Spring Boot in the Cloud

Advanced Optimization Deep Dive

Patrick Baumgartner
42talents GmbH, Zürich, Switzerland

@patbaumgartner
patrick.baumgartner@42talents.com

Abstract

Spring Boot in the Cloud: Advanced Optimization Deep Dive

Spring Boot makes it easy to launch cloud-ready applications, but building truly lean and efficient cloud services requires going beyond the basics. In this three-hour deep dive, you'll immerse yourself in advanced Spring Boot techniques to optimize your applications for the cloud.

We will examine in depth how to reduce memory usage, improve startup times, and streamline your Spring Boot apps for modern cloud environments. Topics include Spring AOT, classpath exclusions, lazy beans, actuator configuration, custom JVM options, and additional state-of-the-art tools.

This session combines short presentations, live coding, and practical hands-on exercises. You will experiment with advanced configurations, observe real-time results, and troubleshoot performance challenges alongside the instructor. Real-world scenarios and best practices will help you immediately apply these techniques to your own projects.

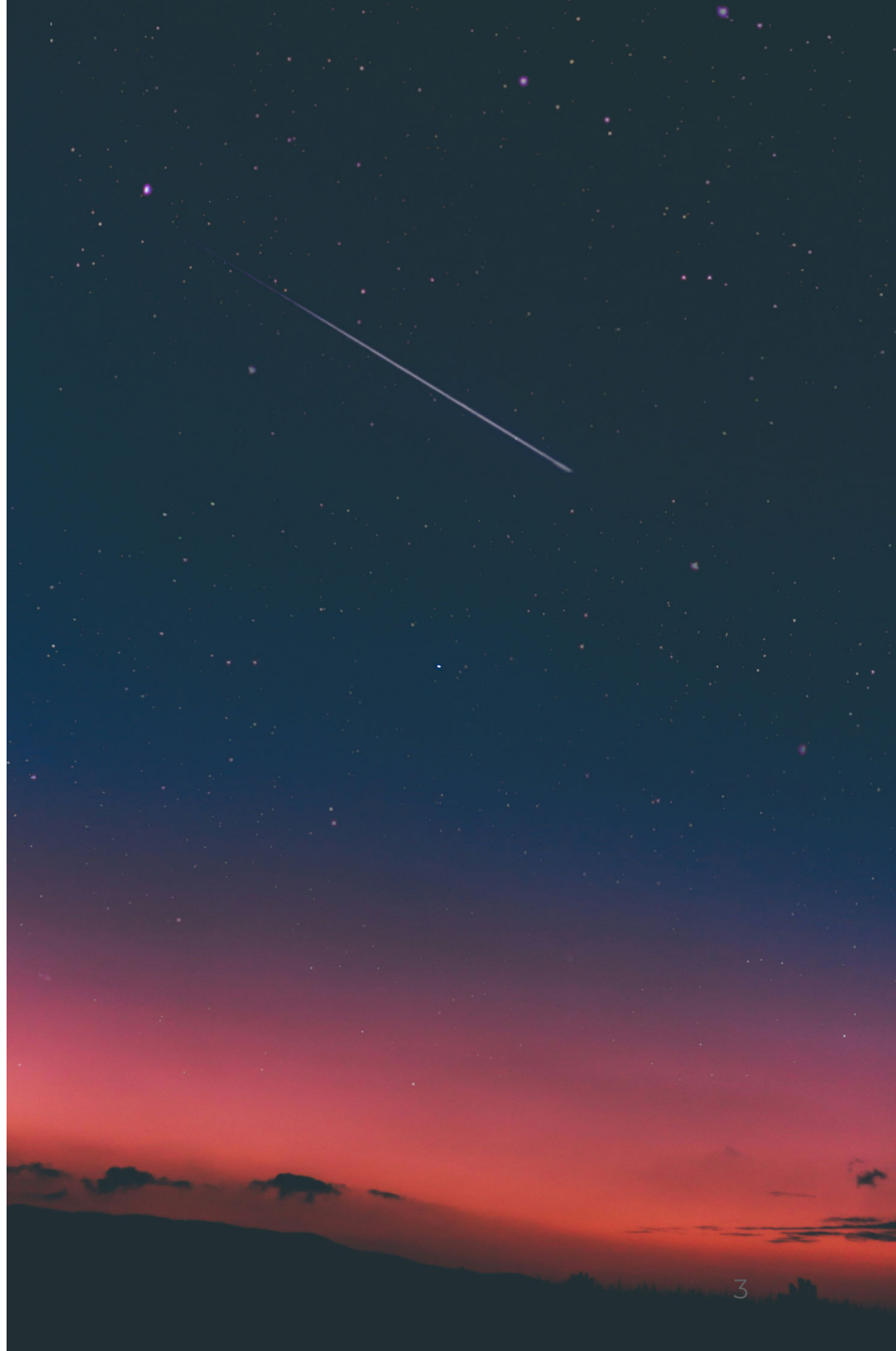
Whether you want to fine-tune your CI/CD pipeline or push Spring Boot to its limits in production, you will leave with actionable skills, a collection of code samples, and a deeper understanding of cloud-native application optimization. Get ready for an in-depth, practical exploration of building lean Spring Boot applications for the cloud.

