What Is Serverless?

What Is Serverless?

Serverless doesn’t mean there are no servers, it means you don’t care about them.

Serverless can be grouped into two areas:

- Backend as a Service (BaaS) Replacing server-side, self-managed components with off-the-shelf services
- Functions as a Service (FaaS) A new way of building and deploying server-side software, oriented around deploying individual functions

The key is that with both, you don't have to manage your own server hosts or server processes and can focus on business value!

What Is Serverless?

Key characteristics

A Serverless service …

- … does not require managing a long-lived host or application instance
- … self auto-scales and auto-provisions, dependent on load
- … has implicit high availability
- … has performance capabilities defined in terms other than host size/count
- … has costs that are based on precise usage, up from and down to zero usage

Why Serverless?

Reduced risk

Shorter lead time

Reduced packaging and deployment complexity

Increased flexibility of scaling

Reduced labor and resource costs

Reduced risk

Drawbacks / Limitations of Serverless

- Unpredictable costs
- Spinning up machines takes time - from a few seconds to minutes
- Most Serverless applications are stateless and the management of state can be somewhat tricky
- Vendor lock-in unless you are using OSS projects like e.g. Knative
- Loss of control over
  - absolute configuration
  - the performance of Serverless components
  - issue resolution
  - security
- Higher latency due to inter-component communication over HTTP APIs and “cold starts”

More Drawbacks / Limitations of Serverless

- Problematic with downstream systems that cannot increase their capacity quickly enough
- Typically limited in how long each invocation is allowed to run
- Multitenancy problems
- Debugging is more complicated (a single request can travel between several machines and some of those machines disappear at times)
- Added work is needed to provide tracing and monitoring solutions, which can add complexity and cost to the project
- Security can be more demanding in a serverless environment
- The difficulty of local testing

What Is Knative?

An open-source community project which adds components for deploying, running, and managing applications on any Kubernetes in a Serverless way.

To primary components:

- Serving: Supports deploying and serving of serverless applications and functions
- Eventing: Enables developers to use an event-driven architecture with serverless applications

Summary

The primary drivers for the adoption of Serverless are:

