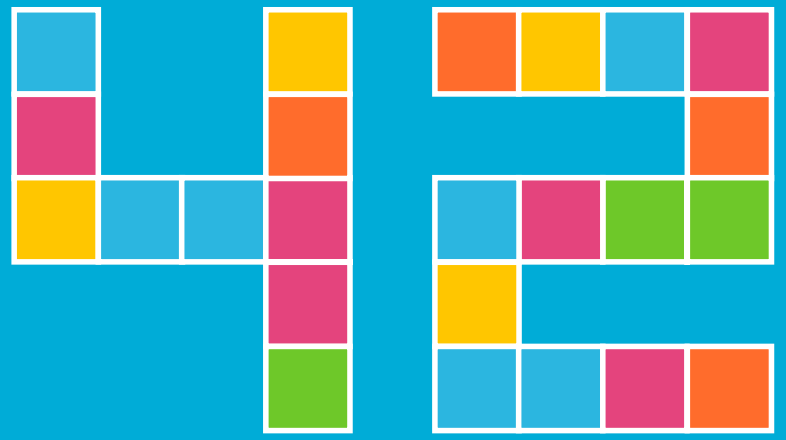
Lean Spring Boot

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 preconfigured. 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, and more. In addition, we're also looking at switching to a different JVM and other tools. All state-of-the-art technology, of course.

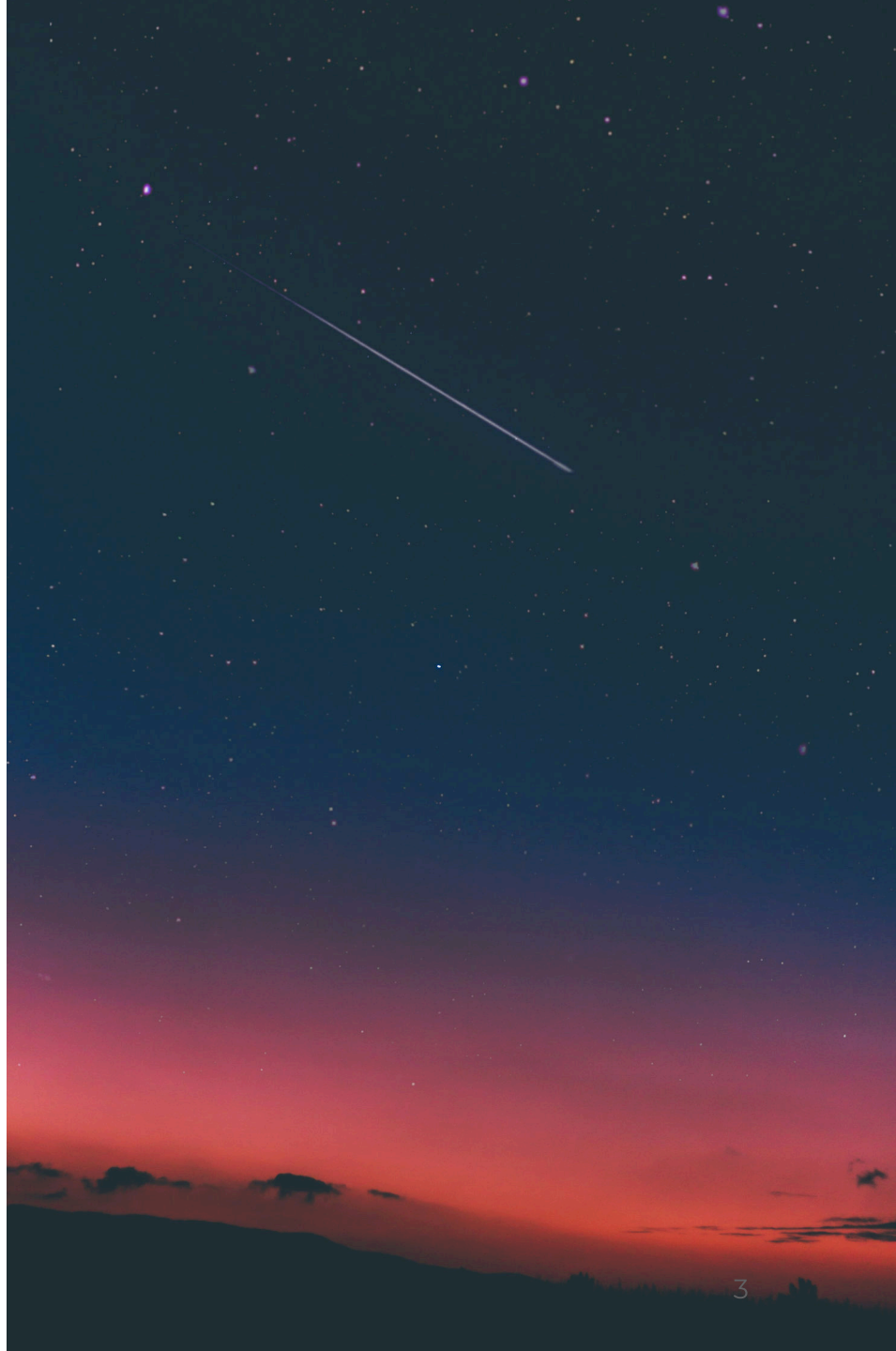
Let's make Spring Boot great again!

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 @ **Zurich University of Applied Sciences ZHAW**

Co-Organizer of **Voxxed Days Zurich**, **JUG Switzerland**, Oracle ACE Pro Java ...

@patbaumgartner

What's the challenge?

Why does this matter?

