete:0.0.1-SNAPSHOT
    ports:
      - "8080:8080"
```

And then run

```
$ docker-compose up
```

The startup time should be less than `100ms`, compared to the roughly `1500ms` when starting the application on the JVM.

Now that the service is up, visit `localhost:8080/greeting`, where you should see:

```
{"id":1,"content":"Hello, World!"}
```

2.2. Getting started with Native Build Tools

This section gives you a practical overview of building a Spring Boot native application using the GraalVM native build tools. This is a practical guide that uses the RESTful Web Service getting started guide.

2.2.1. System Requirements

A number of prerequisites are required before installing the GraalVM `native-image` compiler. You then need a local installation of the native image compiler.

There are various distributions of the `native-image` compiler available, here we focus on those 2 ones:

- GraalVM CE based on the GraalVM open-source repository and LabsJDK
- Bellsoft Liberica Native Image Kit (NIK) based on the GraalVM open-source repository and Liberica JDK

Linux and MacOS

To install the native image compiler on MacOS or Linux, we recommend using SDKMAN:

- Install SDKMAN.
- Install a GraalVM native-image distribution, either GraalVM CE (`grl` suffix) or Bellsoft Liberica NIK (`nik` suffix), here we go with Liberica NIK Java 11 variant: `sdk install java 22.1.r11-nik`
- Make sure to use the newly installed JDK with `sdk use java 22.1.r11-nik`
- Run `gu install native-image` to bring in the native-image extensions to the JDK.

Alternatively, you can manually install builds from GraalVM or Liberica NIK. Don't forget to set `JAVA_HOME` / `PATH` appropriately if needed and to run `gu install native-image` to bring in the native-image extensions.

Windows

On Windows, follow those instructions to install either GraalVM or Liberica NIK, Visual Studio Build Tools and Windows SDK. Due to a well-known Windows limitations related command-line maximum length, make sure to use x64 Native Tools Command Prompt instead of the regular Windows command line to run Maven or Gradle plugins.

2.2.2. Sample Project Setup

The completed "RESTful Web Service" guide can be retrieved using the following commands:

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

Validate Spring Boot version

Spring Native 0.12.2 only supports Spring Boot 2.7.7, so change the version if necessary.

Maven

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.7.7</version>
    <relativePath/>
</parent>
```

Gradle Groovy

```
plugins {
    id 'org.springframework.boot' version '2.7.7'
    // ...
}
```

Gradle Kotlin

```
plugins {
    id("org.springframework.boot") version "2.7.7"
    // ...
}
```

Add the Spring Native dependency

`org.springframework.experimental:spring-native` provides native configuration APIs like `@NativeHint` as well as other mandatory classes required to run a Spring application as a native image.

Maven

```
<dependencies>
    <!-- ... -->
    <dependency>
        <groupId>org.springframework.experimental</groupId>
        <artifactId>spring-native</artifactId>
        <version>0.12.2</version>
    </dependency>
</dependencies>
```

Gradle Groovy

```
// No need to add the spring-native dependency explicitly with Gradle, the Spring AOT
plugin will add it automatically.
```

Gradle Kotlin

```
// No need to add the spring-native dependency explicitly with Gradle, the Spring AOT
plugin will add it automatically.
```

Add the Spring AOT plugin

The Spring AOT plugin performs ahead-of-time transformations required to improve native image compatibility and footprint.

Maven

```
<build>
    <plugins>
        <!-- ... -->
        <plugin>
            <groupId>org.springframework.experimental</groupId>
            <artifactId>spring-aot-maven-plugin</artifactId>
            <version>0.12.2</version>
            <executions>
                <execution>
                    <id>generate</id>
                    <goals>
                        <goal>generate</goal>
                    </goals>
                </execution>
                <execution>
                    <id>test-generate</id>
                    <goals>
                        <goal>test-generate</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

Gradle Groovy

```
plugins {
    // ...
    id 'org.springframework.experimental.aot' version '0.12.2'
}
```

Gradle Kotlin

```
plugins {
    // ...
    id("org.springframework.experimental.aot") version "0.12.2"
}
```

The plugin provides a number of options to customize the transformations, see AOT generation for more details.

Add the native build tools plugin

GraalVM provides Gradle and Maven plugins to invoke the native image compiler from your build. The following example adds a `native` profile that triggers the plugin during the `package` phase:

Maven

```
<profiles>
        <profile>
            <id>native</id>
            <dependencies>
                <!-- Required with Maven Surefire 2.x -->
                <dependency>
                    <groupId>org.junit.platform</groupId>
                    <artifactId>junit-platform-launcher</artifactId>
                    <scope>test</scope>
                </dependency>
            </dependencies>
            <build>
                <plugins>
                    <plugin>
                        <groupId>org.graalvm.buildtools</groupId>
                        <artifactId>native-maven-plugin</artifactId>
                        <version>0.9.13</version>
                        <extensions>true</extensions>
                        <executions>
                            <execution>
                                <id>build-native</id>
                                <goals>
                                    <goal>build</goal>
                                </goals>
                                <phase>package</phase>
                            </execution>
                            <execution>
                                <id>test-native</id>
                                <goals>
                                    <goal>test</goal>
                                </goals>
                                <phase>test</phase>
                            </execution>
                        </executions>
                        <configuration>
                            <!-- ... -->
                        </configuration>
                    </plugin>
                    <!-- Avoid a clash between Spring Boot repackaging and native-
maven-plugin -->
                    <plugin>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-maven-plugin</artifactId>
                        <configuration>
                            <classifier>exec</classifier>
                        </configuration>
                    </plugin>
                </plugins>
            </build>
```

```
        </profile>
    </profiles>
```

Gradle Groovy

```
// The GraalVM native build tools plugin is applied and configured automatically
```

Gradle Kotlin

```
// The GraalVM native build tools plugin is applied and configured automatically
```

When used with Spring AOT, Native Build Tools Gradle toolchain support is disabled by default in order to avoid current limitations related to identifying in a reliable way JDK with native capabilities. See this related Gradle issue.

Maven Repository

Configure your build to include the milestone repository for the `spring-native` dependency, and the Maven Central one with Gradle for the native build tools one as follows:

Maven

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

Gradle Groovy

```
repositories {
    // ...
    mavenCentral()
    maven { url 'https://repo.spring.io/milestone' }
}
```

Gradle Kotlin

```
repositories {
    // ...
    mavenCentral()
    maven { url = uri("https://repo.spring.io/milestone") }
}
```

Same thing for the plugins:

Maven

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

Gradle Groovy

```
pluginManagement {
    repositories {
        // ...
        mavenCentral()
        maven { url 'https://repo.spring.io/milestone' }
    }
}
```

Gradle Kotlin

```
pluginManagement {
    repositories {
        // ...
        mavenCentral()
        maven { url = uri("https://repo.spring.io/milestone") }
    }
}
```

2.2.3. Build the native application

The native application can be built as follows:

Maven

```
$ mvn -Pnative -DskipTests package
```

Gradle Groovy

```
$ gradle nativeCompile
```

Gradle Kotlin

```
$ gradle nativeCompile
```

This command creates a native executable containing your Spring Boot application in the `target` directory.

2.2.4. Run the native application

To run your application, invoke the following:

```
$ target/gs-rest-service
```

The startup time should be less than `100ms`, compared to the roughly `1500ms` when starting the application on the JVM.

Now that the service is up, visit `localhost:8080/greeting`, where you should see:

```
{"id":1,"content":"Hello, World!"}
```

2.2.5. Test the native application

The native application can be tested as follows:

Maven

