Lean Spring Boot

Applications for The Cloud

Patrick Baumgartner
42talents GmbH, Zürich, Switzerland

@patbaumgartner
patrick.baumgartner@42talents.com

Abstract

Lean Spring Boot Applications for The Cloud

With the starters, Spring-Boot offers a functionality that allows you to set up a new software project with little effort and start programming right away. You don't have to worry about the dependencies since the "right" ones are already pre-configured. But how can you, for example, optimize the start-up times and reduce the memory footprint and thus better prepare the application for the cloud?

In this talk, we will go into Spring-Boot features like Spring AOT, classpath exclusions, lazy spring beans, actuator, JMX. In addition, we also look at switching to a different JVM and other tools.

Let's make Spring Boot great again!

Lean Spring Boot

Applications for The Cloud

Patrick Baumgartner

42talents GmbH, Zürich, Switzerland

@patbaumgartner

patrick.baumgartner@42talents.com

⚠️ WARNING:

Numbers shown in this talk are **not** based on **real data** but **only estimates** and **assumptions** made by the **author** for **educational purposes** only.

Introduction

Patrick Baumgartner

Technical Agile Coach @ 42talents

My focus is on the development of software solutions with humans.

Coaching, Architecture, Development, Reviews, and Training.

Lecturer @ Zürcher Fachhochschule für Angewandte Wissenschaften ZHAW

@patbaumgartner

What is the problem?
Why this talk?

JAVA 😉 & Spring Boot ❤️

Requirements
When Choosing a Cloud

- How many vCPUs per server are required for my application?
- How much RAM do I need?
- How much storage is necessary?
- Which technology stack should I use?

Considerations

Resources

- CPU & RAM not linearly scalable
- Image Size & Network Bandwidth

Scalability

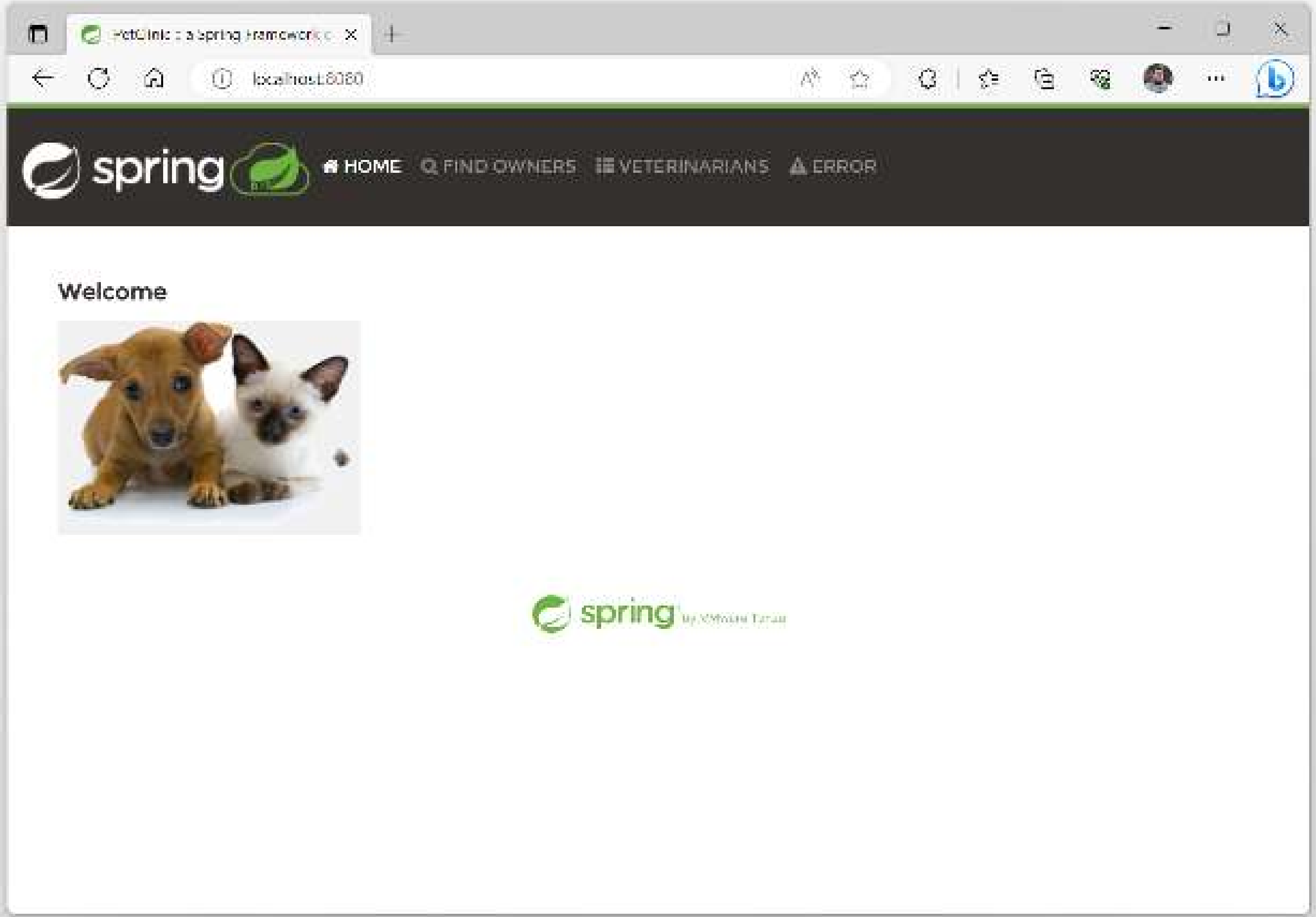
- Fast Startup
- Graceful Shutdown
- Throughput
- Latency

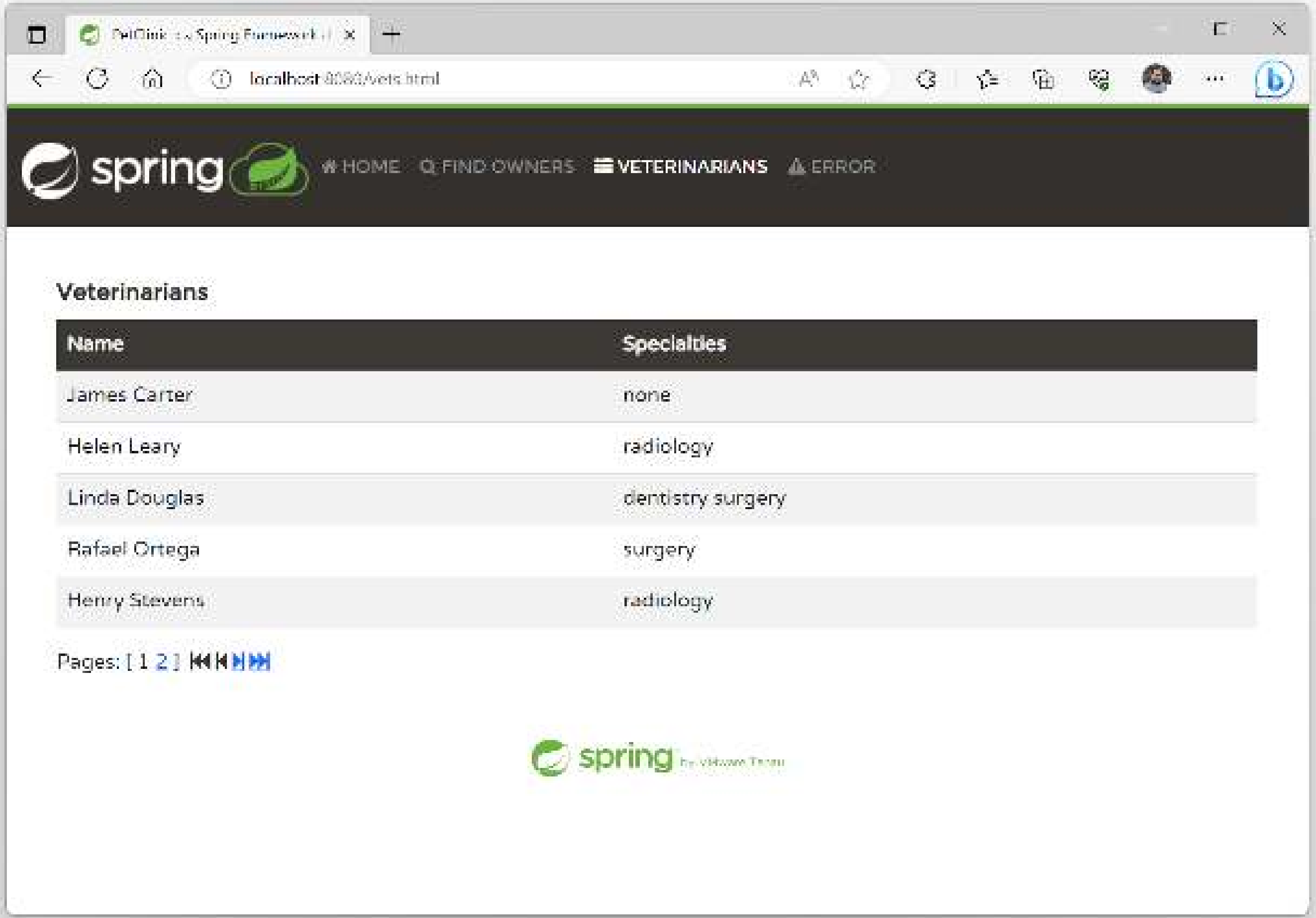
Agenda

Agenda

- Spring PetClinic & Baseline for Comparison
- Java Optimizations
- Spring Boot Optimizations
- Application Optimizations
- Other Runtimes
- Conclusions
- A Few Simple Optimizations Applied (OpenJDK Example)