I ❤️ Java 😉 & Spring Boot 🍃

Requirements

When Deploying to a Cloud

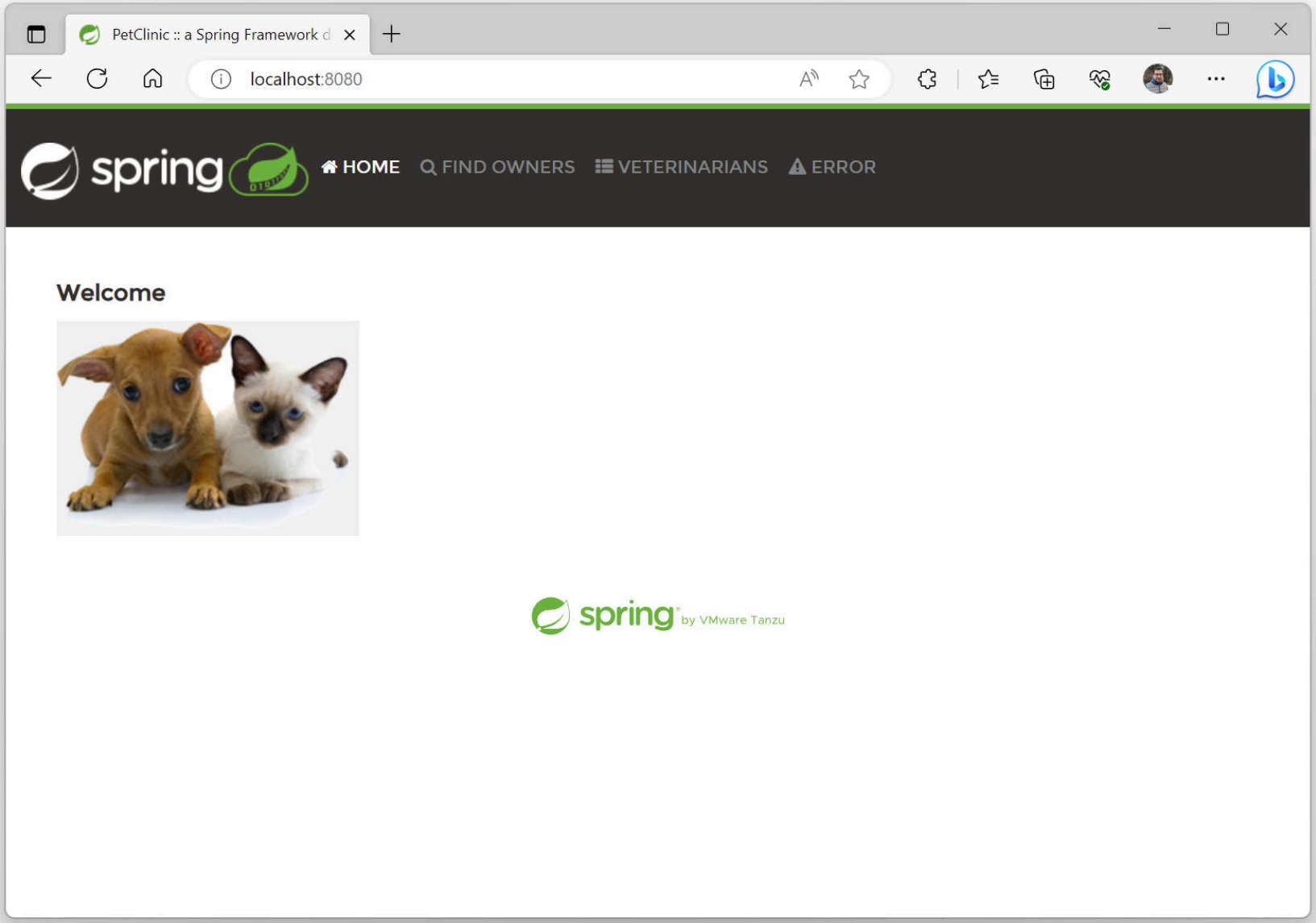
- How many vCPUs will my application need?
- How much RAM do I need?
- How much storage do I need?
- What technology stack should I use?
- What type of application do we build?

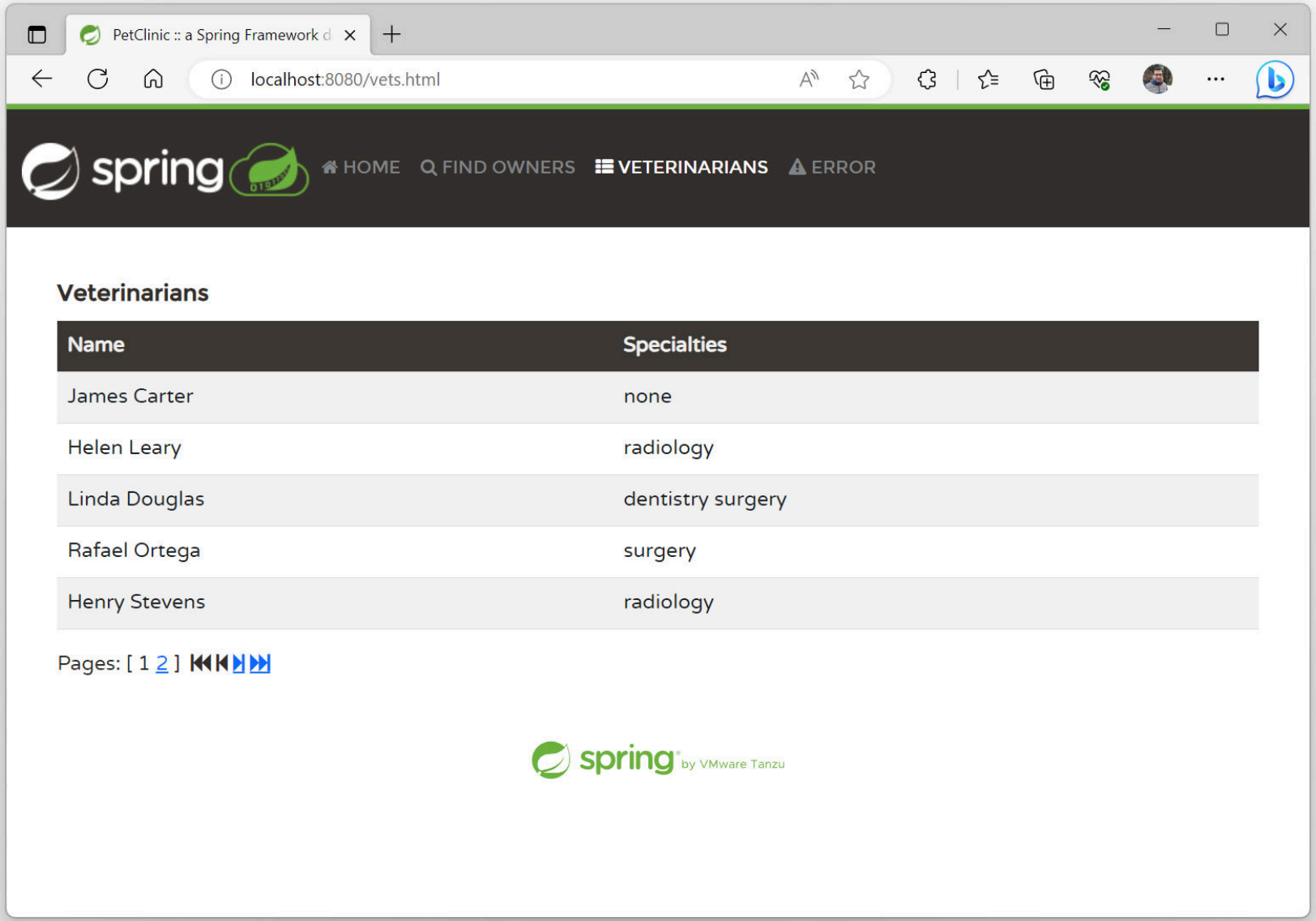
Agenda

Agenda

- Spring PetClinic & Baseline for comparison
- JVM Optimisations
- Spring Boot Optimisations
- Application Optimisations
- Other Runtimes
- Conclusions
- Some simple optimisations applied (OpenJDK examples)