```
$ mvn -Pnative test
```

Gradle Groovy

```
$ gradle nativeTest
```

Gradle Kotlin

```
$ gradle nativeTest
```

You can find more details about the native build tools here.

Chapter 3. Support

Spring Native is now superseded by Spring Boot 3 official native support, see the related reference documentation for more details.

This section defines the GraalVM version, languages and dependencies that have been validated against Spring Native 0.12.2, which provides beta support on the scope defined in this section. You can try it on your projects if they are using those supported dependencies, and raise bugs or contribute pull requests if something goes wrong.

Beta support also means that breaking changes will happen, but a migration path will be provided and documented.

3.1. GraalVM

GraalVM version 22.1.0 is supported, see the related release notes. GraalVM issues impacting the Spring ecosystem are identified on their issue tracker using the `spring` label.

3.2. Language

Java 11, Java 17 and Kotlin 1.5+ are supported.

Java compiler `-parameters` flag is required since the `.class` resources, used as a fallback mechanism on the JVM to retrieve parameter names, are typically not available on native. Spring Boot plugin automatically configures it on modules where it is applied, but make sure to configure it explicitly when that's not the case (typically in multi modules projects or when using Kotlin multiplatform).

3.3. Tooling

Maven and Gradle (version 7 or above) are supported.

3.4. Feature flags

Some features like HTTPS may require some additional flags, check Native image options for more details. When it recognizes certain usage scenarios, Spring Native tries to set required flags automatically.

3.5. Spring Boot

Spring Native 0.12.2 has been tested against Spring Boot 2.7.7.

The following starters are supported, the group ID is `org.springframework.boot` unless specified otherwise.

3.5.1. Starters requiring special build configuration

- `spring-boot-starter-web`
  - Only Tomcat is supported for now.
  - `--enable-https` flag is required for server HTTPS support.
  - `org.apache.tomcat.experimental:tomcat-embed-programmatic` dependency should be used for optimized footprint.

Maven

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <exclusions>
        <exclusion>
            <groupId>org.apache.tomcat.embed</groupId>
            <artifactId>tomcat-embed-core</artifactId>
        </exclusion>
        <exclusion>
            <groupId>org.apache.tomcat.embed</groupId>
            <artifactId>tomcat-embed-websocket</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>org.apache.tomcat.experimental</groupId>
    <artifactId>tomcat-embed-programmatic</artifactId>
    <version>${tomcat.version}</version>
</dependency>
```

Gradle Groovy

```
implementation('org.springframework.boot:spring-boot-starter-web') {
    exclude group: 'org.apache.tomcat.embed', module: 'tomcat-embed-core'
    exclude group: 'org.apache.tomcat.embed', module: 'tomcat-embed-websocket'
}
implementation "org.apache.tomcat.experimental:tomcat-embed-
programmatic:${dependencyManagement.importedProperties["tomcat.version"]}"
```

Gradle Kotlin

```
implementation("org.springframework.boot:spring-boot-starter-web") {
    exclude(group = "org.apache.tomcat.embed", module = "tomcat-embed-core")
    exclude(group = "org.apache.tomcat.embed", module = "tomcat-embed-websocket")
}
implementation("org.apache.tomcat.experimental:tomcat-embed-
programmatic:${dependencyManagement.importedProperties["tomcat.version"]}")
```

- spring-boot-starter-actuator
  - WebMvc and WebFlux are supported, as well as metrics and tracing infrastructure.
  - Exclude io.micrometer:micrometer-core when metrics are not used for optimized footprint.

Maven

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
    <exclusions>
        <exclusion>
            <groupId>io.micrometer</groupId>
            <artifactId>micrometer-core</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

Gradle Groovy

```
implementation('org.springframework.boot:spring-boot-starter-actuator') {
    exclude group: 'io.micrometer', module: 'micrometer-core'
}
```

Gradle Kotlin

```
implementation("org.springframework.boot:spring-boot-starter-actuator") {
    exclude(group = "io.micrometer", module = "micrometer-core")
}
```

- spring-boot-starter-test
  - Mockito is not yet supported.
  - See testing support documentation in Getting started with Native Build Tools.

3.5.2. Starters requiring no special build configuration

- spring-boot-starter-amqp
- spring-boot-starter-aop
  - May require additional @AotProxyHint.
- spring-boot-starter-batch
  - May require some additional hints, see batch-io sample.
  - See related #459 issue about supporting class that implements multiple interfaces.
- spring-boot-starter-data-elasticsearch
- spring-boot-starter-data-jdbc
- spring-boot-starter-data-jpa
  - If you want a lighter alternative, spring-boot-starter-data-jdbc which provides a smaller

native footprint is a great alternative.

  - You need to configure Hibernate build-time bytecode enhancement
  - `hibernate.bytecode.provider=none` is automatically set
- `spring-boot-starter-data-mongodb`
  - Multi Document Transactions are currently not supported.
- `spring-boot-starter-data-neo4j`
- `spring-boot-starter-data-r2dbc`
- `spring-boot-starter-data-redis`
- `spring-boot-starter-hateoas`
- `spring-boot-starter-jdbc`
- `spring-boot-starter-logging`
  - Logback is supported with some limitations
    - Configuration with embedded `logback.xml` is not supported yet.
    - Configuration with embedded `logback-spring.xml`, via `myapp -Dlogging.config=logback-config.xml` or `myapp --logging.config=logback-config.xml` is supported but you need to enable XML support and add `org.codehaus.janino:janino` dependency (see the `logger` sample).
    - Conditional processing in Logback configuration with Janino library has limited support. Only simple expressions of `isDefined()` and `isNull()` having string literal as argument are supported.
  - Log4j2 is not supported yet, see #115.
- `spring-boot-starter-mail`
- `spring-boot-starter-oauth2-resource-server`: WebMvc and WebFlux are supported.
- `spring-boot-starter-oauth2-client`: WebMvc and WebFlux are supported.
- `spring-boot-starter-rsocket`
- `spring-boot-starter-security`: WebMvc and WebFlux form login, HTTP basic authentication, OAuth 2.0 and LDAP are supported. RSocket security is also supported.
- `spring-boot-starter-thymeleaf`
- `spring-boot-starter-validation`
- `spring-boot-starter-webflux`:
  - Client and server are supported.
  - For Web support, only Reactor Netty is supported for now.
  - For WebSocket support, Tomcat, Jetty 9, Undertow and Reactor Netty are supported. Jetty 10 is not supported.
- `spring-boot-starter-websocket`
- `com.wavefront:wavefront-spring-boot-starter`
- `spring-boot-starter-quartz`
  - Supports the Quartz Job Scheduling engine.
  - It adds types required by Quartz, and automatically registers any `Job` subclasses for

reflection.

3.6. Spring Cloud

Spring Native 0.12.2 has been tested against Spring Cloud 2021.0.3.

Group ID is `org.springframework.cloud`.

When using Spring Native, `spring.cloud.refresh.enabled` is set to `false` for compatibility and footprint reasons. `spring.sleuth.async.enabled` is also set to `false` since this feature leads to too much proxies created for a reasonable footprint.

- `spring-cloud-starter-config`
- `spring-cloud-config-client`
- `spring-cloud-config-server`
- `spring-cloud-starter-netflix-eureka-client`
- `spring-cloud-starter-task`
- `spring-cloud-function-web`
  - `FunctionalSpringApplication` is not supported
  - `--enable-https` flag is required for HTTPS support.
- `spring-cloud-function-adapter-aws`
- `spring-cloud-starter-function-webflux`
  - `--enable-https` flag is required for HTTPS support.
- `spring-cloud-starter-sleuth`
- `spring-cloud-sleuth-zipkin`

Spring Cloud Bootstrap is no longer supported.