Spring Boot in the Cloud

Advanced Optimization Deep Dive

Patrick Baumgartner
42talents GmbH, Zürich, Switzerland

@patbaumgartner
patrick.baumgartner@42talents.com



Spring Boot in the Cloud

Advanced Optimization Deep Dive

Patrick Baumgartner
42talents GmbH, Zürich, Switzerland

@patbaumgartner
patrick.baumgartner@42talents.com
bit.ly/48i5U9x

⚠ 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**, Java Champion, Oracle ACE Pro Java ...

@patbaumgartner

What's the challenge?

Why does this matter?

I ❤️ Java 😉 & Spring Boot 🍃

Introduction to Optimization Goals

Exercises

Questions ...

Think about your current project.

- What would you optimise your application for?
- What are the key metrics?
- How would you measure them?

> 3' in pairs

Measures

- Use Case
- Build Time
- Image Size
- Startup Time
- CPU Usage
- Memory Usage
- Throughput / Requests per Second
- Latency / Response Time

Requirements

When Deploying to a Cloud

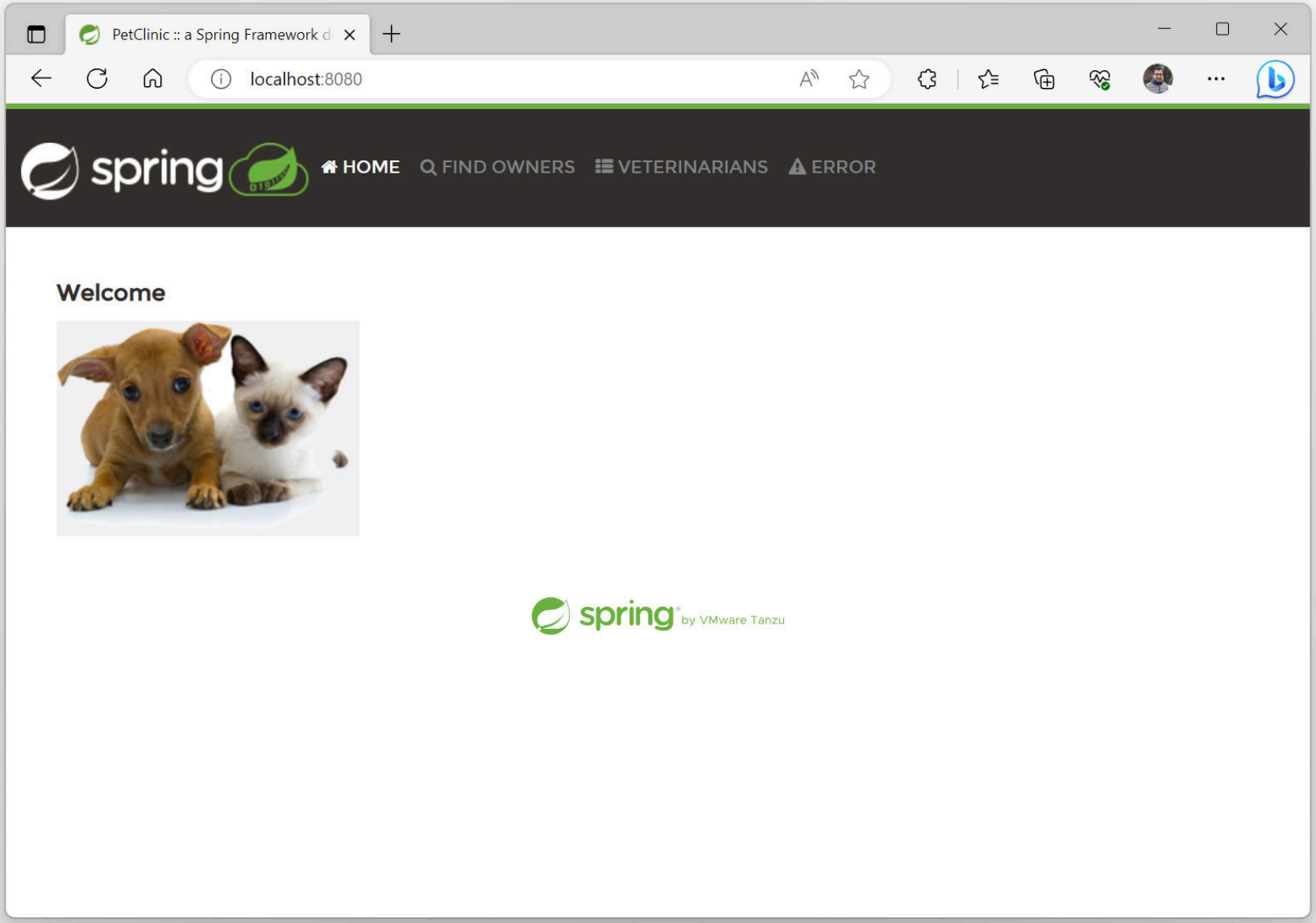
- How many vCPUs will my application need?
- How much RAM do I need?
- How much storage do I need?
- What technology stack should I use?
- What type of application do we build?