Spring PetClinic


PetClinic :: a Spring Framework d
localhost:8080
spring
HOME
FIND OWNERS
VETERINARIANS
ERROR
Welcome
spring by VMware Tanzu


PetClinic :: a Spring Framework d
localhost:8080/vets.html
spring
HOME
FIND OWNERS
VETERINARIANS
ERROR
Veterinarians
Name
Specialties
James Carter
none
Helen Leary
radiology
Linda Douglas
dentistry surgery
Rafael Ortega
surgery
Henry Stevens
radiology
Pages: [1 2]
spring by VMware Tanzu

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! 🐾🏥

Based on Spring Boot, Caffeine, Thymeleaf, Spring Data JPA, H2 and Spring MVC ...

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

Optimisation Experiments

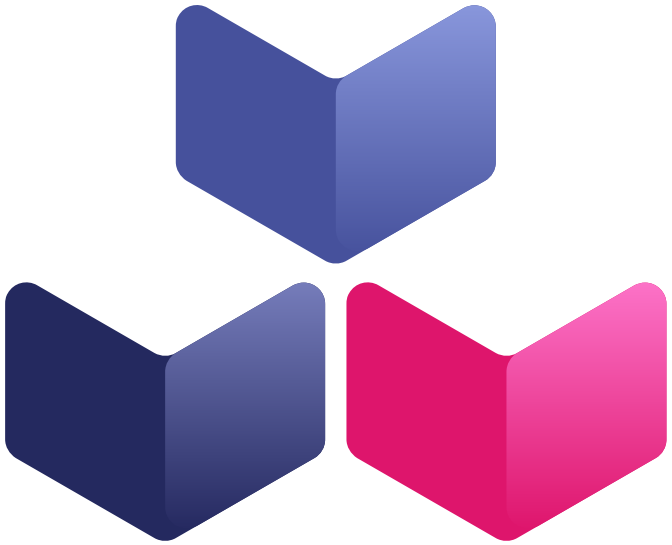
Baseline

Technology Stack

- OCI Container built with Buildpacks
- Java JDK 17 LTS
- Spring Boot 3.3.5
- Testcontainers
- DB migration using SQL scripts

Examination

- Build time
- Startup time
- Resource usage
- Container image size
- Throughput



Buildpacks.io

paketo

buildpacks

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

```
Calculating JVM memory based on 9314948K 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 -Xmx8681953K -XX:MaxMetaspaceSize=120994K -XX:ReservedCodeCacheSize=240M -Xss1M (Total
Memory: 9314948K, Thread Count: 250, Loaded Class Count: 18948, Headroom: 0%)
Enabling Java Native Memory Tracking
Adding 146 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:MaxDirectMemorySize=10M -Xmx8681953K -XX:MaxMetaspaceSize=120994K -XX:ReservedCodeCacheSize=240M -Xss1M -XX:
  +UnlockDiagnosticVMOptions -XX:NativeMemoryTracking=summary -XX:+PrintNMTStatistics -Dorg.springframework.cloud.bindings.boot.enable=true


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

:: Built with Spring Boot :: 3.3.5


2024-11-05T08:05:35.026Z  INFO 1 --- [           main] o.s.s.petclinic.PetClinicApplication     : Starting PetClinicApplication v3.3.0-SNAPSHOT using Java 17.0.13 with PID 1
                                                         (/workspace/BOOT-INF/classes started by cnb in /workspace)
2024-11-05T08:05:35.034Z  INFO 1 --- [           main] o.s.s.petclinic.PetClinicApplication     : No active profile set, falling back to 1 default profile: "default"
2024-11-05T08:05:36.886Z  INFO 1 --- [           main] .s.d.r.c.RepositoryConfigurationDelegate : Bootstrapping Spring Data JPA repositories in DEFAULT mode.
2024-11-05T08:05:37.055Z  INFO 1 --- [           main] .s.d.r.c.RepositoryConfigurationDelegate : Finished Spring Data repository scanning in 145 ms. Found 2 JPA repository interfaces.
2024-11-05T08:05:41.020Z  INFO 1 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port 8080 (http)
2024-11-05T08:05:41.094Z  INFO 1 --- [           main] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2024-11-05T08:05:41.096Z  INFO 1 --- [           main] o.apache.catalina.core.StandardEngine    : Starting Servlet engine: [Apache Tomcat/10.1.31]
2024-11-05T08:05:41.418Z  INFO 1 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext
2024-11-05T08:05:41.428Z  INFO 1 --- [           main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 6314 ms
2024-11-05T08:05:43.464Z  INFO 1 --- [           main] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Starting...
2024-11-05T08:05:44.732Z  INFO 1 --- [           main] com.zaxxer.hikari.pool.HikariPool        : HikariPool-1 - Added connection conn0: url=jdbc:h2:mem:778989e7-8df9-4853-97e0-09d9e8c6808b user=SA
2024-11-05T08:05:44.740Z  INFO 1 --- [           main] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Start completed.
2024-11-05T08:05:45.761Z  INFO 1 --- [           main] o.hibernate.jpa.internal.util.LogHelper  : HHH000204: Processing PersistenceUnitInfo [name: default]
2024-11-05T08:05:46.060Z  INFO 1 --- [           main] org.hibernate.Version                    : HHH000412: Hibernate ORM core version 6.5.3.Final
2024-11-05T08:05:46.254Z  INFO 1 --- [           main] o.h.c.internal.RegionFactoryInitiator    : HHH000026: Second-level cache disabled
2024-11-05T08:05:47.876Z  INFO 1 --- [           main] o.s.o.j.p.SpringPersistenceUnitInfo      : No LoadTimeWeaver setup: ignoring JPA class transformer
2024-11-05T08:05:51.042Z  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)
2024-11-05T08:05:51.048Z  INFO 1 --- [           main] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit 'default'
2024-11-05T08:05:51.835Z  INFO 1 --- [           main] o.s.d.j.r.query.QueryEnhancerFactory     : Hibernate is in classpath; If applicable, HQL parser will be used.
2024-11-05T08:05:57.805Z  INFO 1 --- [           main] o.s.b.a.e.web.EndpointLinksResolver      : Exposing 14 endpoints beneath base path '/actuator'
2024-11-05T08:05:58.297Z  INFO 1 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port 8080 (http) with context path '/'
2024-11-05T08:05:58.398Z  INFO 1 --- [           main] o.s.s.petclinic.PetClinicApplication     : Started PetClinicApplication in 23.913 seconds (process running for 24.339)
```

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/

Friends don't let friends write Dockerfiles!