Spring PetClinic


PetClinic :: a Spring Framework d
localhost:8080
HOME
FIND OWNERS
VETERINARIANS
ERROR
Welcome



Veterinarians

Name	Specialties
James Carter	none
Helen Leary	radiology
Linda Douglas	dentistry surgery
Rafael Ortega	surgery
Henry Stevens	radiology

Pages: [1 2]

Spring Petclinic Community

- spring-framework-petclinic
- spring-petclinic-angular(js)
- spring-petclinic-rest
- spring-petclinic-graphql
- spring-petclinic-microservices
- spring-petclinic-data-jdbc
- spring-petclinic-cloud
- spring-petclinic-mustache
- spring-petclinic-kotlin
- spring-petclinic-reactive
- spring-petclinic-hilla
- spring-petclinic-angularjs
- spring-petclinic-vaadin-flow
- spring-petclinic-reactjs
- spring-petclinic-htmx
- spring-petclinic-istio
- …

Projects on GitHub: https://github.com/spring-petclinic

NO!

The official Spring PetClinic! 🐾🏥

Which is based on Spring Boot, Caffeine, Thymeleaf, Spring Data JPA, H2 and Spring MVC …

Project on GitHub: https://github.com/spring-projects/spring-petclinic

Optimizing Experiments

Baseline

Technology Stack

- OCI Container made with Buildpacks
- Java JDK 17 LTS
- Spring Boot 3.2.0
- DB Migration with SQL Scripts

Examination

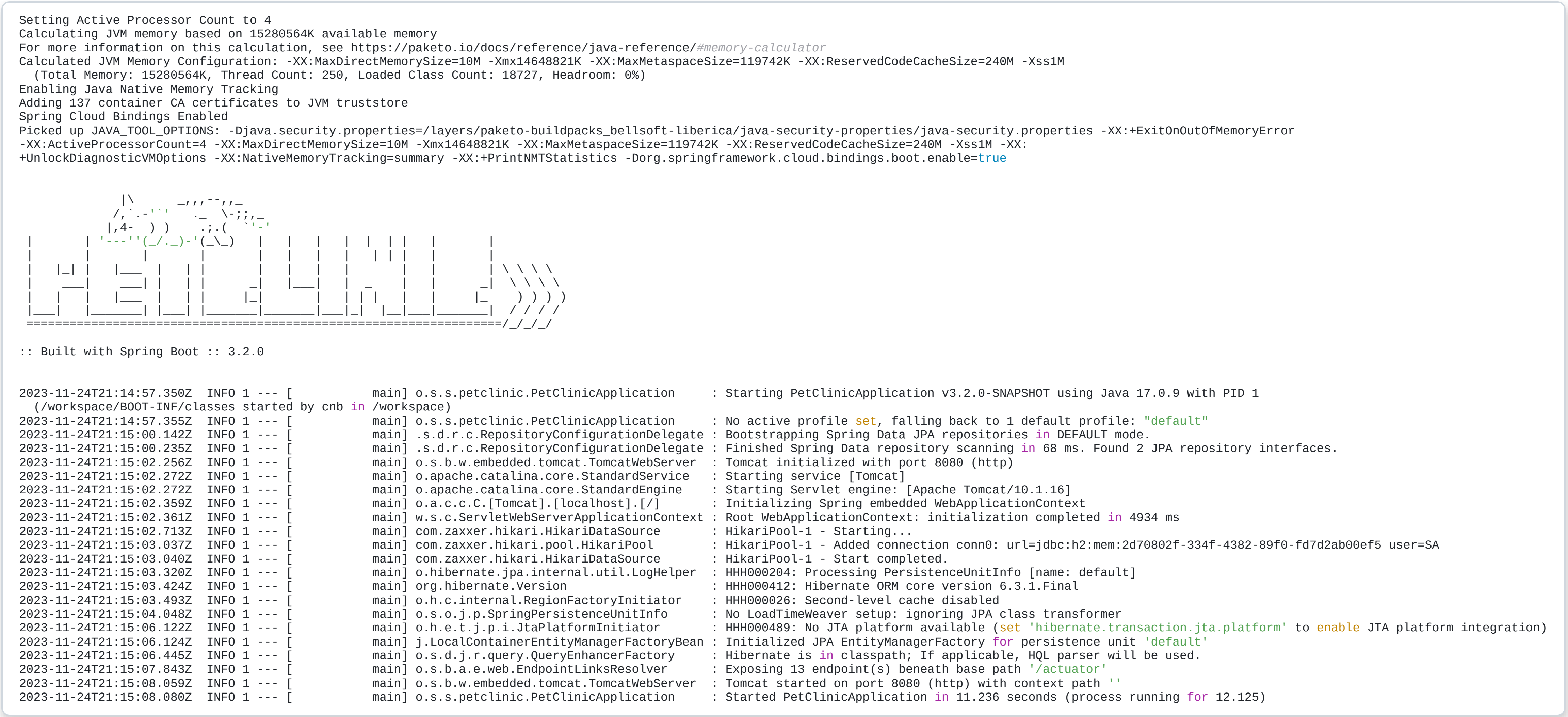
- Build Time
- Startup Time
- Resource Usage
- Container Image Size
- Throughput