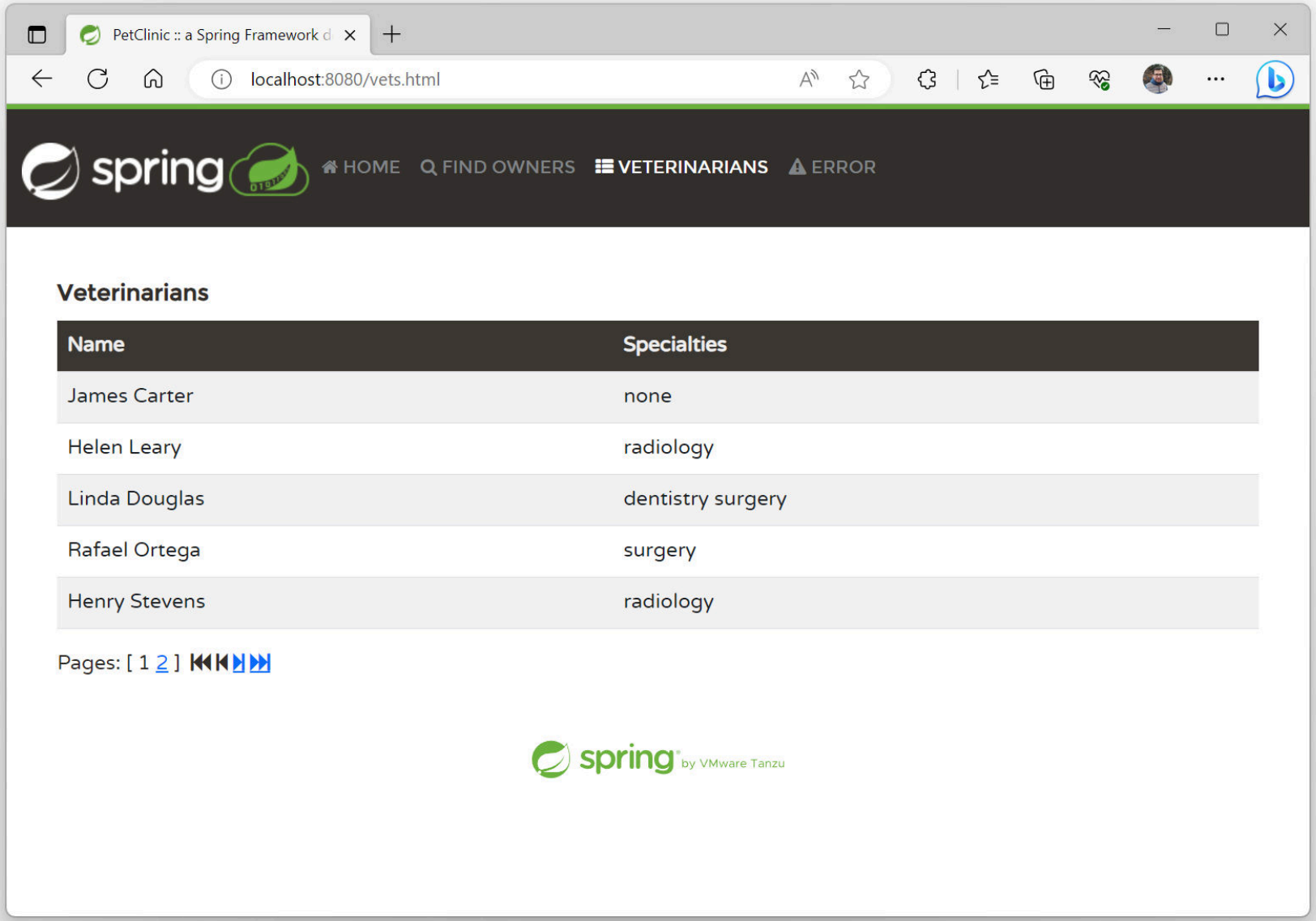
Agenda

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

Limitations

Limitations

- Docker / Java memory calculator ignores CPU limits in WSL

```
docker container run --rm -it -d --name spring-petclinic \
    --cpus=2 --memory=1024m spring-petclinic:3.5.0-SNAPSHOT
```

- Check your container usage with the following command

```
docker container stats
```

- Using `.wslconfig` to restrict VM resources

```
[wsl2]
memory=16GB # Limits VM memory in WSL 2 to 16 GB
processors=2 # Makes the WSL 2 VM only use 2 virtual processors
```

Limitations (2)

- Docker stats prints wrong values for memory usage, use docker top instead. (See: https://quarkus.io/guides/performance-measure#measuring-memory-correctly-on-docker)

```
docker top <CONTAINER ID> -o pid,rss,args
```