While building a project that contains Spring Cloud Config Client, it is necessary to make sure that the configuration data source that it connects to (such as, Spring Cloud Config Server, Consul, Zookeeper, Vault, etc.) is available. For example, if you retrieve configuration data from Spring Cloud Config Server, make sure you have its instance running and available at the port indicated in the Config Client setup. This is necessary because the application context is being optimized at build time and requires the target environment to be resolved.

3.7. Others

- Micrometer
- Google Cloud Platform libraries via `com.google.cloud:native-image-support` dependency, see this repository for more information
- Lombok

- Spring Kafka
- Spring Session (Redis and JDBC)
- GRPC
- H2 database
- Mysql JDBC driver
- PostgreSQL JDBC driver
- Wavefront

3.8. Limitations

- When using programmatic APIs like `RestTemplate` or `WebClient`, reflection-based serialization like Jackson requires additional `@TypeHint`, this limitation could be removed later via #1152.
- Kotlin Coroutines are supported but currently require additional reflection entries due to how Coroutines generates bytecode with an `Object` return type.
- Sealed class are not supported yet due to github.com/oracle/graal/issues/3870.
- Custom repository implementation fragments need to be annotated with `@Component`.

Chapter 4. AOT generation

This section covers AOT (Ahead Of Time) generation plugins, including how to configure your build for Maven or Gradle. You'll also learn more about AOT runtime modes and more details on the AOT engine.

4.1. Build setup

4.1.1. Maven

The plugin should be declared in your `pom.xml` file:

Maven

```
<dependencies>
    <!-- This is a mandatory dependency for your application -->
    <groupId>org.springframework.experimental</groupId>
    <artifactId>spring-native</artifactId>
</dependencies>
<build>
    <plugins>
        <!-- ... -->
        <plugin>
            <groupId>org.springframework.experimental</groupId>
            <artifactId>spring-aot-maven-plugin</artifactId>
            <version>0.12.2</version>
            <executions>
                <execution>
                    <id>generate</id>
                    <goals>
                        <goal>generate</goal>
                    </goals>
                </execution>
                <execution>
                    <id>test-generate</id>
                    <goals>
                        <goal>test-generate</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

Maven goals `spring-aot:generate` (`prepare-package` phase) and `spring-aot:test-generate` (`process-test-classes` phase) are automatically invoked in the Maven lifecycle when using the `mvn verify` or `mvn package` commands. The `spring-aot:*` goals are not meant to be called directly since they rely on other parts of the lifecycle. Sources are generated in `target/generated-runtime-sources/spring-aot/`

and test sources in `target/generated-runtime-test-sources/spring-aot/`.

When `spring-aot-maven-plugin` is applied, `mvn test -DspringAot=false` runs tests in regular mode while `mvn test` generates related sources and run tests in AOT mode.

Configuration can be performed if needed within the `<configuration>` element, for example to remove SpEL support at build-time if your application does not use it in order to optimize the footprint:

```
<configuration>
    <removeSpelSupport>true</removeSpelSupport>
</configuration>
```

See AOT configuration for a list of the configuration options available.

4.1.2. Gradle

You can configure the Gradle Spring AOT plugin by declaring first the plugin repositories in your `settings.gradle(.kts)` file:

Gradle Groovy

```
pluginManagement {
    repositories {
        // ...
        maven { url 'https://repo.spring.io/milestone' }
    }
}
```

Gradle Kotlin

```
pluginManagement {
    repositories {
        // ...
        maven { url = uri("https://repo.spring.io/milestone") }
    }
}
```

Gradle Groovy

```
plugins {
    // ...
    id 'org.springframework.experimental.aot' version '0.12.2'
}
```

Gradle Kotlin

```
plugins {
    // ...
    id("org.springframework.experimental.aot") version "0.12.2"
}
```

The plugin creates two `SourceSets` for testing and running the application: "aot" and "aotTest". The resulting classes and resources are automatically added to the runtime classpath of the application when running the `aotTest`, `bootRun` and `bootJar` tasks. You can also call directly `generateAot` and `generateTestAot` tasks to perform only the generation.

Sources are generated in `build/generated/runtimeSources/aotMain/`, `build/generated/resources/aotMain/` and test sources in `build/generated/runtimeSources/aotTest/`, `build/generated/resources/aotTest/`.

`test` task runs tests in regular mode while `aotTest` task generates related sources and run tests in AOT mode.

Configuration can be performed if needed using the `springAot` DSL extension, for example to remove SpEL support at build-time if your application does not use it in order to optimize the footprint:

Gradle Groovy

```
springAot {
    removeSpelSupport = true
}
```

Gradle Kotlin

```
springAot {
    removeSpelSupport.set(true)
}
```

Here is a complete code sample showing all the default values and how to set them:

Gradle Groovy

```
import org.springframework.aot.gradle.dsl.AotMode

// ...

springAot {
    mode = AotMode.NATIVE
    debugVerify = false
    removeXmlSupport = true
    removeSpelSupport = false
    removeYamlSupport = false
    removeJmxSupport = true
    verify = true
}
```

Gradle Kotlin

```
import org.springframework.aot.gradle.dsl.AotMode

// ...

springAot {
    mode.set(AotMode.NATIVE)
    debugVerify.set(false)
    removeXmlSupport.set(true)
    removeSpelSupport.set(false)
    removeYamlSupport.set(false)
    removeJmxSupport.set(true)
    verify.set(true)
}
```

The non-idomatic `property.set(…)` syntax in the Gradle Kotlin DSL is due to gradle#9268, feel free to vote for this issue.

See AOT configuration for more details on the configuration options.

4.1.3. AOT configuration

The Spring AOT plugins allow you to express opinions about the source generation process. Here are all the options available:

- `mode` switches how much configuration the plugin actually provides to the native image compiler:
  - `native` (default) generates AOT Spring factories, application context bootstrap, native configuration, native-image.properties as well as substitutions.
  - `native-agent` is designed to be used with the configuration generated by the tracing agent. Generates AOT Spring factories, application context bootstrap, native-image.properties as

well as substitutions.

- **`removeXmlSupport` is set to `true` by default to optimize the footprint, setting it to `false` restores Spring XML support (XML converters, codecs and XML application context support).**
- `removeSpelSupport` is set to `false` by default, setting it to `true` removes Spring SpEL support to optimize the footprint (should be used only on applications not requiring SpEL).
- `removeYamlSupport` is set to `false` by default, setting it to `true` removes Spring Boot Yaml support to optimize the footprint.
- **`removeJmxSupport` is set to `true` by default to optimize the footprint, setting it to `false` restores Spring Boot JMX support.**
- `verify` is set to `true` by default and perform some automated verification to ensure your application is native compliant, setting it to `false` switches off the verifications.
- `debugVerify` is set to `false` by default and enables verification debug when set to `true`.
- `mainClass` allows to specify a main class, useful when multiple ones are present.
- `applicationClass` allows to specify an application class (typically annotated with `@SpringBootApplication`), useful when multiple ones are present.

4.2. Debugging the source generation

The Spring AOT plugins spawns a new process to perform the source generation. To remote debug this process, you can set a debug System property on the command line; then, the source generation process launches with a listener accepting a remote debugger on port `8000` for Maven or `5005` for Gradle.

Maven

```
$ # use the port 8000 by default
$ mvn spring-aot:generate -Dspring.aot.debug=true
$ # configure custom debug options
$ mvn spring-aot:generate -Dspring.aot.debug=
-agentlib:jdwp=transport=dt_socket,server=y,suspend=y,address=8000
$ mvn spring-aot:generate -Dspring.aot.debug="-Xdebug
-Xrunjdwp:transport=dt_socket,server=y,suspend=y,address=9000 -Xnoagent"
```

Gradle