Buildpacks.io

paketo
buildpacks

paketo BUILDPACKS

Your app,
in your favorite language,
ready to run in the cloud



```
Setting Active Processor Count to 4
Calculating JVM memory based on 15280564K available memory
For more information on this calculation, see https://paketo.io/docs/reference/java-reference/#memory-calculator
Calculated JVM Memory Configuration: -XX:MaxDirectMemorySize=10M -Xmx14648821K -XX:MaxMetaspaceSize=119742K -XX:ReservedCodeCacheSize=240M -Xss1M
  (Total Memory: 15280564K, Thread Count: 250, Loaded Class Count: 18727, Headroom: 0%)
Enabling Java Native Memory Tracking
Adding 137 container CA certificates to JVM truststore
Spring Cloud Bindings Enabled
Picked up JAVA_TOOL_OPTIONS: -Djava.security.properties=/layers/paketo-buildpacks_bellsoft-liberica/java-security-properties/java-security.properties -XX:+ExitOnOutOfMemoryError
-XX:ActiveProcessorCount=4 -XX:MaxDirectMemorySize=10M -Xmx14648821K -XX:MaxMetaspaceSize=119742K -XX:ReservedCodeCacheSize=240M -Xss1M -XX:
+UnlockDiagnosticVMOptions -XX:NativeMemoryTracking=summary -XX:+PrintNMTStatistics -Dorg.springframework.cloud.bindings.boot.enable=true


              |\      _,,,--,,_
             /,`.-'`'   ._  \-;;,_
  _______ __|,4-  ) )_   .;.(__`'-'__     ___ __    _ ___ _______
 |       | '---''(_/._)-'(_\_)   |   |   |   | |  | |   |       |
 |    _  |    ___|_     _|       |   |   |   |   |_| |   |       | __ _ _
 |   |_| |   |___  |   | |       |   |   |   |       |   |       | \ \ \ \
 |    ___|    ___| |   | |       |   |___|   |  _    |   |      _|  \ \ \ \
 |   |   |   |___  |   | |       |      _|   | | |   |   |     |_    ) ) ) )
 |___|   |_______| |___| |_______|_______|___|_|  |__|___|_______|  / / / /
 ==================================================================/_/_/_/

:: Built with Spring Boot :: 3.2.0


2023-11-24T21:14:57.350Z  INFO 1 --- [           main] o.s.s.petclinic.PetClinicApplication     : Starting PetClinicApplication v3.2.0-SNAPSHOT using Java 17.0.9 with PID 1
  (/workspace/BOOT-INF/classes started by cnb in /workspace)
2023-11-24T21:14:57.355Z  INFO 1 --- [           main] o.s.s.petclinic.PetClinicApplication     : No active profile set, falling back to 1 default profile: "default"
2023-11-24T21:15:00.142Z  INFO 1 --- [           main] .s.d.r.c.RepositoryConfigurationDelegate : Bootstrapping Spring Data JPA repositories in DEFAULT mode.
2023-11-24T21:15:00.235Z  INFO 1 --- [           main] .s.d.r.c.RepositoryConfigurationDelegate : Finished Spring Data repository scanning in 68 ms. Found 2 JPA repository interfaces.
2023-11-24T21:15:02.256Z  INFO 1 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port 8080 (http)
2023-11-24T21:15:02.272Z  INFO 1 --- [           main] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2023-11-24T21:15:02.272Z  INFO 1 --- [           main] o.apache.catalina.core.StandardEngine    : Starting Servlet engine: [Apache Tomcat/10.1.16]
2023-11-24T21:15:02.359Z  INFO 1 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext
2023-11-24T21:15:02.361Z  INFO 1 --- [           main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 4934 ms
2023-11-24T21:15:02.713Z  INFO 1 --- [           main] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Starting...
2023-11-24T21:15:03.037Z  INFO 1 --- [           main] com.zaxxer.hikari.pool.HikariPool        : HikariPool-1 - Added connection conn0: url=jdbc:h2:mem:2d70802f-334f-4382-89f0-fd7d2ab00ef5 user=SA
2023-11-24T21:15:03.040Z  INFO 1 --- [           main] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Start completed.
2023-11-24T21:15:03.320Z  INFO 1 --- [           main] o.hibernate.jpa.internal.util.LogHelper  : HHH000204: Processing PersistenceUnitInfo [name: default]
2023-11-24T21:15:03.424Z  INFO 1 --- [           main] org.hibernate.Version                    : HHH000412: Hibernate ORM core version 6.3.1.Final
2023-11-24T21:15:03.493Z  INFO 1 --- [           main] o.h.c.internal.RegionFactoryInitiator    : HHH000026: Second-level cache disabled
2023-11-24T21:15:04.048Z  INFO 1 --- [           main] o.s.o.j.p.SpringPersistenceUnitInfo      : No LoadTimeWeaver setup: ignoring JPA class transformer
2023-11-24T21:15:06.122Z  INFO 1 --- [           main] o.h.e.t.j.p.i.JtaPlatformInitiator       : HHH000489: No JTA platform available (set 'hibernate.transaction.jta.platform' to enable JTA platform integration)
2023-11-24T21:15:06.124Z  INFO 1 --- [           main] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit 'default'
2023-11-24T21:15:06.445Z  INFO 1 --- [           main] o.s.d.j.r.query.QueryEnhancerFactory     : Hibernate is in classpath; If applicable, HQL parser will be used.
2023-11-24T21:15:07.843Z  INFO 1 --- [           main] o.s.b.a.e.web.EndpointLinksResolver      : Exposing 13 endpoint(s) beneath base path '/actuator'
2023-11-24T21:15:08.059Z  INFO 1 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port 8080 (http) with context path ''
2023-11-24T21:15:08.080Z  INFO 1 --- [           main] o.s.s.petclinic.PetClinicApplication     : Started PetClinicApplication in 11.236 seconds (process running for 12.125)
```

1000x Better **than your regular Dockerfile** 📶 …

… **more** **secure** 🔒 **and maintained by the Buildpacks community.**

See also: https://buildpacks.io/ and https://www.cncf.io/projects/buildpacks/

Startup Reporting

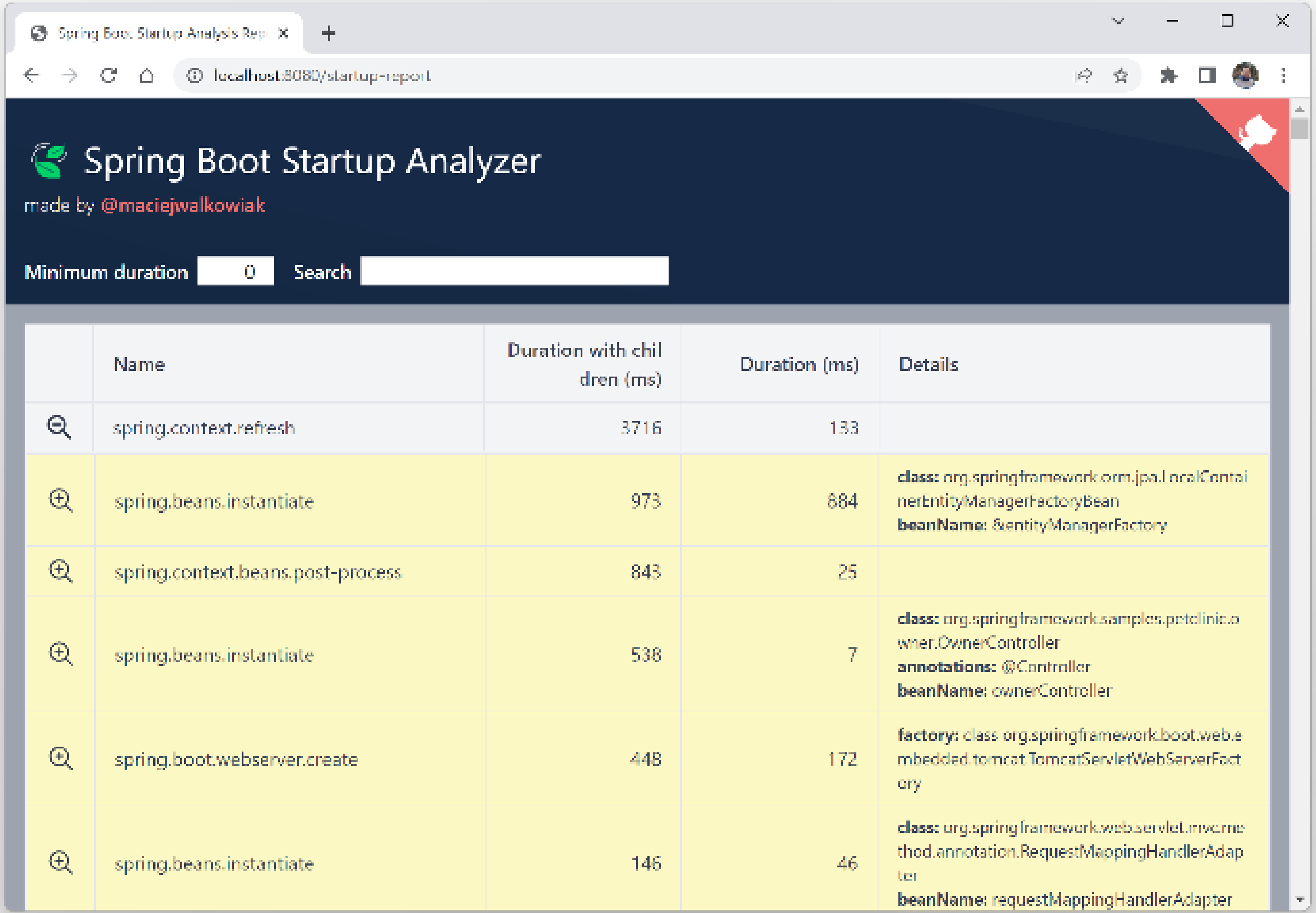
Spring Boot Startup Report

By Maciej Walkowiak

- Startup report available in runtime as an interactive HTML page
- Generating startup reports in integration tests
- Flame chart for timings
- Search by class or annotation

```xml
<dependency>
    <groupId>com.maciejwalkowiak.spring</groupId>
    <artifactId>spring-boot-startup-report</artifactId>
    <version>0.2.0</version>
    <optional>true</optional>
</dependency>
```