- Thermal throttling of your notebook CPU may affect your results

Benchmark Steps

Using oha

Build with Maven

Set the Java version and build the OCI container image.

```
sdk use java 17.0.16-librca
java -version

build_time=$(./mvnw clean spring-boot:build-image | grep "Total time:" | awk '{print $4 $5}')
echo -e  "\033[0;32mBuild time: $build_time\033[0m"

image_size=$(docker images | grep "spring-petclinic" | awk '{print $7}')
echo -e  "\033[0;32mImage Size: $image_size\033[0m"
```

Container Startup Time

Run the container with limited memory and measure the startup time.

```
docker container run -d --rm --memory="1g" -p 8080:8080 -t spring-petclinic:3.5.0-SNAPSHOT

# Wait for 10 seconds, Openj9 uses more time to start
sleep 10

# Get the container ID of the running container
container_id=$(docker ps -q)

startup_time=$(docker logs "$container_id" | grep "Started" | awk '{print $14 $15}')
echo -e  "\033[0;32mStartup time: $startup_time\033[0m"
```

Memory Usage after Startup

Extract the memory usage of the running container. Convert `rss` to MB.

```
memory=$(docker top $(docker ps -lq) -o pid,rss,args | tail -n +2 | awk '{$2=int($2/1024)"MB";}{ print $2;}')
echo -e  "\033[0;32mStartup memory usage: $memory\033[0m"
```

Warm-up Application

Warm up the application with for 60s, json output and no UI.

```
oha --no-tui -z60s --disable-keepalive -j "http://localhost:8080/owners?page=2"
```

Load-Test Application

Load test during 60s, json output and no UI.

```
throughput=$(oha --no-tui -z60s --disable-keepalive -j "http://localhost:8080/owners?page=2" 2>&1 | tee $filename  | jq '.summary.requestsPerSec')
echo -e  "\033[0;32mAvg Throughput: $throughput Requests/sec\033[0m"
```

Memory Usage after Load-Test

Extract the memory usage of the running container. Convert `rss` to MB.

```
memory=$(docker top $(docker ps -lq) -o pid,rss,args | tail -n +2 | awk '{$2=int($2/1024)"MB";}{ print $2;}')
echo -e  "\033[0;32mEnd memory usage: $memory\033[0m"
```

Docker Cleanup

- Stop all running containers

```
docker kill $(docker ps -q)
```

- Remove all spring-petclinic images and those without a version tag.

```
docker rmi $(docker images | grep "none" | awk '{print $3}') --force
docker rmi $(docker images | grep "spring-petclinic" | awk '{print $3}') --force
```

- Remove all containers, networks, and volumes (but not images)

```
docker container prune --force
docker network prune --force
docker volume prune --force
```

Using JMeter

Load-Testing with JMeter

```
<plugin>
  <groupId>com.lazerycode.jmeter</groupId>
  <artifactId>jmeter-maven-plugin</artifactId>
  <version>${jmeter-maven-plugin.version}</version>
  <configuration>
      <generateReports>true</generateReports>
  </configuration>
  <executions>
    <!-- Generate JMeter configuration -->
    <!-- Run JMeter tests -->
    <!-- Fail build on errors in test -->
    </execution>
  </executions>
</plugin>
```

See also: https://github.com/jmeter-maven-plugin/jmeter-maven-plugin/wiki/Basic-Configuration

Using Gatling

Load-Testing with Gatling

```
<plugin>
  <groupId>io.gatling</groupId>
  <artifactId>gatling-maven-plugin</artifactId>
  <version>${gatling-maven-plugin.version}</version>
  <configuration>
    <runMultipleSimulations>true</runMultipleSimulations>
    <includes>
      <include>org.springframework.samples.gatling.simulation.SpringPetclinic</include>
    </includes>
  </configuration>
</plugin>
```