Developer productivity

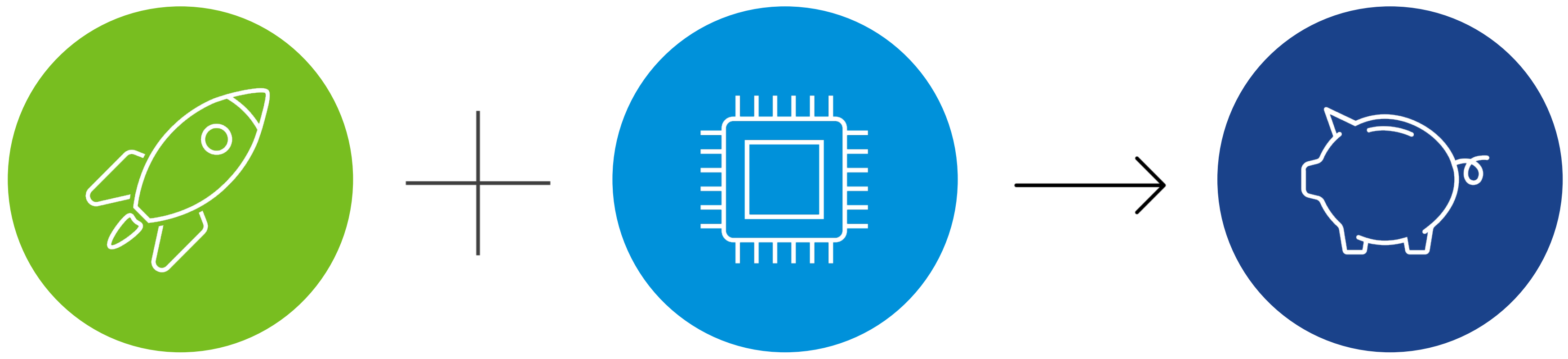
Platform elasticity

Cost savings

Going Serverless

with your Spring Boot applications

Unleash the Full Potential of Serverless for Our Application



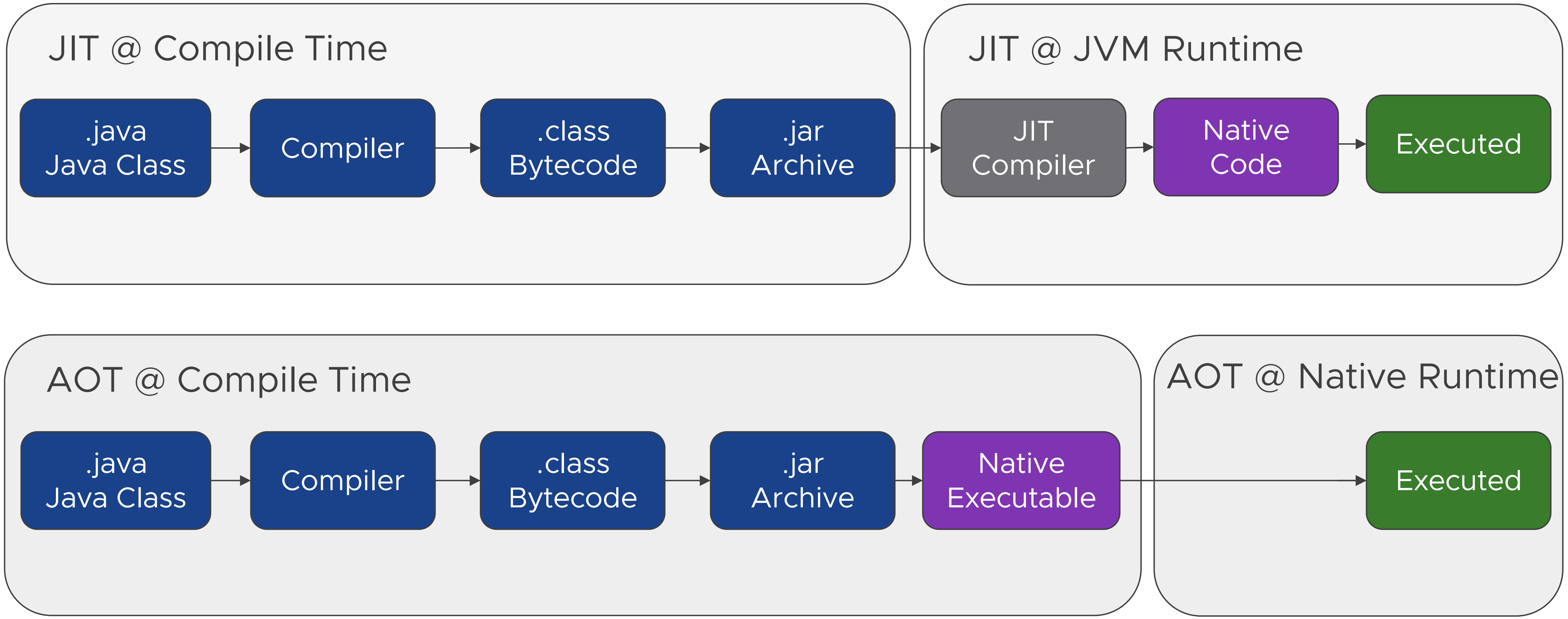
Faster startup time

Lower resource consumption (memory, CPU)

More cost savings

Just-In-Time vs Ahead-Of-Time

Interpretation vs Compilation



What Are Native Images?