https://github.com/maciejwalkowiak/spring-boot-startup-report


Spring Boot Startup Analysis Rep

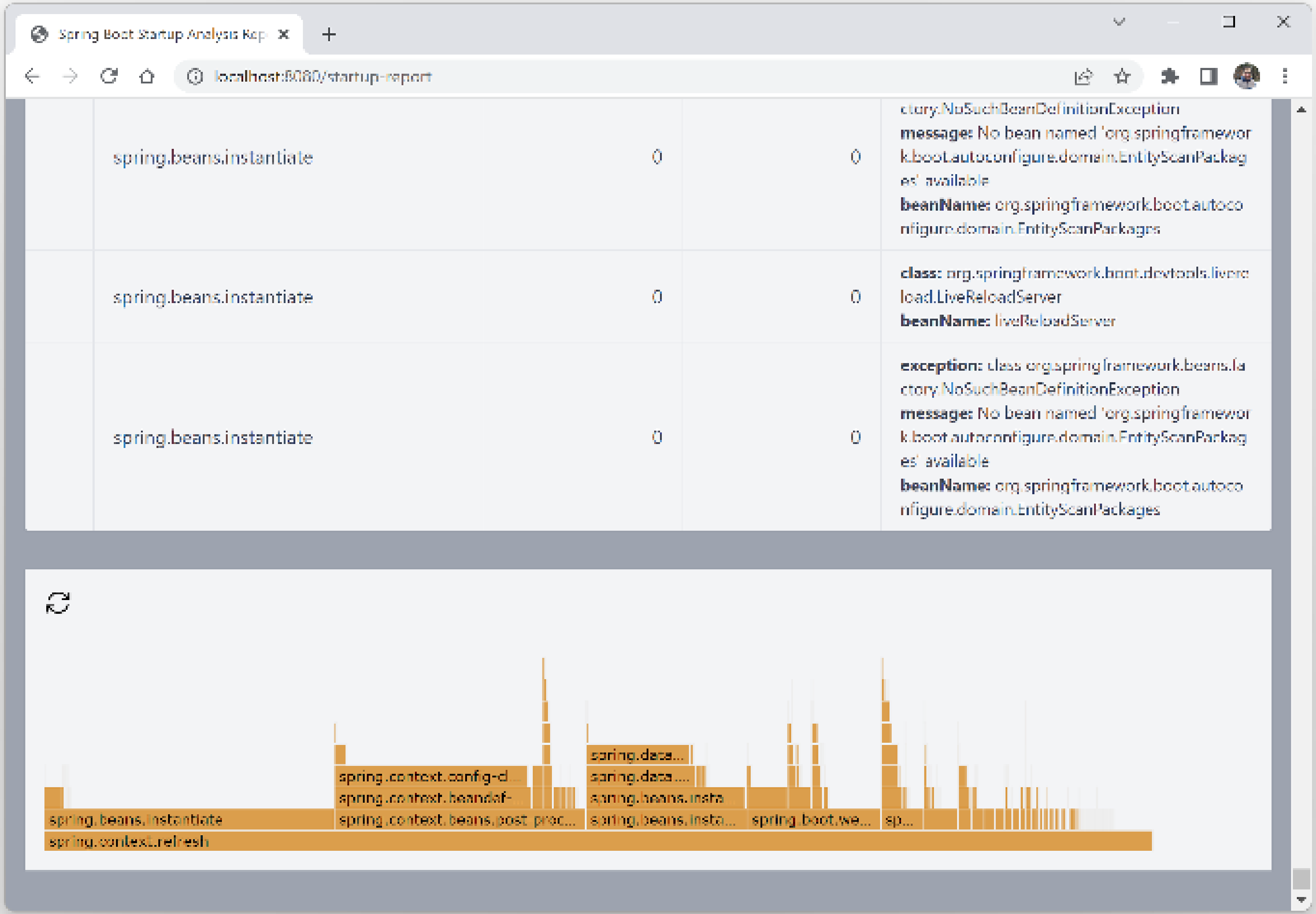
localhost:8080/startup-report

Spring Boot Startup Analyzer

made by @maciejwalkowiak

Minimum duration 0 Search

	Name	Duration with children (ms)	Duration (ms)	Details
	spring.context.refresh	3716	133	
	spring.beans.instantiate	973	884	**class:** org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean **beanName:** &entityManagerFactory
	spring.context.beans.post-process	843	25	
	spring.beans.instantiate	538	7	**class:** org.springframework.samples.petclinic.owner.OwnerController **annotations:** @Controller **beanName:** ownerController
	spring.boot.webserver.create	448	172	**factory:** class org.springframework.boot.web.embedded.tomcat.TomcatServletWebServerFactory
	spring.beans.instantiate	146	46	**class:** org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter **beanName:** requestMappingHandlerAdapter


Spring Boot Startup Analysis Rep
localhost:8080/startup-report

	spring.beans.instantiate	0	0	ctory.NoSuchBeanDefinitionException **message:** No bean named 'org.springframework.boot.autoconfigure.domain.EntityScanPackages' available **beanName:** org.springframework.boot.autoconfigure.domain.EntityScanPackages
	spring.beans.instantiate	0	0	**class:** org.springframework.boot.devtools.livereload.LiveReloadServer **beanName:** liveReloadServer
	spring.beans.instantiate	0	0	**exception:** class org.springframework.beans.factory.NoSuchBeanDefinitionException **message:** No bean named 'org.springframework.boot.autoconfigure.domain.EntityScanPackages' available **beanName:** org.springframework.boot.autoconfigure.domain.EntityScanPackages

Benchmarks

Benchmarks

- Build
  - Maven build time
  - Artifact / Container Image size
- Startup
  - Startup time
  - Memory usage
- Throughput & Latency
  - `wrk2 -t4 -c200 -d60s -R2000 --latency <HOST>`
  - 1 min warmup, 1min measurement
  - Docker container with 4 vCPU and 1 GB RAM

No Optimizing - Baseline

No Optimizing - Baseline JDK 17

- Spring PetClinic (no adjustments)
- Bellsoft Liberica JDK 17.0.9
- Java Memory Calculator

```
sdk use java 17.0.9-librca

mvn spring-boot:build-image

docker run -p 8080:8080 -t spring-petclinic:3.2.0-SNAPSHOT
```

Build	Image	Startup	Initial RAM	Throughput	RAM	50%	75%	90%	99%	99.90%	99.99%
⏱️60s	349MB	4.813s	299MB	1994/s	465MB	7ms	13ms	19ms	52ms	109ms	164ms

No Optimizing - Baseline JDK 21

- Spring PetClinic (JDK 21 adjustments)
- Bellsoft Liberica JDK 21.0.1
- Java Memory Calculator

```
sdk use java 21.0.1-librca

mvn -Djava.version=21 spring-boot:build-image \
    -Dspring-boot.build-image.imageName=spring-petclinic:3.2.0-SNAPSHOT-jdk21

docker run -p 8080:8080 -t spring-petclinic:3.2.0-SNAPSHOT-jdk21
```

Build	Image	Startup	Initial RAM	Throughput	RAM	50%	75%	90%	99%	99.90%	99.99%
⏱️60s	349MB	4.813s	299MB	1994/s	465MB	7ms	13ms	19ms	52ms	109ms	164ms
61s	401MB	4.845s	307MB	1990/s	471MB	7ms	13ms	19ms	49ms	92ms	146ms

-noverify

-noverify

The verifier is turned off because some of the bytecode rewriting stretches the meaning of some of the bytecodes - in a way that doesn't bother the JVM, but doesn't bother the verifier.

Warning: Options `-Xverify:none` and `-noverify` were deprecated in JDK 13 and will likely be removed in a future release.

```
docker run -p 8080:8080 -e "JAVA_TOOL_OPTIONS=-noverify" \
  -t spring-petclinic:3.2.0-SNAPSHOT
```

Build	Image	Startup	Initial RAM	Throughput	RAM	50%	75%	90%	99%	99.90%	99.99%
⏱60s	349MB	4.813s	299MB	1994/s	465MB	7ms	13ms	19ms	52ms	109ms	164ms
-	-	4.25s	286MB	1993/s	460MB	7ms	12ms	19ms	51ms	104ms	161ms

-XX:TieredStopAtLevel=1

-XX:TieredStopAtLevel=1

The tiered compilation is enabled by default since Java 8. Unless explicitly specified, the JVM decides which JIT compiler to use based on our CPU. For multi-core processors or 64-bit VMs, the JVM will select C2.

To disable C2 and only use the C1 compiler with no profiling overhead, we can apply the `-XX:TieredStopAtLevel=1` parameter.

```
docker run -p 8080:8080 -e "JAVA_TOOL_OPTIONS=-XX:TieredStopAtLevel=1" \
  -t spring-petclinic:3.2.0-SNAPSHOT
```

It will slow down the JIT later at the expense of the saved startup time!