See also: https://gatling.io/docs/gatling/reference/current/http/recorder/

Optimisation Experiments

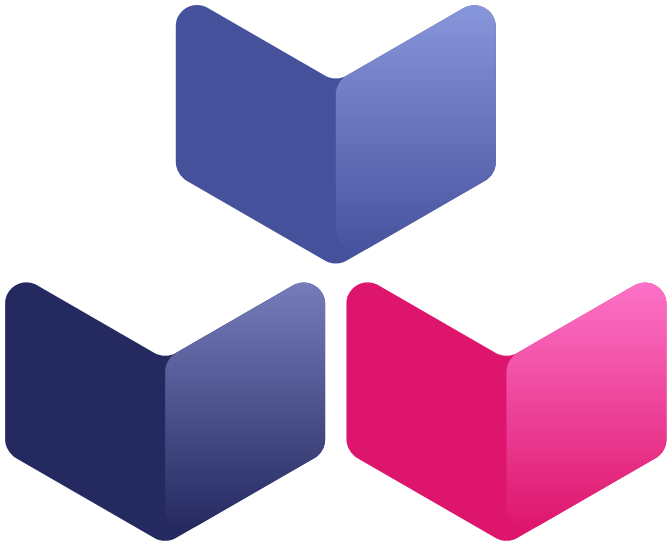
Baseline

Technology Stack

- OCI Container built with Buildpacks
- Java JDK 17 LTS
- Spring Boot 3.5.6
- 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

```
Calculated JVM Memory Configuration: -XX:MaxDirectMemorySize=10M -Xmx407113K -XX:MaxMetaspaceSize=129462K -XX:ReservedCodeCacheSize=240M
-Xss1M (Total Memory: 1G, Thread Count: 250, Loaded Class Count: 20443, 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 -Xmx407113K -XX:MaxMetaspaceSize=129462K -XX:ReservedCodeCacheSize=240M -Xss1M
-XX:+UnlockDiagnosticVMOptions -XX:NativeMemoryTracking=summary -XX:+PrintNMTStatistics -Dorg.springframework.cloud.bindings.boot.enable=true


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

:: Built with Spring Boot :: 3.5.6


2025-10-04T14:40:46.144Z  INFO 1 --- [           main] o.s.s.petclinic.PetClinicApplication     : Starting PetClinicApplication v3.5.6-SNAPSHOT using Java 25 with PID 1
    (/workspace/BOOT-INF/classes started by cnb in /workspace)
2025-10-04T14:40:46.147Z  INFO 1 --- [           main] o.s.s.petclinic.PetClinicApplication     : No active profile set, falling back to 1 default profile: "default"
2025-10-04T14:40:47.504Z  INFO 1 --- [           main] .s.d.r.c.RepositoryConfigurationDelegate : Bootstrapping Spring Data JPA repositories in DEFAULT mode.
2025-10-04T14:40:47.565Z  INFO 1 --- [           main] .s.d.r.c.RepositoryConfigurationDelegate : Finished Spring Data repository scanning in 51 ms. Found 3 JPA repository interfaces.
2025-10-04T14:40:48.097Z  INFO 1 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port 8080 (http)
2025-10-04T14:40:48.109Z  INFO 1 --- [           main] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2025-10-04T14:40:48.109Z  INFO 1 --- [           main] o.apache.catalina.core.StandardEngine    : Starting Servlet engine: [Apache Tomcat/10.1.46]
2025-10-04T14:40:48.147Z  INFO 1 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext
2025-10-04T14:40:48.148Z  INFO 1 --- [           main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 1909 ms
2025-10-04T14:40:48.341Z  INFO 1 --- [           main] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Starting...
2025-10-04T14:40:48.544Z  INFO 1 --- [           main] com.zaxxer.hikari.pool.HikariPool        : HikariPool-1 - Added connection conn0: url=jdbc:h2:mem:c7d856d8-a8cd-48da-9083-87da64aadace user=SA
2025-10-04T14:40:48.545Z  INFO 1 --- [           main] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Start completed.
2025-10-04T14:40:48.679Z  INFO 1 --- [           main] o.hibernate.jpa.internal.util.LogHelper  : HHH000204: Processing PersistenceUnitInfo [name: default]
2025-10-04T14:40:48.715Z  INFO 1 --- [           main] org.hibernate.Version                    : HHH000412: Hibernate ORM core version 6.6.29.Final
2025-10-04T14:40:48.739Z  INFO 1 --- [           main] o.h.c.internal.RegionFactoryInitiator    : HHH000026: Second-level cache disabled
2025-10-04T14:40:48.949Z  INFO 1 --- [           main] o.s.o.j.p.SpringPersistenceUnitInfo      : No LoadTimeWeaver setup: ignoring JPA class transformer
2025-10-04T14:40:49.886Z  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)
2025-10-04T14:40:49.889Z  INFO 1 --- [           main] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit 'default'
2025-10-04T14:40:50.138Z  INFO 1 --- [           main] o.s.d.j.r.query.QueryEnhancerFactory     : Hibernate is in classpath; If applicable, HQL parser will be used.
2025-10-04T14:40:50.984Z  INFO 1 --- [           main] o.s.b.a.e.web.EndpointLinksResolver      : Exposing 13 endpoints beneath base path '/actuator'
2025-10-04T14:40:51.058Z  INFO 1 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port 8080 (http) with context path '/'
2025-10-04T14:40:51.080Z  INFO 1 --- [           main] o.s.s.petclinic.PetClinicApplication     : Started PetClinicApplication in 5.397 seconds (process running for 5.761)
```

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>`
  - 60s 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 17.0.16
- Java Memory Calculator

```
sdk use java 17.0.16-librca

mvn spring-boot:build-image

docker run -p 8080:8080 -t spring-petclinic:3.5.6-SNAPSHOT
```

	Build	Image	Startup	Initial RAM	Requests/s	RAM	50%	75%	90%	99%	99.9%
⏱️ Java 17	105s	287MB	8.232s	260MB	86/s	352MB	499ms	770ms	1125ms	1861ms	2453ms

No Optimizing - Baseline JDK 21

- Spring PetClinic (JDK 21 adjustments)
- Bellsoft Liberica JDK 21.0.8
- Java Memory Calculator

```
sdk use java 21.0.8-librca

mvn -Djava.version=21 spring-boot:build-image \
    -Dspring-boot.build-image.imageName=spring-petclinic:3.5.6-SNAPSHOT-jdk21

docker run -p 8080:8080 -t spring-petclinic:3.5.6-SNAPSHOT-jdk21
```