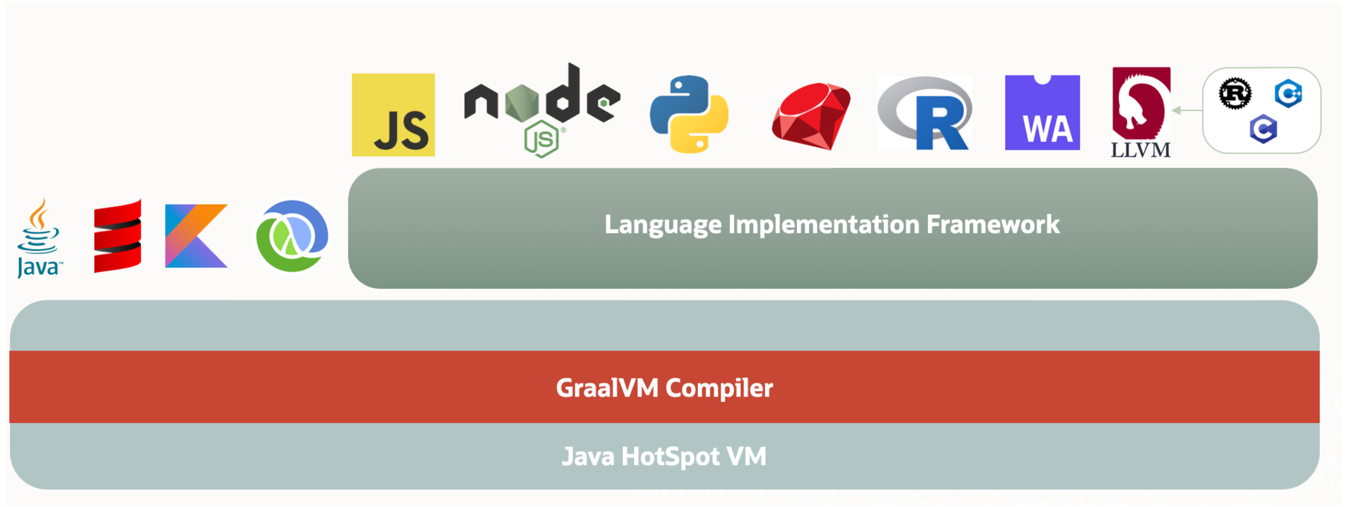
Smaller, faster and lower resource consumption

- Standalone executable of ahead-of-time compiled Java code
- Includes the application classes, classes from its dependencies, runtime library classes, and statically linked native code from JDK
- Runs without the need of a JVM, necessary components are included in a runtime system, called "Substrate VM"
- Specific to the OS and machine architecture for which it was compiled
- Requires fewer resources than regular Java applications running on a JVM

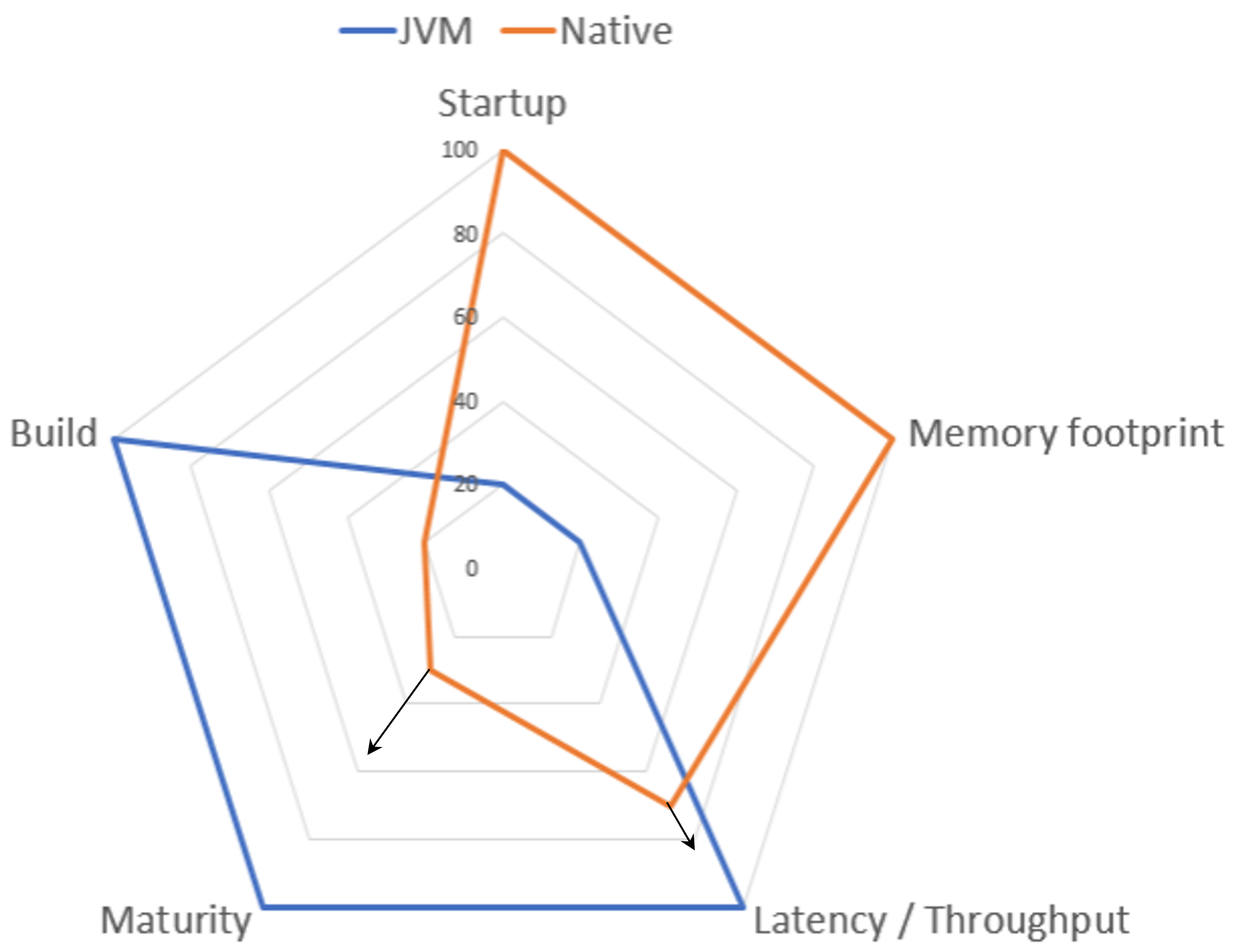
What Is GraalVM?