```
$ # use the port 5005 by default
$ ./gradlew generateAot -Dspring.aot.debug=true
$ # configure a custom port
$ ./gradlew generateAot -Dspring.aot.debug=true -Dspring.aot.debug.port=9000
```

If the need to debug the plugins that are involved before the source generation, use the regular related commands:

Maven

```
$ # use the port 8000 by default
$ mvnDebug spring-aot:generate
```

Gradle

```
$ # use the port 5005 by default
$ ./gradlew generateAot -Dorg.gradle.debug=true --no-daemon
```

4.3. AOT runtime modes

The generated sources are automatically used by the native image compilation, but are not used by default when running your application with a JVM. This means that running the application or its tests from the IDE or the command line will not involve those classes.

Any application using Spring AOT can use the `springAot` System property in order to use the AOT classes with a regular JVM. This is mainly useful for debugging purposes in case of issues during native image generation.

When AOT mode is enabled, Spring Boot Developer Tools are ignored as they are not compatible with an AOT approach.

You can set such a property when running an executable Jar from the command line:

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

4.3.1. IDEs

In IDEs, you can specify `-DspringAot=true` when running the application to enable the AOT mode. It requires AOT generation has been invoked before manually via Maven or Gradle.

With IntelliJ IDEA Gradle support, be aware running application in AOT mode is broken in IDEA when delegated to Gradle, see IDEA-287067 related issue. As a workaround, you can go to "File → Plugins ... → Build, Execution, Deployment → Build tools → Gradle" and change "Build and run using" from "Gradle" to "IntelliJ IDEA".

4.3.2. Plugins

For running an application with `gradle bootRun` or `mvn spring-boot:run`, configure your build as following:

Maven

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

Gradle Groovy

```
bootRun {
    systemProperty 'springAot', 'true'
}
```

Gradle Kotlin

```
tasks.getByName<BootRun>("bootRun") {
    systemProperty("springAot", "true")
}
```

4.4. AOT engine

Spring AOT inspects an application at build-time and generates an optimized version of it. Based on your `@SpringBootApplication`-annotated main class, the AOT engine generates a persistent view of the beans that are going to be contributed at runtime in a way that bean instantiation is as straightforward as possible. Additional post-processing of the factory is possible using callbacks. For instance, these are used to generate the necessary reflection configuration that GraalVM needs to initialize the context in a native image.

The engine goes through the following phases:

1. Prepare the underlying `BeanFactory` so that the relevant bean definitions are available. This typically includes bean definitions model parsing (such as `@Configuration`-annotated classes) as well as any additional post-processing of the bean factory.
2. Code generation based on the prepared `BeanFactory`. Each bean definition is handled one by one and the necessary code to instantiate the bean and its necessary runtime semantics (such as primary flag) is generated.
3. Additional processing of the bean factory used to optimize the runtime.

4.4.1. BeanFactory Preparation

As the `BeanFactory` is fully prepared at build-time, conditions are also evaluated. This has an

important difference compared to what a regular Spring Boot application does at runtime. For instance, if you want to opt-in or opt-out for certain features, you need to configure the environment used at build time to do so.

While certain properties like passwords or url can be changed once the application has been prepared, properties that affect, typically, auto-configurations should be set at build-time.

Conditions on the runtime environment, such as enabling features based on your chosen cloud platform, will no longer run at runtime.

A profile is a special sort of condition so these are also evaluated at build-time. It is recommended to avoid the use of profiles as processing them at build-time does not allow you to enable or disable them at runtime anyway. If you want to keep using them, they should be enabled at build-time, for instance by adding the `spring.profiles.active` property in `application.properties`.

Low-level framework callbacks, such as `BeanDefinitionRegistryPostProcessor` are invoked at build-time to create any additional bean definitions. To prevent such a callback to be invoked at runtime again, it is not registered as bean, unless it does not have an `infrastructure` role.

4.4.2. Code Generation

Based on a bean name and a merged `RootBeanDefinition`, the engine identifies a suitable `BeanRegistrationWriter` that is responsible to write the necessary code to instantiate the bean at runtime.

It is not expected that projects have to define their own writers, but this could happen for corner-cases. Writers are identified via implementations of `BeanRegistrationWriterSupplier`, registered in `META-INF/spring.factories`. Suppliers are ordered with a first-win approach, and a default implementation with lowest precedence that handles most use cases is provided.

Explicit care is required if a bean requires privileged access in more than one package. This happens typically if the bean use `protected` access and extends from another class in a different package that does the same. As a rule of thumb, make sure that each custom bean of yours can be instantiated in a test in a usable form.

4.4.3. Additional Processing

Additional processing of the `BeanFactory` currently only scans for `@EventListener`-annotated methods, but future versions may provide additional implementations.

More core to GraalVM support is the generation of an optimized set of native configuration based on the actual beans of the application, as covered by the next section.

Chapter 5. Native hints

GraalVM native image supports configuration via static files that are automatically discovered when located in `META-INF/native-image`. Those files can be native-image.properties, reflect-config.json, proxy-config.json, or resource-config.json.

Spring Native is generating such configuration files (that would sit alongside any user provided ones) via the Spring AOT build plugin. However, there are situations where specifying additional native configuration is required:

- When reflection-based serialization is used in a programmatic API like `WebClient` with Jackson.
- To use a feature or library not yet supported by Spring Native.
- To specify native configuration related to your own application.

Here is the full list of what can be specified in a hint:

- `options` as defined in here that will be passed when executing `native-image`
- `jdkProxies` which list interfaces for which JDK proxy types are needed and should be built into the image.
- `aotProxies` which lists more complex proxy configurations where the proxy will extend a class.
- `types` which lists any reflective needs. It should use class references but string names for classes are allowed if visibility (private classes) prevents a class reference. If these are types that are accessed via JNI and should go into a `jni-config.json` file rather than `reflect-config.json` then ensure the access bit JNI is set when defining access.
- `serializables` which lists any serialization needs via a list of `@SerializationHint` annotations.
- `resources` which lists patterns that match resources (including .class files) that should be included in the image.
- `initialization` which lists classes/packages that should be explicitly initialized at either build-time or run-time. There should not really be a trigger specified on hints included `initialization`.
- `imports` can be useful if two hints share a number of `@TypeHint`/`@JdkProxyHint`/etc in common.

Hints can be provided statically using an annotated model, or programmatically by implementing one of the callback interfaces.

5.1. Annotated Hints

Annotated hints can be put on any `@Configuration`-annotated class of your application, including `@SpringBootApplication`:

- `@TypeHint` for simple reflection hints
- `@NativeHint` is a container for `@TypeHint` and offer more options.

Let us take an example of an application using `WebClient` to deserialize a `Data` class with a `SuperHero` nested class using Jackson. Such process requires reflective access to the class and can be

configured as shown in the following example.