Build	Image	Startup	Initial RAM	T...	RAM	50%	75%	90%	99%	99.90%	99.99%
⏱60s	349MB	4.813s	299MB	1994/s	465MB	7ms	13ms	19ms	52ms	109ms	164ms
-	-	4.084s	213MB	1414/s	331MB	9955ms	13542ms	16008ms	19063ms	20895ms	21773ms

-XX:+UseZGC

The Z Garbage Collector (ZGC) is a scalable low latency garbage collector. ZGC performs all expensive work concurrently, without stopping the execution of application threads for more than 10ms, which makes is suitable for applications that require low latency and/or use a very large heap (multi-terabytes).

```
docker run -p 8080:8080 -e "JAVA_TOOL_OPTIONS=-XX:+UseZGC" \
  -t spring-petclinic:3.1.0-SNAPSHOT
```

See also: https://wiki.openjdk.org/display/zgc/Main

Build	Image	Startup	Initial RAM	Throughput	RAM	50%	75%	90%	99%	99.90%	99.99%
⏱60s	349MB	4.813s	299MB	1994/s	465MB	7ms	13ms	19ms	52ms	109ms	164ms
-	-	4.807s	1088MB	1995/s	1537MB	28ms	67ms	123ms	324ms	545ms	751ms

VM Options Explorer

VM Options Explorer

The VM Options Explorer is a tool that helps to find the best combination of JVM options for a specific application. The JVM has more than 800 options, and it is not easy to find the right ones for a specific application.

https://chriswhocodes.com

Lazy Spring Beans

Lazy Spring Beans (1)

Configure lazy initialization across the whole application. A Spring Boot property makes all Beans lazy by default and only initializes them when they are needed. `@Lazy` can be used to override this behavior with e.g. `@Lazy(false)`.

```
docker run -p 8080:8080  \
  -e spring.main.lazy-initialization=true  \
  -e spring.data.jpa.repositories.bootstrap-mode=lazy \
  -t spring-petclinic:3.2.0-SNAPSHOT
```

Build	Image	Startup	Initial RAM	Throughput	RAM	50%	75%	90%	99%	99.90%	99.99%
⏱60s	349MB	4.813s	299MB	1994/s	465MB	7ms	13ms	19ms	52ms	109ms	164ms
-	-	2.836s	254MB	1995/s	462MB	7ms	12ms	18ms	48ms	106ms	161ms

Lazy Spring Beans (2)

Pros

- Faster startup useful in cloud environments
- Application startup is a CPU-intensive task. Spreading the load over time

Cons

- The initial requests may take more time
- Class loading issues and misconfigurations unnoticed at startup
- Beans creation errors only be found at the time of loading the bean

No Spring Boot Actuators

No Spring Boot Actuators

Don’t use actuators if you can afford not to. 😊

- No. of Spring Beans
  - Spring Pet Clinic with Actuators: 449
  - Spring Pet Clinic no Actuators: 274 🔥

```
sdk use java 17.0.9-librca

mvn spring-boot:build-image

docker run -p 8080:8080 -t spring-petclinic-no-actuator:3.2.0-SNAPSHOT
```

Build	Image	Startup	Initial RAM	Throughput	RAM	50%	75%	90%	99%	99.90%	99.99%
⏱60s	349MB	4.813s	299MB	1994/s	465MB	7ms	13ms	19ms	52ms	109ms	164ms
57s	346MB	3.593s	270MB	1991/s	435MB	7ms	12ms	18ms	47ms	92ms	138ms

Fixing Spring Boot Config Location

Fixing Spring Boot Config Location

Fix the location of the Spring Boot config file(s).
Considered in the following order (`application.properties` and `YAML` variants):

- Application properties packaged inside your jar
- Profile-specific application properties packaged inside your jar
- Application properties outside of your packaged jar
- Profile-specific application properties outside of your packaged jar

```
docker run -p 8080:8080 -e spring.config.location=classpath:application.properties \
  -t spring-petclinic:3.2.0-SNAPSHOT
```

Build	Image	Startup	Initial RAM	Throughput	RAM	50%	75%	90%	99%	99.90%	99.99%
⏱60s	349MB	4.813s	299MB	1994/s	465MB	7ms	13ms	19ms	52ms	109ms	164ms
-	-	4.786s	316MB	1993/s	475MB	6ms	12ms	19ms	47ms	103ms	155ms

Disabling JMX

Disabling JMX

JMX is `spring.jmx.enabled=false` by default in Spring Boot since 2.2.0 and later. Setting `BPL_JMX_ENABLED=true` and `BPL_JMX_PORT=9999` on the container will add the following arguments to the `java` command.

```
-Djava.rmi.server.hostname=127.0.0.1
-Dcom.sun.management.jmxremote.authenticate=false
-Dcom.sun.management.jmxremote.ssl=false
-Dcom.sun.management.jmxremote.port=9999
-Dcom.sun.management.jmxremote.rmi.port=9999
```

```
docker run -p 8080:8080 -p 9999:9999 \
  -e BPL_JMX_ENABLED=false \
  -e BPL_JMX_PORT=9999 \
  -e spring.jmx.enabled=false \
  -t spring-petclinic:3.2.0-SNAPSHOT
```

I ❤️

Spring Boot 🍃 & Buildpacks

Dependency Cleanup

Dependency Cleanup (2)

DepClean detects and removes all the unused dependencies declared in the `pom.xml` file of a project or imported from its parent. It does not touch the original `pom.xml` file.

```
<plugin>
  <groupId>se.kth.castor</groupId>
  <artifactId>depclean-maven-plugin</artifactId>
  <version>2.0.6</version>
  <executions>
    <execution>
      <goals>
        <goal>depclean</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

```
mvn se.kth.castor:depclean-maven-plugin:2.0.6:depclean -DfailIfUnusedDirect=true -DignoreScopes=provided,test,runtime,system,import
```

```
-------------------------------------------------------
[INFO] Starting DepClean dependency analysis
-------------------------------------------------------
 D E P C L E A N   A N A L Y S I S   R E S U L T S
-------------------------------------------------------
USED DIRECT DEPENDENCIES [9]:
        com.h2database:h2:2.2.224:runtime (2 MB)
        com.mysql:mysql-connector-j:8.1.0:test (2 MB)
        org.postgresql:postgresql:42.6.0:test (1 MB)
        com.github.ben-manes.caffeine:caffeine:3.1.8:compile (868 KB)
        jakarta.xml.bind:jakarta.xml.bind-api:4.0.1:compile (126 KB)
        org.springframework.boot:spring-boot-testcontainers:3.2.0:test (104 KB)
        javax.cache:cache-api:1.1.1:compile (50 KB)
        ...
USED TRANSITIVE DEPENDENCIES [89]:
        org.testcontainers:testcontainers:1.19.3:test (16 MB)
        org.hibernate.orm:hibernate-core:6.3.1.Final:compile (10 MB)
        net.bytebuddy:byte-buddy:1.14.10:runtime (4 MB)
        org.apache.tomcat.embed:tomcat-embed-core:10.1.16:compile (3 MB)
        org.aspectj:aspectjweaver:1.9.20.1:compile (2 MB)
        com.github.docker-java:docker-java-transport-zerodep:3.3.4:test (1 MB)
        org.springframework.boot:spring-boot-autoconfigure:3.2.0:compile (1 MB)
        net.java.dev.jna:jna:5.13.0:test (1 MB)
        org.springframework:spring-web:6.1.1:compile (1 MB)
        org.springframework:spring-core:6.1.1:compile (1 MB)
        com.fasterxml.jackson.core:jackson-databind:2.15.3:test (1 MB)
        org.springframework.boot:spring-boot:3.2.0:compile (1 MB)
        ...
USED INHERITED DIRECT DEPENDENCIES [0]:
USED INHERITED TRANSITIVE DEPENDENCIES [0]:
POTENTIALLY UNUSED DIRECT DEPENDENCIES [11]:
        org.webjars.npm:bootstrap:5.3.2:compile (1 MB)
        org.webjars.npm:font-awesome:4.7.0:compile (665 KB)
        org.springframework.boot:spring-boot-devtools:3.2.0:test (198 KB)
        org.springframework.boot:spring-boot-docker-compose:3.2.0:test (177 KB)
        org.springframework.boot:spring-boot-starter-web:3.2.0:compile (4 KB)
        org.springframework.boot:spring-boot-starter-test:3.2.0:test (4 KB)
        ...
POTENTIALLY UNUSED TRANSITIVE DEPENDENCIES [11]:
        org.attoparser:attoparser:2.0.7.RELEASE:compile (240 KB)
        org.thymeleaf:thymeleaf-spring6:3.1.2.RELEASE:compile (184 KB)
        org.unbescape:unbescape:1.1.6.RELEASE:compile (169 KB)
        org.awaitility:awaitility:4.2.0:test (93 KB)
        com.google.errorprone:error_prone_annotations:2.21.1:compile (16 KB)
        org.slf4j:jul-to-slf4j:2.0.9:compile (6 KB)
        org.springframework.boot:spring-boot-starter-tomcat:3.2.0:compile (4 KB)
        org.springframework.boot:spring-boot-starter-jdbc:3.2.0:compile (4 KB)
        org.springframework.boot:spring-boot-starter-aop:3.2.0:compile (4 KB)
        org.springframework.boot:spring-boot-starter-logging:3.2.0:compile (4 KB)
        org.hamcrest:hamcrest-core:2.2:test (1 KB)
POTENTIALLY UNUSED INHERITED DIRECT DEPENDENCIES [0]:
POTENTIALLY UNUSED INHERITED TRANSITIVE DEPENDENCIES [0]:
```

Dependency Cleanup (2)

But there are some challenges:

- Component & Entity Scanning through Classpath Scanning
- Spring Boot uses `META-INF/spring-boot/org.springframework.boot.autoconfigure.AutoConfiguration.imports`
- Spring Context Indexer uses `META-INF/spring.components`
- Spring XML configuration and `web.xml`