@starbuxman

Benchmarks

Benchmarks

- Build
  - Maven build time
  - Artifact / Container Image size
- Startup
  - Startup time
  - Memory usage
- Throughput & Latency
  - `oha --no-tui -z60s --disable-keepalive <URL>`
  - 30s warmup, 60s measurement
  - Docker container with 2 vCPU and 1 GB RAM

No Optimizing - Baseline

No Optimizing - Baseline JDK 17

- Spring PetClinic (no adjustments)
- Bellsoft Liberica JDK 17.0.13
- Java Memory Calculator

```
sdk use java 17.0.13-librca

mvn spring-boot:build-image

docker run -p 8080:8080 -t spring-petclinic:3.3.0-SNAPSHOT
```

	Build	Image	Startup	Initial RAM	Requests/s	RAM	50%	75%	90%	99%	99.9%
⏱️ Java 17	65s	360MB	5.257s	262MB	797/s	369MB	65ms	84ms	100ms	175ms	253ms

No Optimizing - Baseline JDK 21

- Spring PetClinic (JDK 21 adjustments)
- Bellsoft Liberica JDK 21.0.4
- Java Memory Calculator

```
sdk use java 21.0.5-librca

mvn -Djava.version=21 spring-boot:build-image \
    -Dspring-boot.build-image.imageName=spring-petclinic:3.3.0-SNAPSHOT-jdk21

docker run -p 8080:8080 -t spring-petclinic:3.3.0-SNAPSHOT-jdk21
```

	Build	Image	Startup	Initial RAM	Requests/s	RAM	50%	75%	90%	99%	99.9%
⏱ Java 17	65s	360MB	5.257s	262MB	797/s	369MB	65ms	84ms	100ms	175ms	253ms
Java 21	64s	388MB	5.268s	273MB	968/s	394MB	47ms	72ms	91ms	161ms	246ms

No Optimizing - Baseline JDK 23

- Spring PetClinic (JDK 23 adjustments)
- Bellsoft Liberica JDK 23
- Java Memory Calculator

```
sdk use java 23.0.1-librca

mvn -Djava.version=23 spring-boot:build-image \
    -Dspring-boot.build-image.imageName=spring-petclinic:3.3.0-SNAPSHOT-jdk23

docker run -p 8080:8080 -t spring-petclinic:3.3.0-SNAPSHOT-jdk23
```

	Build	Image	Startup	Initial RAM	Requests/s	RAM	50%	75%	90%	99%	99.9%
⏱️ Java 17	65s	360MB	5.257s	262MB	797/s	369MB	65ms	84ms	100ms	175ms	253ms
Java 23	63s	388MB	5.27s	274MB	1136/s	462MB	35ms	61ms	81ms	144ms	211ms

JVM Optimisations

-noverify

-noverify

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

Warning: The `-Xverify:none` and `-noverify` options are deprecated in JDK 13 and are likely to be removed in a future release.

```
docker run -p 8080:8080 -e "JAVA_TOOL_OPTIONS=-noverify" \
  -t spring-petclinic:3.3.0-SNAPSHOT
```

	Build	Image	Startup	Initial RAM	Requests/s	RAM	50%	75%	90%	99%	99.9%
⏱ Java 17	65s	360MB	5.257s	262MB	797/s	369MB	65ms	84ms	100ms	175ms	253ms
Java 17	-	-	4.685s	244MB	842/s	358MB	59ms	81ms	98ms	172ms	247ms
Java 21	-	-	4.666s	264MB	996/s	384MB	45ms	70ms	89ms	157ms	242ms
Java 23	-	-	4.99s	265MB	1018/s	425MB	43ms	68ms	88ms	163ms	247ms

-XX:TieredStopAtLevel=1

-XX:TieredStopAtLevel=1

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

```
docker run -p 8080:8080 -e "JAVA_TOOL_OPTIONS=-XX:TieredStopAtLevel=1" \
  -t spring-petclinic:3.3.0-SNAPSHOT
```

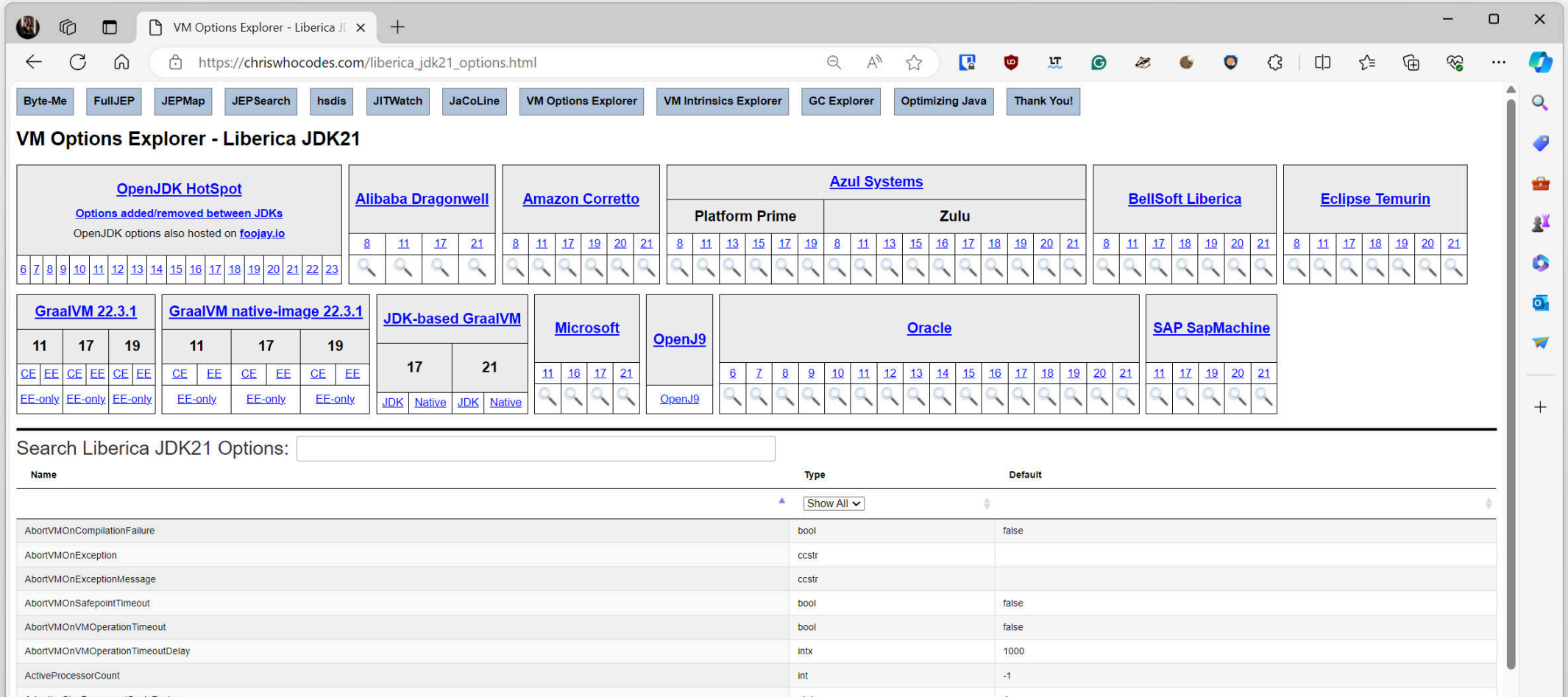
It will slow down the JIT later at the expense of the saved startup time!