```
@TypeHint(types = Data.class, typeNames = "com.example.webclient.Data$SuperHero")
@SpringBootApplication
public class SampleApplication {
    // ...
}
```

Either the `Class` itself or its fully qualified name can be provided. For nested classes, the `$` separator should be used.

Spring Native itself provides hints for a number of libraries so that they work out-of-the-box. Hints classes should implement `NativeConfiguration` and be registered in `META-INF/spring.factories`. If you need some concrete example of hints, you can browse ours.

5.2. Programmatic Hints

Spring Native provides a programmatic registry which exposes a high-level API for all hints.

Three callbacks are provided:

1. `BeanFactoryNativeConfigurationProcessor` provides the `BeanFactory` so that it can be introspected for matching beans.
2. `BeanNativeConfigurationProcessor` provides a `BeanInstanceDescriptor` for each bean.
3. `NativeConfiguration` typically used for hints not related to beans or `BeanFactory`.

Those types are available via the `org.springframework.experimental:spring-aot` dependency which should not be in the runtime classpath, so you should typically use `<scope>provided</scope>` with Maven or `compileOnly` configuration with Gradle.

`BeanFactoryNativeConfigurationProcessor` should be used when a particular aspect of matching beans is requested. A typical example is automatically processing beans having a certain annotation. `BeanNativeConfigurationProcessor`, however, is more suited when processing all beans, regardless of their nature.

Let us take an example of an application that has `@CustomClient`-annotated beans. Such bean uses a `WebClient` internally and the return types of public methods are DTO used for transfer. As we have seen in the previous example, reflection access is required for those. The sample below registers those hints automatically:

```
class CustomClientNativeConfigurationProcessor implements
BeanFactoryNativeConfigurationProcessor {

    void process(ConfigurableListableBeanFactory beanFactory,
NativeConfigurationRegistry registry) {
        String[] beanNames = beanFactory.getBeanNamesForAnnotation(CustomClient.class
);
        for (String beanName : beanNames) {
            Class<?> clientType = beanFactory.getMergedBeanDefinition(beanName)
.getResolvableType().toClass();
            ReflectionUtils.doWithMethods(clientType, registerNativeConfiguration
(registry), publicDtoMethods());
        }
    }

    private MethodCallback registerNativeConfiguration(NativeConfigurationRegistry
registry) {
        return (method) -> {
            registry.reflection().forType(method.getReturnType())
                    .withAccess(TypeAccess.DECLARED_CONSTRUCTORS, TypeAccess
.DECLARED_METHODS);
        };
    }

    private MethodFilter publicDtoMethods() {
        return (method) -> Modifier.isPublic(method.getModifiers())
                && method.getReturnType() != Void.class;
    }
}
```

Two important bits are worth mentioning:

- The `BeanFactory` parameter is the prepared bean factory and could technically create instances. To prevent that from happening we are retrieving the bean definition, not the bean itself.
- `getMergedBeanDefinition` is preferred as it contains the full resolution.

Custom implementations, such as the `CustomClientNativeConfigurationProcessor` above, should be registered in `META-INF/spring.factories`.

5.3. Manual Hints

Annotations and programmatic hints are automatically invoked as part of the build process if the AOT build plugin is configured. It is also possible to provide directly GraalVM native configuration files if you prefer to do so, but annotation based configuration is usually easier to write and to maintain thanks to auto-completion and compilation type checks. Programmatic hints are easily testable as well.

Chapter 6. Samples

There are numerous samples in the `samples` subfolder of the root project.

Maven projects can be built and tested using a local `native-image` installation using the `build.sh` script file present at the root of each sample. Maven or Gradle projects can be built using Buildpack support using `mvn spring-boot:build-image` or `gradle bootBuildImage` commands which require Docker to be installed.

Beware that native image compilation can take a long time and uses a lot of RAM.

The samples show the wide variety of tech that is working fine: Spring MVC with Tomcat, Spring WebFlux with Netty, Thymeleaf, JPA, and others. The Petclinic samples brings multiple technologies together in one application.

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

Chapter 7. Native image options

GraalVM `native-image` options are documented here. Spring Native is enabling automatically some of those, and some others especially useful are documented here as well.

They can be specified using the `BP_NATIVE_IMAGE_BUILD_ARGUMENTS` environment variable in Spring Boot plugin if you are using Cloud Native Buildpacks or using the `<buildArgs></buildArgs>` configuration element if you are using `native-maven-plugin`.

7.1. Options enabled by default

These options are enabled by default when using Spring 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.
- `--no-fallback` enforces native image only runtime and disable fallback on regular JVM.
- `--install-exit-handlers` allows to react to a shutdown request from Docker.

7.2. Useful options

- `--verbose` makes image building output more verbose.
- `-H:+ReportExceptionStackTraces` provides more detail should something go wrong.
- `--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.
- `-H:Log=registerResource:3` prints the resources included in the native image. You can find more details in GraalVM reports documentation.
- `-H:ReportAnalysisForbiddenType=com.example.Foo` helps to find why the specified class is included in the native image.
- `--trace-class-initialization` provides a comma-separated list of fully-qualified class names that a class initialization is traced for.
- `--trace-object-instantiation` provides a comma-separated list of fully-qualified class names that an object instantiation is traced for.
- `--enable-https` enables HTTPS support (common need when using `WebClient` or `RestTemplate` for example).

7.3. Unsupported options

- `--initialize-at-build-time` without class or package specified is not supported since Spring Native for GraalVM is designed to work with runtime class initialization by default (a selected set of classes are enabled at buildtime).

Chapter 8. Tracing agent

The GraalVM native image tracing agent allows to intercept reflection, resources or proxy usage on the JVM in order to generate the related native configuration. Spring Native should generate most of this native configuration automatically, but the tracing agent can be used to quickly identify the missing entries.

When using the agent to compute configuration for native-image, there are a couple of approaches:

- Launch the app directly and exercise it.
- Run application tests to exercise the application.

The first option is interesting for identifying the missing native configuration when a library or a pattern is not recognized by Spring Native.

The second option sounds more appealing for a repeatable setup but by default the generated configuration will include anything required by the test infrastructure, which is unnecessary when the application runs for real. To address this problem the agent supports an access-filter file that will cause certain data to be excluded from the generated output.

8.1. Running the application with the agent to compute configuration

It is possible to use the tracing agent to run the application in AOT mode in order to compute the native configuration:

- Configure the AOT plugin to use the `native-agent` mode.
- Build the Spring Boot application with the AOT plugin enabled.
- Run the app with the tracing agent and generate the config temporarily in `src/main/resources` with for example `java -DspringAot=true -agentlib:native-image-agent=config-output-dir=src/main/resources/META-INF/native-image -jar target/myapp-0.0.1-SNAPSHOT.jar`.
- Check if the application is now working fine on native.

8.2. Testing with the agent to compute configuration

8.2.1. A basic access-filter file

Create the following `access-filter.json` file at the root of your project.

```
{ "rules": [
  {"excludeClasses": "org.apache.maven.surefire.**"},
  {"excludeClasses": "net.bytebuddy.**"},
  {"excludeClasses": "org.apiguardian.**"},
  {"excludeClasses": "org.junit.**"},
  {"excludeClasses": "org.mockito.**"},
  {"excludeClasses": "org.springframework.test.**"},
  {"excludeClasses": "org.springframework.boot.test.**"},
  {"excludeClasses": "com.example.demo.test.**"}
  ]
}
```

Most of these lines would apply to any Spring application, except for the last one which is application specific and will need tweaking to match the package of a specific applications tests.

8.2.2. Using the access-filter file

The access-filter.json file is specified with the `access-filter-file` option as part of the agentlib string:

```
-agentlib:native-image-agent=access-filter-file=access-filter.json,config-output
-dir=target/classes/META-INF/native-image
```

8.2.3. Using it with maven

Let's look at how to pull the ideas here together and apply them to a project.

Since Spring takes an eager approach when building the application context, a very basic test that starts the application context will exercise a lot of the Spring infrastructure that needs to produce native-image configuration. This test would suffice for that and could be placed in `src/test/java`:

```
package com.example.demo.test;

import org.junit.jupiter.api.Test;

import org.springframework.boot.test.context.SpringBootTest;