	Build	Image	Startup	Initial RAM	Requests/s	RAM	50%	75%	90%	99%	99.9%
⏱️ Java 17	105s	287MB	8.232s	260MB	86/s	352MB	499ms	770ms	1125ms	1861ms	2453ms
Java 21	104s	315MB	8.107s	264MB	98/s	355MB	439ms	644ms	965ms	1769ms	2631ms

No Optimizing - Baseline JDK 25

- Spring PetClinic (JDK 25 adjustments)
- Bellsoft Liberica JDK 25
- Java Memory Calculator

```
sdk use java 25-librca

mvn -Djava.version=25 spring-boot:build-image \
    -Dspring-boot.build-image.imageName=spring-petclinic:3.5.6-SNAPSHOT-jdk25

docker run -p 8080:8080 -t spring-petclinic:3.5.6-SNAPSHOT-jdk25
```

	Build	Image	Startup	Initial RAM	Requests/s	RAM	50%	75%	90%	99%	99.9%
⏱️ Java 17	105s	287MB	8.232s	260MB	86/s	352MB	499ms	770ms	1125ms	1861ms	2453ms
Java 25	118s	383MB	8.706s	255MB	73/s	397MB	660ms	931ms	1198ms	1805ms	2263ms

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:3.5.6-SNAPSHOT
```

	Build	Image	Startup	Initial RAM	Requests/s	RAM	50%	75%	90%	99%	99.9%
⏱️ Java 17	105s	287MB	8.232s	260MB	86/s	352MB	499ms	770ms	1125ms	1861ms	2453ms
Java 17	-	-	7.396s	246MB	94/s	340MB	493ms	698ms	963ms	1579ms	2223ms
Java 21	-	-	7.538s	251MB	101/s	359MB	442ms	635ms	843ms	1437ms	2034ms
Java 25	-	-	7.353s	242MB	71/s	373MB	633ms	955ms	1277ms	2060ms	2540ms

-XX:+UseCompactObjectHeaders

-XX:+UseCompactObjectHeaders

The Compact Object Headers (JEP 384) feature reduces the memory footprint of Java objects by using a more compact representation for object headers. This can lead to improved cache performance and reduced memory usage.

```
docker run -p 8080:8080 -e "JAVA_TOOL_OPTIONS=-XX:+UseCompactObjectHeaders" \
  -t spring-petclinic:3.5.6-SNAPSHOT
```

	Build	Image	Startup	Initial RAM	Requests/s	RAM	50%	75%	90%	99%	99.9%
⏱️ Java 17	105s	287MB	8.232s	260MB	86/s	352MB	499ms	770ms	1125ms	1861ms	2453ms
Java 25	-	-	8.288s	256MB	64/s	390MB	742ms	1037ms	1339ms	2003ms	2585ms

-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:3.5.6-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	105s	287MB	8.232s	260MB	86/s	352MB	499ms	770ms	1125ms	1861ms	2453ms
Java 17	-	-	6.307s	211MB	86/s	284MB	592ms	698ms	799ms	998ms	1129ms
Java 21	-	-	6.579s	219MB	88/s	297MB	581ms	691ms	796ms	1001ms	1168ms
Java 25	-	-	5.978s	214MB	114/s	370MB	414ms	590ms	703ms	956ms	1158ms

-XX:+UseParallelGC

-XX:+UseParallelGC

The Parallel Garbage Collector (Parallel GC) is a high-throughput garbage collector that is designed to take advantage of multiple processors. It performs minor garbage collections in parallel, which can lead to shorter pause times and improved application performance.

```
docker run -p 8080:8080 -e "JAVA_TOOL_OPTIONS=-XX:+UseParallelGC" \
  -t spring-petclinic:3.5.6-SNAPSHOT
```

	Build	Image	Startup	Initial RAM	T…	RAM	50%	75%	90%	99%	99.9%
⏱️ Java 17	105s	287MB	8.232s	260MB	86/s	352MB	499ms	770ms	1125ms	1861ms	2453ms
Java 17	-	-	8.264s	312MB	86/s	430MB	501ms	766ms	1072ms	1833ms	2571ms
Java 21	-	-	8.327s	368MB	97/s	439MB	449ms	695ms	939ms	1576ms	2368ms
Java 25	-	-	7.977s	337MB	70/s	384MB	701ms	973ms	1298ms	1967ms	2563ms

-XX:+UseZGC -XX:+ZGenerational

-XX:+UseZGC -XX:+ZGenerational

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