	Build	Image	Startup	Initial RAM	T...	RAM	50%	75%	90%	99%	99.9%
⏱ Java 17	65s	360MB	5.257s	262MB	797/s	369MB	65ms	84ms	100ms	175ms	253ms
Java 17	-	-	4.116s	216MB	872/s	286MB	59ms	77ms	93ms	157ms	217ms
Java 21	-	-	4.321s	230MB	796/s	298MB	66ms	83ms	98ms	171ms	252ms
Java 23	-	-	4.187s	229MB	801/s	389MB	64ms	81ms	99ms	181ms	262ms

VM Options Explorer

VM Options Explorer

https://chriswhocodes.com



Spring Boot Optimisations

Lazy Spring Beans

Lazy Spring Beans (1)

Configure lazy initialisation throughout your application. A Spring Boot property makes all beans lazy by default, initialising them only when needed. You can use `@Lazy` to override this behaviour, 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:3.3.0-SNAPSHOT
```

	Build	Image	Startup	Initial RAM	Requests/s	RAM	50%	75%	90%	99%	99.9%
⏱ Java 17	65s	360MB	5.257s	262MB	797/s	369MB	65ms	84ms	100ms	175ms	253ms
Java 17	-	-	3.915s	224MB	795/s	371MB	64ms	84ms	101ms	179ms	261ms

Lazy Spring Beans (2)

Pros

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

Cons

- Initial requests may take longer
- Class loading problems and misconfigurations not detected at startup
- Beans creation errors only be found when the bean is loaded

Fixing Spring Boot Config Location

Fixing Spring Boot Config Location

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

- Application properties packaged in your jar
- Profile-specific application properties packaged within your jar
- Application properties outside your packaged jar
- Profile-specific application properties outside your packaged jar

```
docker run -p 8080:8080 -e spring.config.location=classpath:application.properties \
  -t spring-petclinic:3.3.0-SNAPSHOT
```

	Build	Image	Startup	Initial RAM	Requests/s	RAM	50%	75%	90%	99%	99.9%
⏱ Java 17	65s	360MB	5.257s	262MB	797/s	369MB	65ms	84ms	100ms	175ms	253ms
Java 17	-	-	5.235s	265MB	797/s	371MB	65ms	84ms	100ms	172ms	237ms

No Spring Boot Actuators

No Spring Boot Actuators

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

- Number of Spring Beans
  - Spring Pet Clinic with actuators: 448
  - Spring Pet Clinic without actuators: 272 🔥

```
sdk use java 17.0.13-librca

mvn spring-boot:build-image

docker run -p 8080:8080 -t spring-petclinic-no-actuator:3.3.0-SNAPSHOT
```

	Build	Image	Startup	Initial RAM	Requests/s	RAM	50%	75%	90%	99%	99.9%
⏱️ Java 17	65s	360MB	5.257s	262MB	797/s	369MB	65ms	84ms	100ms	175ms	253ms
Java 17	62s	358MB	4.235s	242MB	1038/s	356MB	43ms	68ms	86ms	147ms	206ms

Ahead-of-Time Processing (AOT)

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

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

As the `BeanFactory` is fully prepared at build time, conditions are also evaluated. E.g. in Configurations, Component- & Entity-Scan, `@Profile`, `@Conditional`, etc.

Spring AOT serves as a replacement for `spring-context-indexer` since Spring Framework 6.1 and Spring Boot 3.2.

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

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

Ahead-of-Time Processing (AOT) (3)

We create a new container image with the AOT-processed application. The Buildpack enables the property `spring.aot.enabled=true`

```
sdk use java 17.0.13-librca

mvn spring-boot:build-image

docker run -p 8080:8080 -t spring-petclinic-aot:3.3.0-SNAPSHOT
```

	Build	Image	Startup	Initial RAM	Requests/s	RAM	50%	75%	90%	99%	99.9%
⏱️ Java 17	65s	360MB	5.257s	262MB	797/s	369MB	65ms	84ms	100ms	175ms	253ms
Java 17	70s	362MB	4.754s	254MB	880/s	360MB	55ms	78ms	96ms	169ms	242ms
Java 21	70s	390MB	4.741s	272MB	962/s	391MB	48ms	72ms	92ms	165ms	250ms
Java 23	67s	389MB	4.819s	273MB	1141/s	526MB	35ms	61ms	82ms	143ms	205ms

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:3.3.0-SNAPSHOT
```

I ❤️ Spring Boot 🍃 & Buildpacks 🚀

Application Optimisations

Dependency Cleanup

Dependency Cleanup (2)