```
sdk use java 17.0.9-librca

mvn spring-boot:build-image

docker run -p 8080:8080 -t spring-petclinic-depclean:3.2.0-SNAPSHOT
```

Build	Image	Startup	Initial RAM	Throughput	RAM	50%	75%	90%	99%	99.90%	99.99%
⏱60s	349MB	4.813s	299MB	1994/s	465MB	7ms	13ms	19ms	52ms	109ms	164ms
67s	340MB	3.45s	258MB	1995/s	435MB	6ms	11ms	17ms	45ms	97ms	149ms

Ahead-of-Time Processing (AOT)

Ahead-of-Time Processing (AOT) (1)

Spring AOT is a process that analyzes your application at build time and generates an optimized version of it.

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

```
<plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
    <executions>
        <execution>
            <id>process-aot</id>
            <goals>
                <goal>process-aot</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

Ahead-of-Time Processing (AOT) (2)

We are creating a new container image with the AOT-processed application.

```
sdk use java 17.0.9-librca

mvn spring-boot:build-image

docker run -e spring.aot.enabled=true -p 8080:8080 -t spring-petclinic-aot:3.2.0-SNAPSHOT
```

Build	Image	Startup	Initial RAM	Throughput	RAM	50%	75%	90%	99%	99.90%	99.99%
⏱60s	349MB	4.813s	299MB	1994/s	465MB	7ms	13ms	19ms	52ms	109ms	164ms
66s	350MB	4.798s	299MB	1996/s	468MB	7ms	12ms	19ms	48ms	99ms	160ms

JLink

JLink (1)

Jlink assembles and optimizes a set of modules and their dependencies into a custom runtime image for your application.

```
$ jlink \
  --add-modules java.base, ... \
  --strip-debug \
  --no-man-pages \
  --no-header-files \
  --compress=2 \
  --output /javaruntime
```

```
$ /javaruntime/bin/java HelloWorld
Hello, World!
```

JLink (2)

```
<plugin>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-maven-plugin</artifactId>
    <configuration>
        <image>
            <env>
                <BP_JVM_JLINK_ENABLED>true</BP_JVM_JLINK_ENABLED>
            </env>
        </image>
    </configuration>
</plugin>
```

Build	Image	Startup	Initial RAM	Throughput	RAM	50%	75%	90%	99%	99.90%	99.99%
⏱60s	349MB	4.813s	299MB	1994/s	465MB	7ms	13ms	19ms	52ms	109ms	164ms
74s	282MB	4.881s	306MB	1991/s	476MB	8ms	14ms	21ms	49ms	105ms	169ms

```
sdk

mvn spring-boot:build-image

docker run -p 8080:8080 -t spring-petclinic-jlink:3.2.0-SNAPSHOT
```

JLink (3)

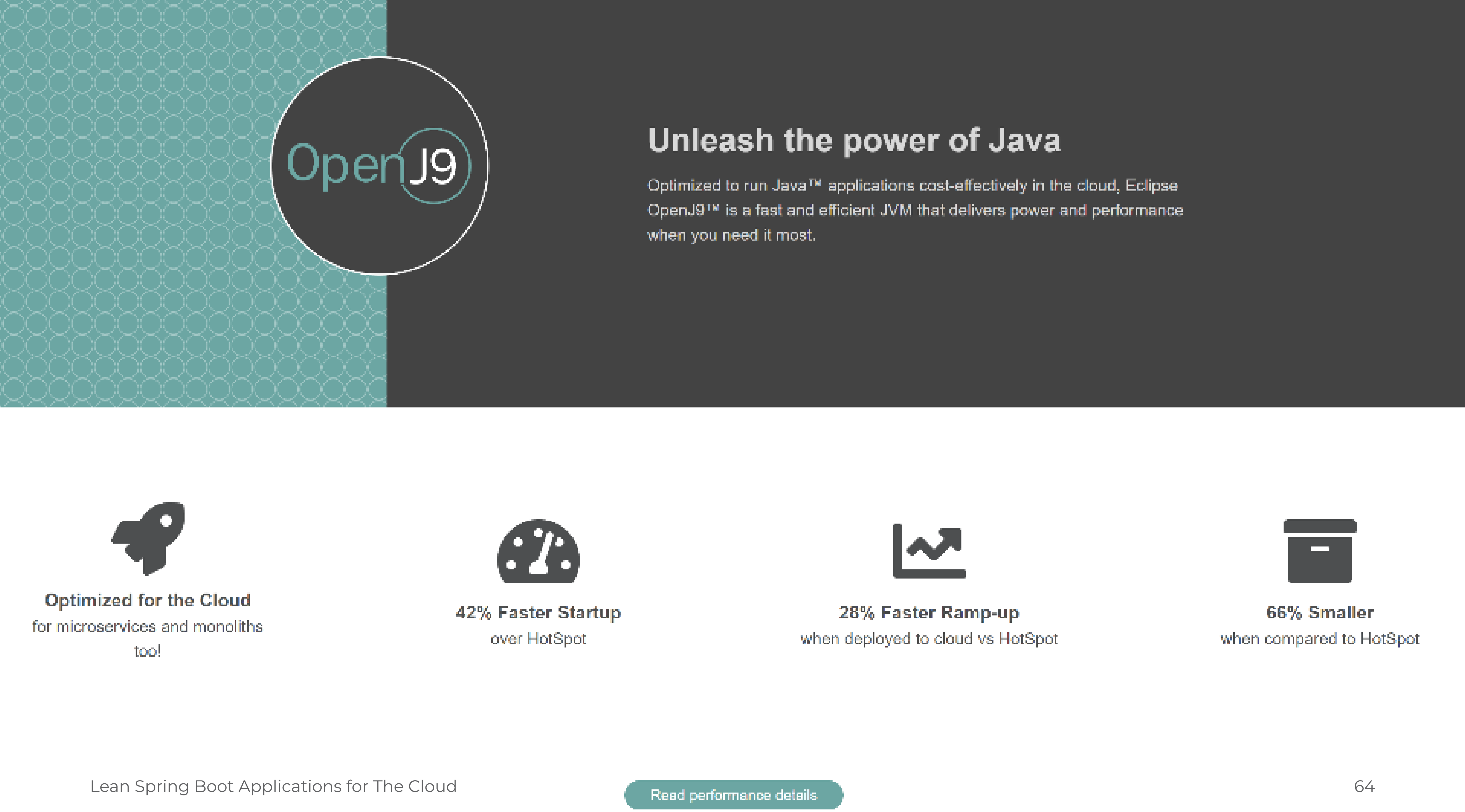
```xml
<plugin>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-maven-plugin</artifactId>
    <configuration>
        <image>
            <env>
                <BP_JVM_JLINK_ENABLED>true</BP_JVM_JLINK_ENABLED>
                <BP_JVM_JLINK_ARGS>--add-modules jdk.management.agent,java.base,java.logging,
                java.xml,jdk.unsupported,java.sql,java.naming,java.desktop,java.management,
                java.security.jgss,java.instrument --compress=2 --no-header-files --no-man-pages
                --strip-debug</BP_JVM_JLINK_ARGS>
            </env>
        </image>
    </configuration>
</plugin>
```

Build	Image	Startup	Initial RAM	Throughput	RAM	50%	75%	90%	99%	99.90%	99.99%
⏱️60s	349MB	4.813s	299MB	1994/s	465MB	7ms	13ms	19ms	52ms	109ms	164ms
71s	269MB	4.852s	313MB	1997/s	478MB	7ms	13ms	19ms	45ms	90ms	149ms

I ❤️

Spring Boot 🍃 & Buildpacks

Eclipse OpenJ9



Unleash the power of Java

Optimized to run Java™ applications cost-effectively in the cloud, Eclipse OpenJ9™ is a fast and efficient JVM that delivers power and performance when you need it most.