@SpringBootTest
public class ApplicationTests {

    @Test
    public void contextLoads() {
    }
}
```

Make sure to exercise all the required code path to allow the agent to generate all the required native configuration. For example, that could mean to request with an http client all the web endpoints.

This following snippet would go into the maven pom:

```
<plugins>
    <!-- ... -->
    <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-antrun-plugin</artifactId>
        <executions>
            <execution>
                <id>copy-agent-config</id>
                <phase>prepare-package</phase>
                <goals>
                    <goal>run</goal>
                </goals>
                <configuration>
                    <target>
                        <mkdir dir="${project.build.directory}/native/agent-
output/main"/>
                        <copy todir="${project.build.directory}/native/agent-
output/main">
                            <fileset dir="${project.build.directory}/native/agent-
output/test" />
                        </copy>
                    </target>
                </configuration>
            </execution>
        </executions>
    </plugin>
    <plugin>
        <groupId>org.graalvm.buildtools</groupId>
        <artifactId>native-maven-plugin</artifactId>
        <extensions>true</extensions>
        <configuration>
            <agent>
                <enabled>true</enabled>
                <options name="test">
                    <option>access-filter-file=$/home/seb/workspace/spring-
native/spring-native-docs/access-filter.json</option>
                </options>
            </agent>
        </configuration>
    </plugin>
</plugins>
```

You need to activate the AOT mode when running the tracing agent on the application as documented in AOT runtime modes, for tests this is not needed since AOT mode is enabled automatically when AOT files are detected.

Also update the `spring-aot` build plugin to enable the `native-agent` mode in order to not generate `*-`

`config.json` files since the agent will take care of that:

```
<plugin>
    <groupId>org.springframework.experimental</groupId>
    <artifactId>spring-aot-maven-plugin</artifactId>
    <configuration>
        <mode>native-agent</mode>
    </configuration>
</plugin>
```

Build the native image with `mvn -Pnative -DskipNativeTests package`. If that's not enough, you can add additional native configuration using `@NativeHint` annotations.

Chapter 9. Executable JAR to native

It is possible to turn a Spring Boot executable JAR to a native executable, or a container image containing a native executable. This can be useful for various use cases:

- Keep the regular JVM pipeline and turn the JVM Spring Boot application to native on the CI/CD platform.
- Keep an architecture neutral deployment artifact, as `native-image` does not support cross-compilation.

A mandatory pre-requisite is to use AOT generation Maven or Gradle upstream to build the Spring Boot executable JAR.

9.1. With Buildpacks

Spring Boot applications usually use Buildpacks via the Maven (`mvn spring-boot:build-image`), or Gradle (`gradle bootBuildImage`) integration. You can also use directly the `pack` CLI to turn a Spring Boot executable JAR built with AOT generation into an optimized container image.

First, make sure that a Docker daemon is available, either locally or remotely. You also need to Install `pack`.

Assuming a Spring Boot executable JAR built as `my-app-0.0.1-SNAPSHOT.jar` in the `target` directory, run:

```
pack build --builder paketobuildpacks/builder:tiny \
    --path target/my-app-0.0.1-SNAPSHOT.jar --env 'BP_NATIVE_IMAGE=true' my-app:0.0.1
```

This does not require a local `native-image` installation.

9.2. With `native-image`

Another option is to turn a Spring Boot executable JAR built with AOT generation into a native executable using the GraalVM `native-image` compiler. For this to work, you need to Install native-image.

Assuming a Spring Boot executable JAR built as `my-app-0.0.1-SNAPSHOT.jar` in the `target` directory:

```
#!/usr/bin/env bash

rm -rf target/native
mkdir -p target/native
cd target/native
jar -xvf ../my-app-0.0.1-SNAPSHOT.jar >/dev/null 2>&1
cp -R META-INF BOOT-INF/classes
native-image -H:Name=my-app -cp BOOT-INF/classes:`find BOOT-INF/lib | tr '\n' ':'`
mv my-app ../
```

This is documented as a simple bash script but can be adapted to whatever is suitable to you environment.

Chapter 10. 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 native configuration, so be sure to check Native hints first thing. Reading Native image reference documentation could also probably help.

This section explores some of the errors that can be encountered and possible fixes or workarounds.

Make sure to check GraalVM native image known issues related to Spring as well as Spring Native open issues before creating a new one.

10.1. `native-image` is failing

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

10.1.1. DataSize was unintentionally initialized at build time

If you see an error like:

```
Error: Classes that should be initialized at run time got initialized during image
building:
 org.springframework.util.unit.DataSize was unintentionally initialized at build time.
To see why org.springframework.util.unit.DataSize got initialized use --trace-class
-initialization
```

You have probably tried to compile a Spring Boot application to native without the `spring-native` dependency and Spring AOT plugin. See related Getting started with Native Build Tools and Getting started with Buildpacks documentation.

10.1.2. WARNING: Could not register reflection metadata

Those warnings are expected for now, and should be removed in a future version, see #502 for more details.

10.1.3. Out of memory error when building the native image

Out of memory error can materialize with error messages like `Error: Image build request failed with exit status 137`.

`native-image` consumes a lot of RAM, we recommend a machine with at least 16G of RAM.

If you are using containers, on Mac, it is recommended to increase the memory allocated to Docker to at least 8G (and potentially to add more CPUs as well) since `native-image` compiler is a heavy process. See this Stackoverflow answer for more details.

On Windows, make sure to enable the Docker WSL 2 backend for better performances.

10.1.4. Builder lifecycle 'creator' failed with status code 145

This is a generic error triggered by Docker and forwarded by Spring Boot Buildpacks support. `native-image` command has likely failed, so check the error messages in the output. If you can't find anything, check if that's not an out of memory error as described above.

10.2. 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.

10.2.1. 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.
```

You should add resource configuration using Native hints.

10.2.2. Application failed to start when running `mvn spring-boot:run`

Because of a temporary limitation of the AOT plugin, developers need to trigger the `package` phase if they wish to run the application with the Spring Boot Maven plugin: please use `mvn package spring-boot:run`.

10.2.3. Missing configuration

The Spring AOT plugin will do the best it can to catch everything but it doesn't understand every bit of code out there. In these situations you can write native configuration yourself, see Native hints, Tracing agent and How to contribute.

10.2.4. AotProxyHint errors

When running native image an error indicating a `AotProxyHint` is missing may be produced, like this:

```
Caused by: java.lang.IllegalStateException: Class proxy missing at runtime, hint
required at build time:
@AotProxyHint(targetClass=com.example.batch.ItemReaderListener.class,
interfaces={org.springframework.aop.scope.ScopedObject.class,
java.io.Serializable.class,
org.springframework.aop.framework.AopInfrastructureBean.class})
```

This indicates a hint was missing to construct the proxy at build time. New classes cannot be generated at runtime in native images. By including these hints on your application (alongside the other hints), the original building of your application will generate the required proxy classes (and their support classes) and they will be included in the native-image. At runtime when a class proxy is required, these classes generated earlier will then be loaded.

The error message includes exactly the hint text that needs to be pasted into the source.

A class proxy is a proxy that extends a class. This is in contrast to a regular `JdkProxyHint` which only specifies a set of interfaces to be implemented by a JDK Proxy class.

10.2.5. No access hint found for import selector: XXX

See [how-to-contribute-design-import-selectors].

10.3. Working with Multi-Modules projects

The Spring Boot and AOT plugins should only be applied to the module that contains the main application class. We've shared a sample application showing how to set up multi-modules projects with Gradle and Maven.