DepClean detects all unused dependencies declared in the `pom.xml` file of a project and creates a `pom-debloated.xml`. The generated report shows possible unused dependencies.

```
<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
```

```
------------------------------------------------------
 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:8.3.0:runtime (2 MB)
        org.postgresql:postgresql:42.7.4:runtime (1 MB)
        com.github.ben-manes.caffeine:caffeine:3.1.8:compile (868 KB)
        jakarta.xml.bind:jakarta.xml.bind-api:4.0.2:compile (127 KB)
        org.springframework.boot:spring-boot-testcontainers:3.3.5:test (117 KB)
        javax.cache:cache-api:1.1.1:compile (50 KB)
        org.testcontainers:junit-jupiter:1.19.8:test (11 KB)
        org.testcontainers:mysql:1.19.8:test (8 KB)
        ...
USED TRANSITIVE DEPENDENCIES [90]:
        org.testcontainers:testcontainers:1.19.8:test (16 MB)
        org.hibernate.orm:hibernate-core:6.5.3.Final:compile (11 MB)
        net.bytebuddy:byte-buddy:1.14.19:runtime (4 MB)
        org.apache.tomcat.embed:tomcat-embed-core:10.1.31:compile (3 MB)
        com.github.docker-java:docker-java-transport-zerodep:3.3.6:test (2 MB)
        org.aspectj:aspectjweaver:1.9.22.1:compile (2 MB)
        ...
USED INHERITED DIRECT DEPENDENCIES [0]:
USED INHERITED TRANSITIVE DEPENDENCIES [0]:
POTENTIALLY UNUSED DIRECT DEPENDENCIES [12]:
        org.webjars.npm:bootstrap:5.3.3:compile (1 MB)
        org.webjars.npm:font-awesome:4.7.0:compile (665 KB)
        org.springframework.boot:spring-boot-devtools:3.3.5:test (198 KB)
        org.springframework.boot:spring-boot-docker-compose:3.3.5:test (192 KB)
        ...
POTENTIALLY UNUSED TRANSITIVE DEPENDENCIES [8]:
        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:4.2.2:test (94 KB)
        org.springframework.boot:spring-boot-starter-tomcat:3.3.5:compile (4 KB)
        ...
POTENTIALLY UNUSED INHERITED DIRECT DEPENDENCIES [0]:
POTENTIALLY UNUSED INHERITED TRANSITIVE DEPENDENCIES [0]:
[INFO] Analysis done in 0min 15s
```

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 XML Configuration and `web.xml`

```
sdk use java 17.0.13-librca

mvn spring-boot:build-image

docker run -p 8080:8080 -t spring-petclinic-depclean:3.3.0-SNAPSHOT
```

	Build	Image	Startup	Initial RAM	Requests/s	RAM	50%	75%	90%	99%	99.9%
⏱️ Java 17	65s	360MB	5.257s	262MB	797/s	369MB	65ms	84ms	100ms	175ms	253ms
Java 17	62s	355MB	3.948s	236MB	952/s	342MB	50ms	73ms	91ms	158ms	233ms

More Application Optimisations

More Application Optimisations

- Application Optimizations
- Better Connection Pooling
- Improved Data Caching
- Database Queries Optimizations
- Batch Processing Optimizations
- Reduce Garbage Collection
- Monitor and Tune JVM Settings
- Reduce Overall Memory Footprint
- Async Logging
- ...

Other Runtimes

JLink

JLink (1)

`jlink` assembles and optimises 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	Requests/s	RAM	50%	75%	90%	99%	99.9%
⏱️ Java 17	65s	360MB	5.257s	262MB	797/s	369MB	65ms	84ms	100ms	175ms	253ms
Java 17	75s	272MB	5.3s	275MB	820/s	381MB	62ms	82ms	99ms	171ms	234ms
Java 21	74s	280MB	5.141s	287MB	970/s	403MB	47ms	72ms	91ms	158ms	244ms
Java 23	70s	278MB	5.231s	288MB	1035/s	419MB	42ms	68ms	87ms	151ms	213ms

Java Dependency Analysis Tool

`jdeps` is a Java command-line tool that analyzes class and jar files to list the package-level or class-level dependencies, helping developers understand module dependencies and improve code modularity.

```
jar xvf target/spring-petclinic-3.3.0-SNAPSHOT.jar

jdeps --ignore-missing-deps -q  \
    --recursive  \
    --multi-release 17  \
    --print-module-deps  \
    --class-path 'BOOT-INF/lib/*'  \
    target/spring-petclinic-3.3.0-SNAPSHOT.jar
```

```
java.base,java.compiler,java.desktop,java.instrument,java.net.http,java.prefs,java.rmi,java.scripting,
java.security.jgss,java.sql.rowset,jdk.jfr,jdk.management,jdk.net,jdk.unsupported
```

JLink (3)

```
<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 java.base,java.compiler,java.desktop,java.instrument,
                java.net.http,java.prefs,java.rmi,java.scripting,java.security.jgss,java.sql.rowset,
                jdk.jfr,jdk.management,jdk.net,jdk.unsupported --compress=2 --no-header-files --no-man-pages
                --strip-debug</BP_JVM_JLINK_ARGS>
            </env>
        </image>
    </configuration>
</plugin>
```

	Build	Image	Startup	Initial RAM	Requests/s	RAM	50%	75%	90%	99%	99.9%
⏱ Java 17	65s	360MB	5.257s	262MB	797/s	369MB	65ms	84ms	100ms	175ms	253ms
Java 17	71s	260MB	5.333s	270MB	859/s	380MB	58ms	80ms	96ms	166ms	235ms
Java 21	71s	268MB	5.221s	285MB	953/s	403MB	49ms	73ms	91ms	161ms	236ms
Java 23	69s	266MB	5.314s	287MB	984/s	409MB	46ms	71ms	90ms	162ms	250ms

Application Class Data Sharing (App CDS)

App CDS (1)

Class Data Sharing (CDS) is a JVM feature that can help reduce the startup time and memory footprint of Java applications. It allows classes to be pre-processed into a shared archive file that can be memory-mapped at runtime.

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

App CDS (2)

To create the archive, two additional JVM flags must be specified:

- `-XX:ArchiveClassesAtExit=application.jsa` : creates the CDS archive on exit
- `-Dspring.context.exit=onRefresh` : starts and then immediately exits

Once the archive is available, the JVM can be started with the additional flag:

- `-XX:SharedArchiveFile=application.jsa` : to use it

	Build	Image	Startup	Initial RAM	Requests/s	RAM	50%	75%	90%	99%	99.9%
⏱ Java 17	65s	360MB	5.257s	262MB	797/s	369MB	65ms	84ms	100ms	175ms	253ms
Java 17	78s	512MB	3.416s	245MB	814/s	361MB	63ms	83ms	100ms	175ms	251ms
Java 17	76s	539MB	3.522s	257MB	948/s	376MB	49ms	74ms	92ms	165ms	235ms
Java 23	75s	538MB	3.5s	242MB	997/s	387MB	45ms	71ms	90ms	155ms	226ms

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!