Optimized for the Cloud
for microservices and monoliths too!

42% Faster Startup
over HotSpot

28% Faster Ramp-up
when deployed to cloud vs HotSpot

66% Smaller
when compared to HotSpot

Eclipse OpenJ9

```xml
<plugin>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-maven-plugin</artifactId>
  <configuration>
    <image>
        <buildpacks>
            <buildpack>gcr.io/paketo-buildpacks/eclipse-openj9:latest</buildpack>
            <!-- Used to inherit all the other buildpacks -->
            <buildpack>gcr.io/paketo-buildpacks/java:latest</buildpack>
        </buildpacks>
    </image>
  </configuration>
</plugin>
```

Build	Image	Startup	Initial RAM	Throughput	RAM	50%	75%	90%	99%	99.90%	99.99%
⏱60s	349MB	4.813s	299MB	1994/s	465MB	7ms	13ms	19ms	52ms	109ms	164ms
73s	340MB	8.068s	176MB	1996/s	363MB	11ms	18ms	27ms	70ms	144ms	224ms

```
sdk

mvn spring-boot:build-image

docker run -p 8080:8080 -t spring-petclinic-custom-jvm-openj9:3.2.0-SNAPSHOT
```

Eclipse OpenJ9 Optimized

Eclipse OpenJ9 Optimized Virtualized

When `-Xtune:virtualized` is used in conjunction with the `-Xshareclasses` option, the JIT compiler is more aggressive with its use of AOT-compiled code compared to setting only `-Xshareclasses`. This action provides additional CPU savings during application start-up and ramp-up but comes at the expense of an additional small loss in throughput.

```
docker run -p 8080:8080 -e "JAVA_TOOL_OPTIONS=-XX:+IgnoreUnrecognizedVMOptions -XX:+UseContainerSupport
-XX:+IdleTuningCompactOnIdle -XX:+IdleTuningGcOnIdle -Xscmx50M -Xshareclasses -Xtune:virtualized" \
-t spring-petclinic-custom-jvm:3.2.0-SNAPSHOT
```

Build	Image	Startup	Initial RAM	Throughput	RAM	50%	75%	90%	99%	99.90%	99.99%
⏱️60s	349MB	4.813s	299MB	1994/s	465MB	7ms	13ms	19ms	52ms	109ms	164ms
-	-	9.976s	185MB	1996/s	355MB	11ms	19ms	30ms	85ms	171ms	277ms

GraalVM

GraalVM

```xml
<plugin>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-maven-plugin</artifactId>
  <configuration>
    <image>
        <buildpacks>
            <buildpack>gcr.io/paketo-buildpacks/graalvm:latest</buildpack>
            <!-- Used to inherit all the other buildpacks -->
            <buildpack>gcr.io/paketo-buildpacks/java:latest</buildpack>
        </buildpacks>
    </image>
  </configuration>
</p
```

Build	Image	Startup	Initial RAM	Throughput	RAM	50%	75%	90%	99%	99.90%	99.99%
⏱60s	349MB	4.813s	299MB	1994/s	465MB	7ms	13ms	19ms	52ms	109ms	164ms
75s	753MB	4.751s	266MB	1993/s	448MB	8ms	13ms	19ms	38ms	86ms	147ms

```
sdk

mvn spring-boot:build-image

docker run -p 8080:8080 -t spring-petclinic-custom-jvm-graalvm:3.2.0-SNAPSHOT
```

Other Buildpack Builders

Bellsoft Buildpack Builder

Bellsoft provides an optimized builder for Spring Boot applications. It uses the Bellsoft Alpaquita, Liberica JDK and the musl C library. A glibc version is also available.

```
<plugin>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-maven-plugin</artifactId>
  <configuration>
    <image>
      <builder>bellsoft/buildpacks.builder:musl</builder>
    </image>
  </configuration>
</plugin>
```

```
sdl

mv

do
```

Build	Image	Startup	Initial RAM	Throughput	RAM	50%	75%	90%	99%	99.90%	99.99%
⏱60s	349MB	4.813s	299MB	1994/s	465MB	7ms	13ms	19ms	52ms	109ms	164ms
(m) 58s	176MB	5.249s	263MB	1994/s	413MB	7ms	12ms	18ms	41ms	79ms	131ms
(g) 58s	190MB	4.787s	319MB	1993/s	480MB	6ms	10ms	15ms	32ms	74ms	124ms

GraalVM Native Image

GraalVM Native Image

A native image is a technology to build Java code to a standalone executable. This executable includes the application classes, classes from its dependencies, runtime library classes, and statically linked native code from JDK. The JVM is packaged into the native image, so there's no need for any Java Runtime Environment at the target system, but the build artifact is platform-dependent.

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

docker run -p 8080:8080 -t spring-petclinic-native:3.2.0-SNAPSHOT
```

Build	Image	Startup	Initial RAM	Throughput	RAM	50%	75%	90%	99%	99.90%	99.99%
⏱60s	349MB	4.813s	299MB	1994/s	465MB	7ms	13ms	19ms	52ms	109ms	164ms
328s	222MB	0.574s	257MB	1992/s	472MB	104ms	211ms	377ms	839ms	1233ms	1497ms

CRaC - OpenJDK

CRaC - OpenJDK (1)

CRaC (Checkpoint and Restart in Java) is a feature that allows to checkpoint of the state of a Java application and restart it from the checkpointed state.

The application starts within milliseconds!

CRaC - OpenJDK (2)

```
export JAVA_HOME=/opt/openjdk-17-crac+5_linux-x64/
export PATH=$JAVA_HOME/bin:$PATH
```

```
mvn clean verify
```

```
java -XX:CRaCCheckpointTo=crac-files -jar target/spring-petclinic-crac-3.2.0.jar
```

```
jcmd target/spring-petclinic-crac-3.2.0.jar JDK.checkpoint
```

```
java -XX:CRaCRestoreFrom=crac-files
```

CRaC - OpenJDK (3)

CRaC is currently in an experimental state and has the following limitations:

- Works with Spring Boot 3.0 & 3.1
    - Patched Tomcat 10.1.7 is available
- Full support since Spring Boot 3.2
    - Spring Framework 6.1.0
- Does not work on Windows or on macOS
    - Does not work in Docker containers via WSL (yet)
- But works in VM with Ubuntu 22.04 LTS

Other JVM Vendors have similar features e.g. OpenJ9 with CRIU support.

Virtual Threads

Virtual Threads

A thread is the smallest unit of processing that can be scheduled. It runs concurrently with — and largely independently of — other such units. It's an instance of `java.lang.Thread`. There are two kinds of threads, platform threads and virtual threads.