10.4. Working with snapshots

Snapshots are regularly published and obviously ahead of releases and milestones. If you wish to use the snapshot versions you should use the following repository:

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

Chapter 11. How to contribute

This section describes how to contribute native support for libraries or features used in Spring applications. This can be done either by submitting submit pull requests to Spring Native for the scope supported on start.spring.io, or by providing native support directly at library or application level otherwise.

11.1. Designing native-friendly Spring libraries

Native support is mostly about making an application and its libraries possible to analyze at build-time to configure what's required or not at runtime. The goal is to do that in an optimal way to have a minimal footprint.

Spring applications are dynamic, which means they typically use Java language features like reflection in various places. Spring Native and its Spring AOT build plugins performs AOT transformations, in the context of a specific application classpath and configuration in order to generate the optimal native configuration. They also generate programmatic versions of `spring.factories` or auto-configurations to reduce the amount of reflection required at runtime.

Each reflection entry (per constructor/method/field) leads to the creation of a proxy class by `native-image`, so from a footprint point of view, these AOT transformations allow a smaller and more optimal configuration to be generated.

The documentation below describes best practices to keep in mind when trying to make Spring code more compatible with native-images.

11.1.1. Use `proxyBeanMethods=false` or method parameter injection in `@Configuration` classes

In native applications, `@Bean` annotated methods do not support cross `@Bean` invocations since they require a CGLIB proxy created at runtime. This is similar to the behavior you get with the so called lite mode or with `@Configuration(proxyBeanMethods=false)`.

It is fine for applications to just use `@Configuration` without setting `proxyBeanMethods=false` and use method parameters to inject bean dependencies, this is handled by Spring Native to not require a CGLIB proxy.

Libraries are encouraged to use `@Configuration(proxyBeanMethods=false)` (most of Spring portfolio currently uses this variant) since it is generally a good idea to avoid CGLIB proxies if not needed and to provide native compatibility. This behavior could potentially become the default in a future Spring Framework version.

11.1.2. Use NativeDetector for native conditional code paths

Spring related code should use `NativeDetector.inNativeImage()` (provided by `spring-core` dependency in the `org.springframework.core` package) to detect native-specific code paths. Spring Framework or Spring Data takes advantage of this utility method to disable CGLIB proxies since they are not supported in native images for example.

Whenever possible, we recommend writing code that works in both contexts rather than always falling back on the `NativeDetector`. This way, common code is easier to reason about and test/debug.

11.1.3. Do classpath checks in static block/fields and configure build-time initialization

It is possible to configure code in your application/dependencies to run at image build time. This will speed up the runtime performance of your image and reduce the footprint.

If the behaviour of some code is conditional on some class being present on the classpath, that presence check can be performed when the image is built because the classpath cannot be changed after that.

A presence check is normally done via an attempt to reflectively load a class. It is optimal if that check can be performed as the native image is built, then no reflective configuration is necessary for that presence check at runtime. To achieve this optimization:

- Perform the presence check in a static block/field in a type.
- Configure that type containing the check to be initialized at build-time using `@NativeHint`

Care must be taken to limit as much as possible the amount of other classes transitively initialized at build-time, since it can introduce serious compatibility issues.

11.1.4. Move reflection to build-time when possible

It is fine to use reflection in a native world but it is most optimal to do it in code executed at build-time:

- In the static block/fields of a class initialized at build-time.
- In an AOT transformation run as a Spring AOT build plugin.

More guidelines will be provided here as AOT generation matures.

11.2. Contributing new hints

When contributing non-Spring related hints, you can use regular native image configuration as documented in their reference documentation.

For most cases Spring Native understands how Spring applications operate - how configurations refer to each other, how beans are going to be instantiated, etc. However, there are some subtleties that it doesn't understand and to plug those knowledge gaps it relies on hints, these tell the system what extra configuration may be needed for the native image build when particular auto configurations or libraries are active in an application.

A hint may indicate that a specific resource must be included or that reflection on a particular type is required.

When adding support for a new area of Spring or new version of a library, the typical approach to work out the missing hints is as follows:

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 `NativeConfiguration` implementation for that configuration, augment it with the extra type info. If there is not, create one, attach a `@NativeHint` to it to identify the triggering configuration and the classes that need to be exposed, and add it to the `META-INF/services/org.springframework.nativex.extension.NativeConfiguration`. You may also need to set the accessibility in the annotation (in the `@TypeHint`). 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.

See Native hints for basic hint documentation. These `@NativeHint` can be hosted in one of two places:

- In the `spring-native-configuration` module, you can see that they are hosted on types that implement the `org.springframework.nativex.extension.NativeConfiguration` interface. Implementations of this interface should be listed in a `src/main/resources/META-INF/spring.factories` file as comma separated values for the `org.springframework.nativex.type.NativeConfiguration` key.
- On Spring configuration classes. That's useful for project-specific hints or while crafting hints on a sample before moving it to the `spring-native-configuration` module (shorter feedback loop).

An `attribute` trigger can be specified on the `@NativeHint` annotation.

- If the hint is on a `NativeConfiguration` class, and no trigger is specified then it is assumed this configuration should **always** apply. This is useful for common configuration necessary for all applications.
- If the hint is on something other than a `NativeConfiguration` class (e.g. on a Spring auto-configuration class) then that type is considered to be the trigger, and if the Spring AOT plugin determines that is 'active', the hint applies.

The `trigger` attribute might be a piece of Spring infrastructure (autoconfiguration) or just a regular class. If the Spring AOT plugin determines that Spring infrastructure may be active when the application runs, or (for a regular class trigger) that the named class is on the classpath, it will activate the associated hints, informing the native-image build process what is needed.

It is best practice to use the hints in a sample (existing or new one) in order to have automated testing of it. Once you are happy with the hints you crafted, you can submit a pull request.

Using the Tracing agent can also be useful an approximation of the required native configuration without having to run too many native builds.

11.3. Dynamic native configuration

You can provide dynamic native configuration by:

- Providing a `org.springframework.nativex.extension.NativeConfiguration` implementation (require a `provided` (Maven) or `compileOnly` (Gradle) dependency on `org.springframework.experimental:spring-aot` dependency).
- Listing this implementation in a `src/main/resources/META-INF/spring.factories` file as comma separated values for the `org.springframework.nativex.type.NativeConfiguration` key.

11.3.1. Implementing `NativeConfiguration`

Sometimes the necessary configuration is hard to statically declare and needs a more dynamic approach. For example, the interfaces involved in a proxy hint might need something to be checked beyond the simple presence of a class. In this case the method `computeHints` can be implemented which allows computation of hints in a more dynamic way, which are then combined with those statically declared via annotations.

The `NativeConfiguration` interface contains a couple of default methods that can be implemented for more control. For example whether the hints on a `NativeConfiguration` should activate may be a more subtle condition that simply whether a configuration is active. It is possible to implement the `isValid` method in a `NativeConfiguration` implementation and perform a more detailed test, returning false from this method will deactivate the associated hints.

11.3.2. Taking more control via processors

Within a Spring application there are going to be a number of active components (the main application, configurations, controllers, etc). There may be much more sophisticated domain specific analysis to be done for these components in order to compute the necessary configuration for the `native-image` invocation. It is possible to implement a couple of interfaces to participate in the process like `BeanFactoryNativeConfigurationProcessor` or `BeanNativeConfigurationProcessor`.

They should be registered via `src/main/resources/META-INF/spring.factories` as well.

11.4. Using container-based build environment

To allow easily reproducible builds of `spring-native`, dedicated interactive Docker images are available for local development (tested on Linux and Mac) and are also used on CI:

- `graalvm-ce`: base image with Ubuntu bionic + GraalVM native, built daily by the CI and available from Docker hub
- `spring-native`: base image with `graalvm-ce` + utilities required to build the project, available from Docker hub
- `spring-native-dev`: local image built via `run-dev-container.sh` designed to share the same user between the host and the container.