Faster Startup

Faster Ramp-up, when deployed to cloud

Smaller

Our Story

Eclipse OpenJ9

```
<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 Java buildpacks -->
            <buildpack>gcr.io/paketo-buildpacks/java:latest</buildpack>
        </buildpacks>
    </image>
  </configuration>
</plugin>
```

	Build	Image	Startup	Initial RAM	Requests/s	RAM	50%	75%	90%	99%	99.9%
⏱ Java 17	65s	360MB	5.257s	262MB	797/s	369MB	65ms	84ms	100ms	175ms	253ms
Java 17	68s	335MB	6.254s	192MB	1396/s	336MB	25ms	52ms	74ms	118ms	198ms
Java 21	68s	353MB	6.454s	188MB	1418/s	340MB	24ms	49ms	72ms	125ms	210ms

GraalVM

GraalVM CE

```
<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 Java buildpacks -->
            <buildpack>gcr.io/paketo-buildpacks/java:latest</buildpack>
        </buildpacks>
    </image>
  </configuration>
</plugin>
```

	Build	Image	Startup	Initial RAM	Requests/s	RAM	50%	75%	90%	99%	99.9%
⏱ Java 17	65s	360MB	5.257s	262MB	797/s	369MB	65ms	84ms	100ms	175ms	253ms
Java 17	75s	758MB	5.229s	289MB	912/s	374MB	53ms	75ms	93ms	161ms	227ms
Java 21	72s	743MB	5.346s	307MB	978/s	397MB	47ms	70ms	88ms	159ms	230ms
Java 23	74s	910MB	5.366s	310MB	1020/s	465MB	44ms	67ms	86ms	153ms	217ms

Oracle JDK

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

	Build	Image	Startup	Initial RAM	Requests/s	RAM	50%	75%	90%	99%	99.9%
⏱ Java 17	65s	360MB	5.257s	262MB	797/s	369MB	65ms	84ms	100ms	175ms	253ms
Java 17	66s	499MB	5.137s	263MB	848/s	370MB	59ms	81ms	98ms	171ms	245ms
Java 21	68s	528MB	5.081s	272MB	951/s	394MB	49ms	74ms	91ms	156ms	219ms

Other Buildpack Builders

Bellsoft Buildpack Builder

Bellsoft provides an optimised 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>
```

	Build	Image	Startup	Initial RAM	Requests/s	RAM	50%	75%	90%	99%	99.9%
⏱ Java 17	65s	360MB	5.257s	262MB	797/s	369MB	65ms	84ms	100ms	175ms	253ms
musl	63s	174MB	5.369s	264MB	766/s	350MB	69ms	86ms	101ms	166ms	211ms
glibc	64s	194MB	5.177s	271MB	914/s	381MB	53ms	76ms	93ms	158ms	210ms

Buildpack Builder Tiny

It's based on Ubuntu Jammy, and works with current JAVA versions. While the base image (default) is bigger the tiny is intended to be used with GraalVM native images.

```
<plugin>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-maven-plugin</artifactId>
  <configuration>
    <image>
      <builder>paketobuildpacks/builder-jammy-tiny</builder>
    </image>
  </configuration>
</plugin>
```

	Build	Image	Startup	Initial RAM	Requests/s	RAM	50%	75%	90%	99%	99.9%
⏱ Java 17	65s	360MB	5.257s	262MB	797/s	369MB	65ms	84ms	100ms	175ms	253ms
Java 17	64s	279MB	5.217s	259MB	814/s	371MB	62ms	83ms	99ms	174ms	254ms
Java 21	65s	307MB	5.265s	272MB	949/s	391MB	49ms	74ms	92ms	162ms	230ms
Java 23	61s	306MB	5.146s	272MB	1171/s	468MB	34ms	60ms	79ms	141ms	203ms

GraalVM Native Image

GraalVM Native Image (CE & Oracle)

A native image is a technology for building Java code into a standalone executable. This executable contains the application classes, classes from its dependencies, runtime library classes and statically linked native code from the JDK. The JVM is packaged in the native image, so there's no need for a Java Runtime Environment on 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:3.3.0-SNAPSHOT
```

	Build	Image	Startup	Initial RAM	Requests/s	RAM	50%	75%	90%	99%	99.9%
⏱ Java 17	65s	360MB	5.257s	262MB	797/s	369MB	65ms	84ms	100ms	175ms	253ms
CE 21	342s	243MB	0.415s	239MB	1229/s	295MB	32ms	57ms	77ms	123ms	190ms
Oracle 21	524s	250MB	0.321s	221MB	1581/s	270MB	23ms	45ms	61ms	99ms	155ms

CRaC - OpenJDK

CRaC - OpenJDK (1)

CRaC (Coordinated Restore at Checkpoint) is a feature that allows you to take a snapshot of the state of a Java application and restart it from that state.

Currently only available from:

- Azul Zulu
- Bellsoft Liberica

The application starts within milliseconds!

.. with many other challenges like: embedded configs and secrets, same OS (Linux) and CPU Architecture, same JVM versions, and more ...

CRaC - OpenJDK (2)

```
export JAVA_HOME=/opt/openjdk-17-crac+6_linux-x64/
export PATH=$JAVA_HOME/bin:$PATH
```

```
mvn clean verify
```

```
java -XX:CRaCCheckpointTo=crac-files -jar target/spring-petclinic-crac-3.3.0-SNAPSHOT.jar
```

```
jcmd target/spring-petclinic-crac-3.3.0-SNAPSHOT.jar JDK.checkpoint
```

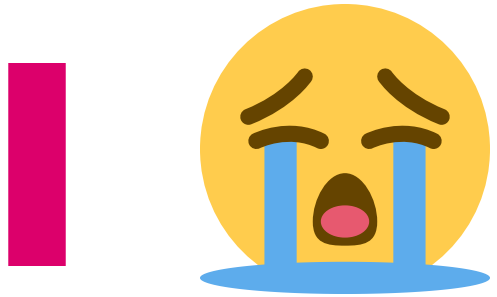
```
java -XX:CRaCRestoreFrom=crac-files
```

CRaC - OpenJDK (3)

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

- Since Spring Boot 3.2 CraC support finalised
  - Spring Framework 6.1.0
- Only available for Linux x86 / ARM 64 Bit
- Azul Zulu Build of OpenJDK with CRaC support for development purposes
  - Available for Windows and macOS
  - Simulated checkpoint/restore mechanism for development and testing

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



No CRaC Buildpacks for Java & Spring Boot

Virtual Threads (Java 21)

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 is an instance of `java.lang.Thread`. There are two types of threads, platform threads and virtual threads.

```
sdk use java 21.0.5-librca

mvn spring-boot:build-image

docker run -e spring.threads.virtual.enabled=true \
  -p 8080:8080 -t spring-petclinic-virtual-threads:3.3.0-SNAPSHOT
```