```
sdk use java 21.0.1-librca

mvn spring-boot:build-image

docker run -e spring.threads.virtual.enabled=true \
  -p 8080:8080 -t spring-petclinic-virtual-threads:3.2.0-SNAPSHOT
```

Build	Image	Startup	Initial RAM	Throughput	RAM	50%	75%	90%	99%	99.90%	99.99%
⏱60s	349MB	4.813s	299MB	1994/s	465MB	7ms	13ms	19ms	52ms	109ms	164ms
60s	401MB	4.873s	304MB	1995/s	453MB	4ms	8ms	11ms	17ms	37ms	51ms

Summary

Summary

- No Optimizations with JDK 17 & JDK 21
- JVM Tuning
- Lazy Spring Beans
- No Spring Boot Actuators
- Fix Spring Boot Config Location
- Disabling JMX
- Dependency Cleanup
- Ahead-of-Time Processing (AOT)
- JLink
- Other JVMs (Eclipse OpenJ9, GraalVM, OpenJDK with CRaC)
- GraalVM Native Image

Conclusions

Conclusions (1)

CPUs

- Your application might not need a full CPU at runtime
  It will require multiple CPUs to start up as quickly as possible (at least 2, 4 are better)
- If you don’t mind a slower startup, you could throttle the CPUs down below 4

See: https://spring.io/blog/2018/11/08/spring-boot-in-a-container

Conclusions (2)

Throughput

- Every application is different and has different requirements
- Using proper load testing can help to find the optimal configuration for your application

Conclusions (3)

Other Runtimes

- CRIU Support for OpenJDK and OpenJ9 is promising
  - Spring is going to support it in Spring Boot 3.2 / Spring Framework 6.1
- GraalVM Native Image is an excellent option for Java applications
  - But build times are long
  - The result is different from what you run in your IDE
- Eclipse OpenJ9 is an excellent option for running apps with less memory
  - But startup times are longer than with HotSpot
- Depending on the distribution, you might get other exciting features
  - Oracle GraalVM Enterprise Edition, Azul Platform Prime, IBM Semeru Runtime, ...

Conclusions (4)

Other Ideas

- Using an Obfuscator like ProGuard*
- App CDS (Class Data Sharing)*
- Importing `AutoConfiguration` classes individually
- Using functional bean definitions
- Spring Context Indexer (removed from Spring Boot 3.2 onwards)
- More JVM tuning (GC, Memory, etc.)
- Project Leyden

See also: https://spring.io/blog/2019/01/21/manual-bean-definitions-in-spring-boot

A Few Simple Optimizations Applied

A Few Simple Optimizations Applied (1)

- Dependency Cleanup
    - DB Drivers, Spring Boot Actuator, Jackson, Tomcat Websocket, ...
- Bellsoft Buildpack (musl)
- JLink
- JVM Parameters (java-memory-calculator)
- Spring AOT
- Lazy Spring Beans
- Fix Spring Boot Config Location
- Virtual Threads

A Few Simple Optimizations Applied (2)

```
sdk use java 17.0.9-librca

mvn spring-boot:build-image

docker run -p 8080:8080 \
  -e spring.aot.enabled=true \
  -e spring.main.lazy-initialization=true \
  -e spring.data.jpa.repositories.bootstrap-mode=lazy \
  -e spring.config.location=classpath:application.properties \
  -t spring-petclinic-optimized-bellsoft-builder:3.2.0-SNAPSHOT
```

Build	Image	Startup	Initial RAM	Throughput	RAM	50%	75%	90%	99%	99.90%	99.99%
⏱60s	349MB	4.813s	299MB	1994/s	465MB	7ms	13ms	19ms	52ms	109ms	164ms
66s	141MB	2.061s	187MB	1994/s	353MB	5ms	9ms	14ms	31ms	72ms	110ms

A Few Simple Optimizations Applied (3)

```
sdk use java 21.0.1-librca

mvn spring-boot:build-image

docker run -p 8080:8080 \
  -e spring.threads.virtual.enabled=true \
  -e spring.aot.enabled=true \
  -e spring.main.lazy-initialization=true \
  -e spring.data.jpa.repositories.bootstrap-mode=lazy \
  -e spring.config.location=classpath:application.properties \
  -t spring-petclinic-optimized-virtual-threads:3.2.0-SNAPSHOT
```

Build	Image	Startup	Initial RAM	Throughput	RAM	50%	75%	90%	99%	99.90%	99.99%
⏱60s	349MB	4.813s	299MB	1994/s	465MB	7ms	13ms	19ms	52ms	109ms	164ms
72s	272MB	1.852s	238MB	1995/s	425MB	5ms	8ms	10ms	17ms	52ms	85ms

Did I miss something? 🧐

Let me/us know! 🙋

… or not! 🙊

Lean Spring Boot

Applications for The Cloud

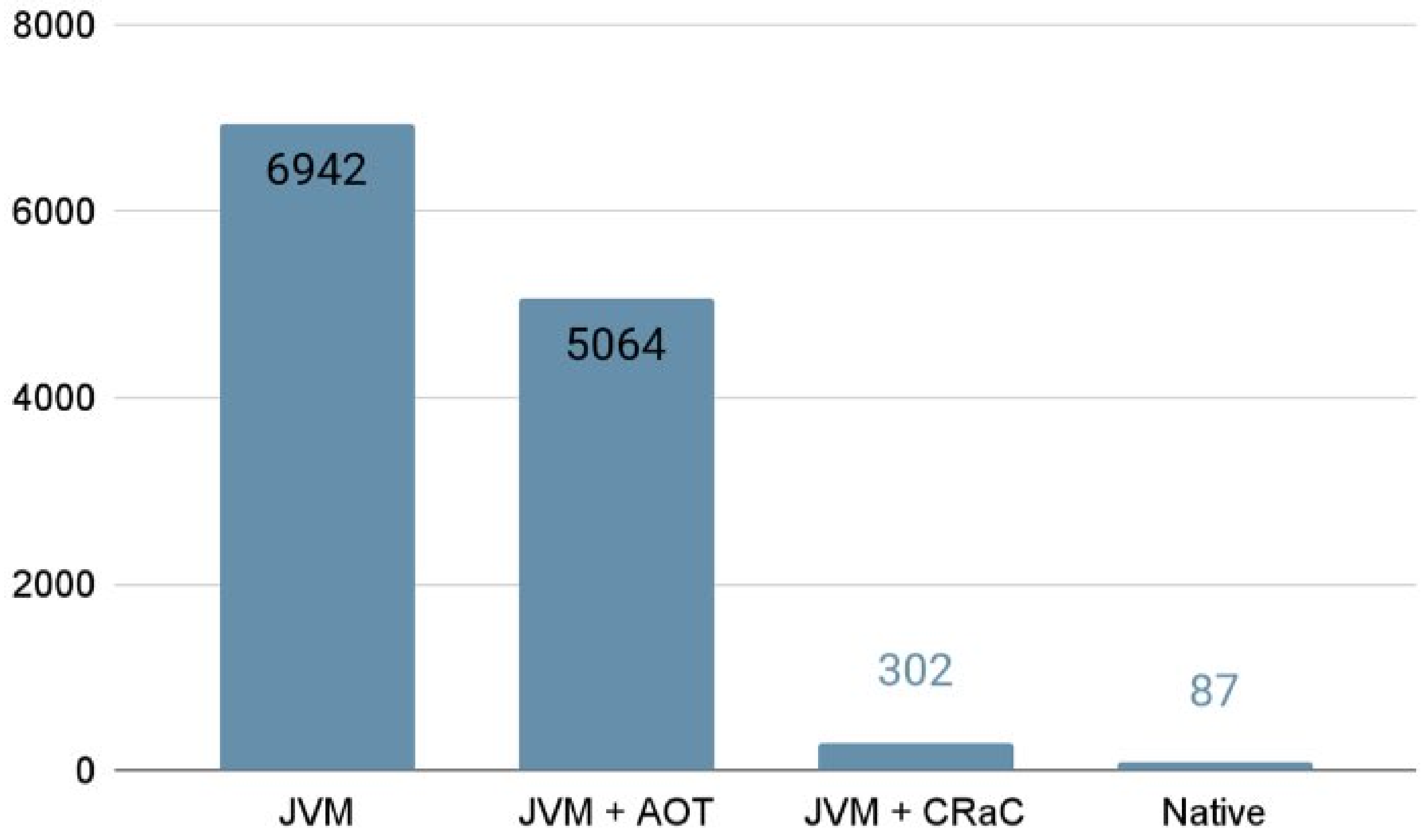
Patrick Baumgartner
42talents GmbH, Zürich, Switzerland

@patbaumgartner
patrick.baumgartner@42talents.com

https://github.com/patbaumgartner/lean-spring-boot-applications-for-the-cloud

Container start to application ready (milliseconds)

Webapp on Azure Container Apps with 1 CPU 2G memory



https://twitter.com/sdeleuze/status/1659129795056115712

Different tradeoffs

	Instant startup with peak performance	Require upfront deployment and checkpoint storage	Compatibility	Run on low resource devices	Compilation time	Compact packaging	Performance
GraalVM native image	Yes	No	Reachability Metadata	Yes	Slow	Yes	EE CE
CRaC JVM image	Yes	Yes for now[1]	Regular JVM[2]	No	Fast	JVM + checkpoint image	Regular JVM

[1] Build-time checkpoint could lift this requirement

[2] Can require custom checkpoint handling for specific use cases

SPRING 10TH ANNIVERSARY 2023

https://twitter.com/sdeleuze/status/1659461015719890944