To use it:

- Install Docker.
- Configure it to allow non-root user if you are on Linux.
- On Mac, ensure in the Docker preferences resources tab that you give it enough memory, ideally 10G or more, otherwise you may see out of memory issues when building images.
- Run `run-dev-container.sh` to run the Docker container with an interactive shell suitable to run `spring-native` build scripts (see below for more documentation).
- The first time, it will download remotely hosted images built by CI.
- The current and the Maven home directories are shared between the host (where is typically the IDE) and the container (where you can run builds).

11.4.1. `run-dev-container.sh`

`run-dev-container.sh` runs Spring Native for GraalVM dev container with an interactive shell.

```
run-dev-container.sh [options]

options:
-h, --help                show brief help
-j, --java=VERSION        specify Java version to use, can be 8 or 11, 11 by default
-g, --graalvm=VERSION     specify GraalVM flavor to use, can be stable or dev, stable
by default
-w, --workdir=/foo        specify the working directory, should be an absolute path,
current one by default
-p, --pull                force pulling of remote container images
-r, --rebuild             force container image rebuild
```

11.4.2. Usual dev workflow

- Import the root project in your IDE.
- Eventually import the sample you are working on as a distinct project in your IDE.
- Run the root project `build.sh` (from the host or the container) if you have made modification to the feature, substitutions or configuration modules.
- Make sure `native-image` is in the `PATH` (usually done by switching to a GraalVM installation with SDKMAN).
- Run `build.sh` of the sample you are working on from the container.

To test the various samples You can also run the root `build.sh` then `build-key-samples.sh` (test only key samples) or `build-samples.sh` (test all samples) from the container.

11.5. Scripts

The `native-image` command supports a number of flags for producing information about what is in an image. However, what can sometimes be really useful is comparing two images. What is in one that isn't in the other? Sometimes sifting through the mass of output is tricky. The scripts folder

provides some tools to help with this.

11.5.1. Comparing images

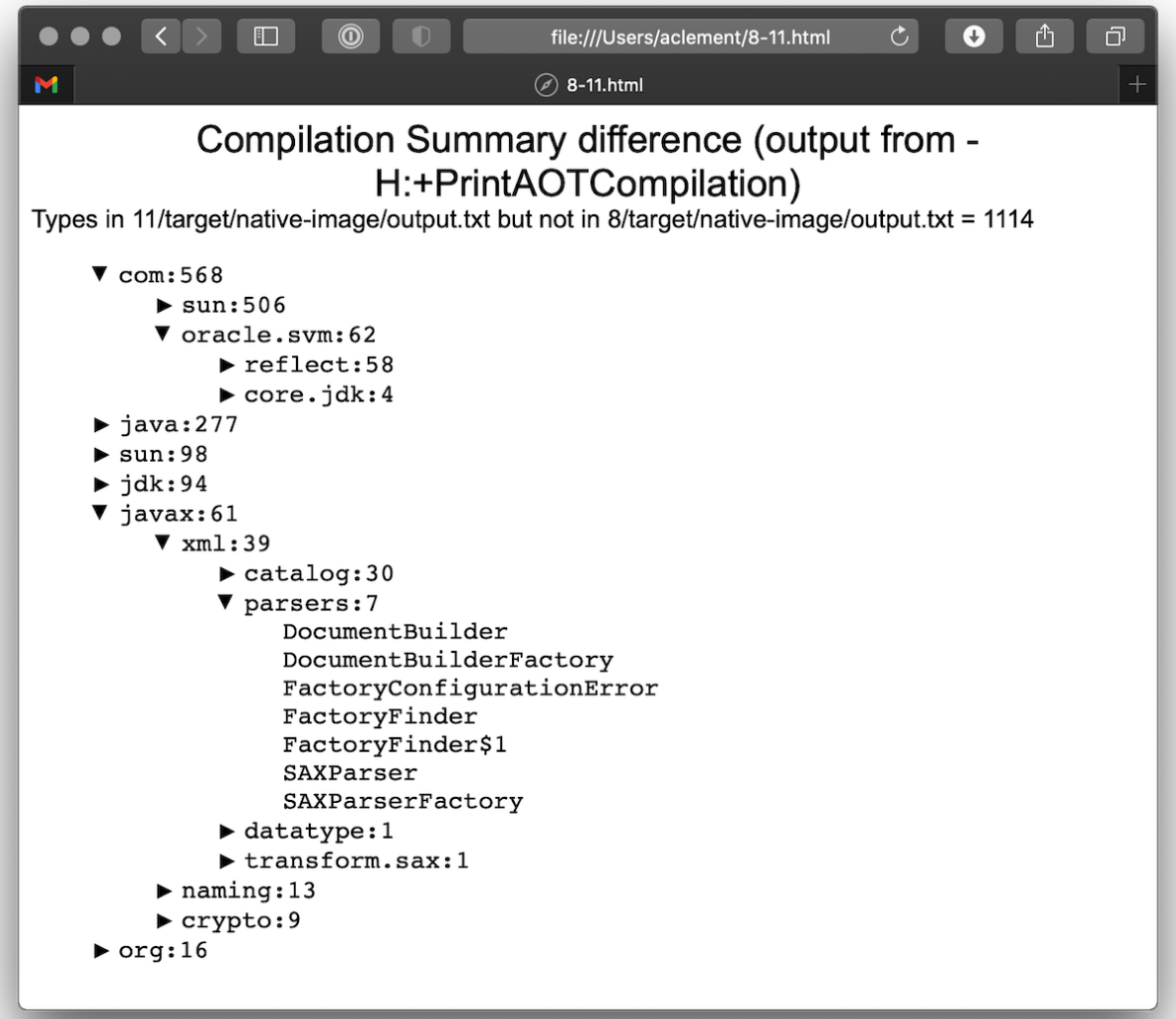
First up is `-H:+PrintAOTCompilation` which prints logging information during compilation, looking a bit like this:

```
Compiling FieldPosition[] java.text.DecimalFormat.getNegativeSuffixFieldPositions()
[Direct call from StringBuffer DecimalFormat.subformat(StringBuffer,
Format$FieldDelegate, boolean, boolean, int, int, int, int)]
Compiling FieldPosition[] java.text.DecimalFormat.getPositiveSuffixFieldPositions()
[Direct call from StringBuffer DecimalFormat.subformat(StringBuffer,
Format$FieldDelegate, boolean, boolean, int, int, int, int)]
```

Thousands and thousands of lines typically. Typically we turn on that option for `native-image` in the `pom.xml`. The output is produced to stdout which our samples capture in `target/native/output.txt`. With two builds done, we can use a script from this folder to produce a tree diff:

```
compilationDiff.sh java8build/target/native/output.txt
java11build/target/native/output.txt 8-11.html
```

The inputs are the two collected PrintAOTCompilation outputs to compare and the name for an HTML file that should be generated (this will contain the navigable tree). Then simply open the HTML file.



One of the key entries to look at in the diff is under the path `com/oracle/svm/reflect` as that shows the entries included due to reflection.

Chapter 12. Contact us

We would love to hear about your successes and failures (with minimal repro projects) through the project issue tracker. Before raising an issue, please check the troubleshooting guide, which is full of information on pitfalls, common problems, and how to deal with them (through fixes and workarounds).

If you want to make a contribution here, see the how to contribute guide. Please be aware this project is still incubating and, as such, some of these options and extension APIs are still evolving and may change before it is finally considered done.