	Build	Image	Startup	Initial RAM	Requests/s	RAM	50%	75%	90%	99%	99.9%
⏱️ Java 17	65s	360MB	5.257s	262MB	797/s	369MB	65ms	84ms	100ms	175ms	253ms
Java 21	64s	388MB	5.375s	274MB	2365/s	373MB	19ms	21ms	26ms	56ms	64ms
Java 23	62s	388MB	5.259s	275MB	2494/s	376MB	17ms	18ms	21ms	46ms	54ms

Virtual Threads Java 21

CPU	VT	Startup	Initial RAM	Request/s	RAM	50%	75%	90%	99%	99.9%	99.99%
4 vCPU		5.133s	308MB	2581/s	414MB	16ms	22ms	32ms	64ms	109ms	160ms
4 vCPU	🧵	5.006s	298MB	3179/s	415MB	15ms	17ms	19ms	28ms	69ms	125ms
2 vCPU		5.289s	273MB	910/s	389MB	52ms	76ms	95ms	170ms	243ms	329ms
2 vCPU	🧵	5.251s	269MB	2296/s	378MB	19ms	22ms	28ms	58ms	65ms	146ms
1 vCPU		14.717s	272MB	109/s	354MB	427ms	621ms	814ms	1199ms	1547ms	1859ms
1 vCPU	🧵	14.3s	272MB	293/s	380MB	136ms	203ms	295ms	525ms	758ms	8725ms
0.5 vCPU		50.992s	266MB	28/s	337MB	1603ms	2366ms	3200ms	5234ms	6990ms	7740ms
0.5 vCPU	🧵	49.201s	264MB	34/s	363MB	1401ms	1732ms	2003ms	2706ms	5342ms	8123ms
0.25 vCPU		104.797s	267MB	9/s	337MB	5732ms	8166ms	10832ms	17114ms	20330ms	20330ms
0.25 vCPU	🧵	105.565s	265MB	9/s	329MB	5064ms	6266ms	8019ms	13570ms	14551ms	14551ms

Virtual Threads Java 23

CPU	VT	Startup	Initial RAM	Request/s	RAM	50%	75%	90%	99%	99.9%	99.99%
4 vCPU		4.98s	305MB	2769/s	477MB	16ms	20ms	28ms	58ms	93ms	130ms
4 vCPU	🧵	5.095s	300MB	3168/s	472MB	14ms	16ms	18ms	56ms	268ms	506ms
2 vCPU		5.391s	274MB	1067/s	464MB	40ms	65ms	85ms	151ms	224ms	326ms
2 vCPU	🧵	5.256s	272MB	2518/s	380MB	17ms	19ms	22ms	48ms	55ms	315ms
1 vCPU		14.518s	272MB	107/s	388MB	431ms	604ms	811ms	1260ms	1578ms	2055ms
1 vCPU	🧵	14.305s	273MB	791/s	367MB	40ms	90ms	107ms	260ms	541ms	700ms
0.5 vCPU		50.664s	273MB	30/s	353MB	1473ms	2165ms	3028ms	4797ms	6373ms	6736ms
0.5 vCPU	🧵	49.993s	269MB	36/s	352MB	1361ms	1625ms	1966ms	2759ms	3623ms	3720ms
0.25 vCPU		104.796s	274MB	9/s	324MB	5731ms	8134ms	11153ms	16702ms	21491ms	21491ms
0.25 vCPU	🧵	107.331s	269MB	10/s	323MB	4834ms	5834ms	7376ms	13829ms	14497ms	14497ms

Summary

Summary

- No Optimisations 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 may not need a full CPU at runtime.
- It will need several CPUs to start as fast as possible (at least 2, 4 is better).
- If you don't mind a slower startup, you can throttle the CPUs below 4.

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

Conclusions (2)

Request/s

- Every application is different and has different requirements.
- Proper load testing can help find the optimal configuration for your application.

Conclusions (3)

Other Runtimes

- CRIU Support for OpenJDK and OpenJ9 is promising.
  - Supported by Spring since Spring Boot 3.2 / Spring Framework 6.1
- GraalVM Native Image is a great 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 applications with less memory
  - But startup times are longer than with HotSpot.
- Depending on the distribution, you may get other exciting features.
  - Oracle GraalVM Enterprise Edition, Azul Platform Prime, IBM Semeru Runtime, ...

Conclusions (4)

Other Ideas

- CRaC (Coordinated Restore at Checkpoint)*
- Tree shaking / Obfuscator such as ProGuard*
- More JVM tuning (GC, Memory, etc.)
- Project Leyden

A Few Simple Optimisations Applied

A Few Simple Optimisations Applied

- Dependency Cleanup
  - DB Drivers, Spring Boot Actuator, Jackson, Tomcat Websocket, ...
- Bellsoft Builder (musl) / Base Builder Tiny
- JLink
- JVM Parameters (java-memory-calculator)
- Application Class Data Sharing (AppCDS)
- Spring AOT
- Lazy Spring Beans
- Fixing Spring Boot Config Location
- Virtual Threads

Bellsoft Builder (musl)

```
sdk use java 21.0.5-librca

mvn spring-boot:build-image

docker run -p 8080:8080 \
  -e spring.threads.virtual.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-builder-bellsoft-musl:3.3.0-SNAPSHOT
```

	VT	Build	Image	Startup	Initial RAM	Requests/s	RAM	50%	75%	90%	99%	99.9%
Java 17		65s	360MB	5.257s	262MB	797/s	369MB	65ms	84ms	100ms	175ms	253ms
Java 17		73s	141MB	3.585s	228MB	929/s	316MB	52ms	75ms	92ms	152ms	203ms
Java 21		75s	203MB	3.343s	225MB	1064/s	321MB	41ms	66ms	85ms	137ms	203ms

Java 21 & Base Builder Tiny

```
sdk use java 21.0.5-librca

mvn spring-boot:build-image

docker run -p 8080:8080 \
  -e spring.threads.virtual.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-builder-tiny-virtual-threads:3.3.0-SNAPSHOT
```

	VT	Build	Image	Startup	Initial RAM	Requests/s	RAM	50%	75%	90%	99%	99.9%
Java 17		65s	360MB	5.257s	262MB	797/s	369MB	65ms	84ms	100ms	175ms	253ms
Java 17		82s	307MB	3.309s	221MB	1014/s	350MB	44ms	68ms	87ms	153ms	208ms
Java 21		81s	314MB	3.265s	226MB	1218/s	371MB	32ms	58ms	77ms	134ms	206ms
Java 23		78s	312MB	3.346s	232MB	1426/s	690MB	26ms	50ms	69ms	114ms	191ms

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://bit.ly/lean-spring-boot