A high-performance JDK distribution

- Basic idea: One VM that can execute applications written in Java and other JVM languages while also providing runtimes for JavaScript, Ruby, Python, and a number of other popular languages
- Started by Oracle: https://graalvm.org/
- GraalVM's polyglot capabilities make it possible to mix multiple programming languages in a single application while eliminating any foreign language call costs
- Still in "early adopter" mode, but matures quickly



Tradeoffs Between JVM and Native Images



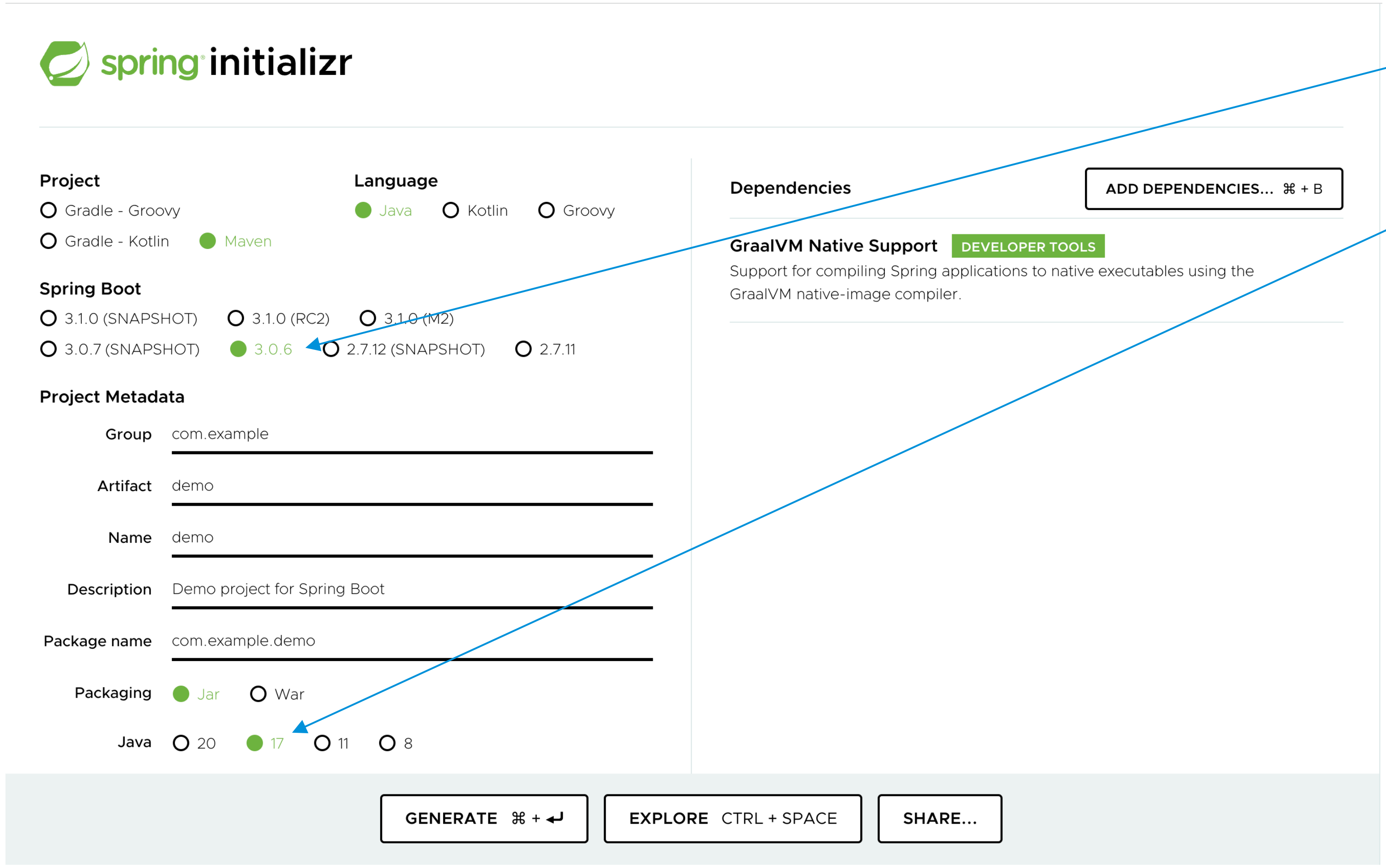
Key Differences Between JVM and GraalVM Native Image Platform