```
docker run -p 8080:8080 -e "JAVA_TOOL_OPTIONS=-XX:+UseZGC -XX:+ZGenerational" \
  -t spring-petclinic:3.5.6-SNAPSHOT
```

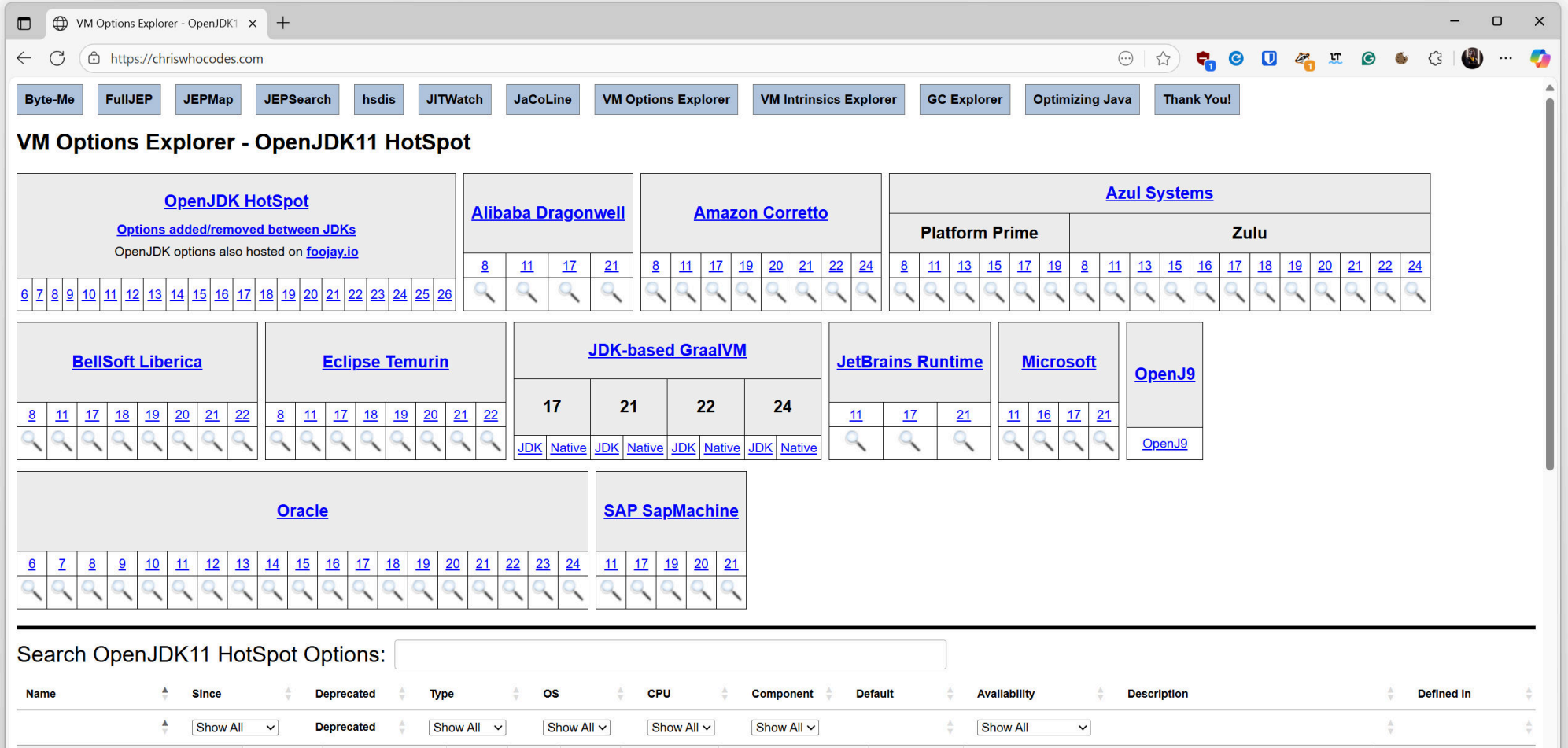
	Build	Image	Startup	Initial RAM	T…	RAM	50%	75%	90%	99%	99.9%
⏱ Java 17	105s	287MB	8.232s	260MB	86/s	352MB	499ms	770ms	1125ms	1861ms	2453ms
Java 21	-	-	8.321s	487MB	93/s	653MB	492ms	704ms	1004ms	1794ms	2595ms
Java 25	-	-	8.109s	495MB	81/s	641MB	566ms	828ms	1102ms	1803ms	2273ms

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

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:3.5.6-SNAPSHOT
```

	Build	Image	Startup	Initial RAM	Requests/s	RAM	50%	75%	90%	99%	99.9%
⏱ Java 17	105s	287MB	8.232s	260MB	86/s	352MB	499ms	770ms	1125ms	1861ms	2453ms
Java 17	-	-	7.344s	243MB	83/s	338MB	533ms	800ms	1122ms	1916ms	2566ms

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:3.5.6-SNAPSHOT
```

	Build	Image	Startup	Initial RAM	Requests/s	RAM	50%	75%	90%	99%	99.9%
⏱️ Java 17	105s	287MB	8.232s	260MB	86/s	352MB	499ms	770ms	1125ms	1861ms	2453ms
Java 17	-	-	8.173s	263MB	85/s	349MB	504ms	767ms	1062ms	1796ms	2597ms

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: 463
  - Spring Pet Clinic without actuators: 283 🔥