- 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

Spring Boot 3 added support for compiling Spring applications to lightweight native images using the GraalVM native-image compiler!

Get Started With Native Images in Spring Boot 3

Create a new project on start.spring.io or use an existing one



spring initializr

Project
- Gradle - Groovy
- Gradle - Kotlin
- [✓ Maven]

Language
- [✓ Java]
- Kotlin
- Groovy

Spring Boot
- 3.1.0 (SNAPSHOT)
- 3.1.0 (RC2)
- 3.1.0 (M2)
- 3.0.7 (SNAPSHOT)
- [✓ 3.0.6]
- 2.7.12 (SNAPSHOT)
- 2.7.11

Project Metadata

Group	com.example
Artifact	demo
Name	demo
Description	Demo project for Spring Boot
Package name	com.example.demo
Packaging	[✓ Jar] War
Java	20 [✓ 17] 11 8

Dependencies

GraalVM Native Support DEVELOPER TOOLS
Support for compiling Spring applications to native executables using the GraalVM native-image compiler.

- Choose Spring Boot 3 as your project's parent
- Spring Boot 3 requires Java 17 a a minimum version

Get Started With Native Images in Spring Boot 3

Create a new project on start.spring.io or use an existing one

```xml
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
  </dependency>

  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>

<build>
  <plugins>
    <plugin>
      <groupId>org.graalvm.buildtools</groupId>
      <artifactId>native-maven-plugin</artifactId>
    </plugin>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
```

- Add the native build tools plugin

Two Options to Build a Native Image in Spring Boot 3

Via Cloud-Native-Buildpacks

- Configure your build to use the Paketo Buildpacks
- Tell the buildpack to produce a native image
- The result is a small container image with the compiled native executable inside
- No local GraalVM installation needed
- Super easy to use
- Run `mvn -Pnative spring-boot:build-image` to create a container with your application

Via GraalVM native image Maven plugin

- Configure your build to compile to a native executable
- Produces a native executable for the platform you are running on
- Requires GraalVM locally installed
- Also super easy to use
- Run `mvn -Pnative native:compile` to build the application.

Can I keep the great Spring UX when going Native?

Use JVM locally, delegate Native Builds to CI/CD…

- Delegate Native Image building to the CI/CD pipeline
- Unchanged Developer UX in your IDE, with the same JVM. Sometimes it’s useful for tests using the AOT-generated initialization code, which is possible by setting the `spring.aot.enabled` system property to `true`

Resources

- Documentation: https://docs.spring.io/spring-boot/docs/current/reference/html/native-image.htm
- Wiki with known limitations: https://github.com/spring-projects/spring-boot/wiki/Spring-Boot-with-GraalVM
- Baeldung “Native Images with Spring Boot and GraalVM”: https://www.baeldung.com/spring-native-intro