```
sdk use java 17.0.16-librca

mvn spring-boot:build-image
```

	Build	Image	Startup	Initial RAM	Requests/s	RAM	50%	75%	90%	99%	99.9%
⏱️ Java 17	105s	287MB	8.232s	260MB	86/s	352MB	499ms	770ms	1125ms	1861ms	2453ms
Java 17	100s	283MB	7.434s	248MB	91/s	340MB	483ms	730ms	1015ms	1769ms	2494ms
Java 21	104s	312MB	7.186s	244MB	94/s	342MB	468ms	724ms	1000ms	1708ms	2489ms
Java 25	101s	380MB	7.167s	239MB	95/s	401MB	518ms	703ms	906ms	1375ms	1835ms

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 17.0.16-librca

mvn spring-boot:build-image

docker run -p 8080:8080 -e spring.aot.enabled=true \
  -t spring-petclinic-aot:3.5.6-SNAPSHOT
```

	Build	Image	Startup	Initial RAM	Requests/s	RAM	50%	75%	90%	99%	99.9%
⏱️ Java 17	105s	287MB	8.232s	260MB	86/s	352MB	499ms	770ms	1125ms	1861ms	2453ms
Java 17	113s	288MB	7.627s	251MB	93/s	338MB	493ms	702ms	991ms	1651ms	2393ms
Java 21	111s	316MB	7.12s	253MB	87/s	346MB	499ms	757ms	1085ms	1868ms	2653ms
Java 25	121s	384MB	7.132s	243MB	69/s	407MB	701ms	1001ms	1302ms	1928ms	2436ms

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:3.5.6-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>2.1.0</version>
  <executions>
    <execution>
      <goals>
        <goal>depclean</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

```
mvn se.kth.castor:depclean-maven-plugin:2.1.0: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 [10]:
        com.h2database:h2:2.3.232:runtime (2 MB)
        com.mysql:mysql-connector-j:9.4.0:runtime (2 MB)
        org.postgresql:postgresql:42.7.7:runtime (1 MB)
        com.github.ben-manes.caffeine:caffeine:3.2.2:compile (874 KB)
        jakarta.xml.bind:jakarta.xml.bind-api:4.0.2:compile (127 KB)
        javax.cache:cache-api:1.1.1:compile (50 KB)
        ...
        org.webjars:webjars-locator-lite:1.1.1:compile (8 KB)
USED TRANSITIVE DEPENDENCIES [91]:
        org.testcontainers:testcontainers:1.21.3:test (16 MB)
        org.hibernate.orm:hibernate-core:6.6.29.Final:compile (11 MB)
        net.bytebuddy:byte-buddy:1.17.7:runtime (8 MB)
        org.apache.tomcat.embed:tomcat-embed-core:10.1.46:compile (3 MB)
        com.github.docker-java:docker-java-transport-zerodep:3.4.2:test (2 MB)
        org.aspectj:aspectjweaver:1.9.24:compile (2 MB)
        org.springframework:spring-web:6.2.11:compile (1 MB)
        org.springframework.boot:spring-boot-autoconfigure:3.5.6:compile (1 MB)
        org.springframework:spring-core:6.2.11:compile (1 MB)
        org.springframework.boot:spring-boot:3.5.6:compile (1 MB)
        net.java.dev.jna:jna:5.13.0:test (1 MB)
        org.springframework.data:spring-data-jpa:3.5.4:compile (1 MB)
        com.fasterxml.jackson.core:jackson-databind:2.19.2:test (1 MB)
        org.springframework.data:spring-data-commons:3.5.4:compile (1 MB)
        org.assertj:assertj-core:3.27.4:test (1 MB)
        org.springframework:spring-context:6.2.11:compile (1 MB)
        org.hibernate.validator:hibernate-validator:8.0.3.Final:compile (1 MB)
        ...
USED INHERITED DIRECT DEPENDENCIES [0]:
USED INHERITED TRANSITIVE DEPENDENCIES [0]:
POTENTIALLY UNUSED DIRECT DEPENDENCIES [12]:
        org.webjars.npm:bootstrap:5.3.8:compile (1 MB)
        org.webjars.npm:font-awesome:4.7.0:compile (665 KB)
        org.springframework.boot:spring-boot-docker-compose:3.5.6:test (224 KB)
        org.springframework.boot:spring-boot-devtools:3.5.6:test (199 KB)
        org.springframework.boot:spring-boot-starter-web:3.5.6:compile (4 KB)
        org.springframework.boot:spring-boot-starter-test:3.5.6:test (4 KB)
        org.springframework.boot:spring-boot-starter-thymeleaf:3.5.6:compile (4 KB)
        org.springframework.boot:spring-boot-starter:3.5.6:compile (4 KB)
        org.springframework.boot:spring-boot-starter-validation:3.5.6:compile (4 KB)
        ...
POTENTIALLY UNUSED TRANSITIVE DEPENDENCIES [7]:
        org.attoparser:attoparser:2.0.7.RELEASE:compile (240 KB)
        org.thymeleaf:thymeleaf-spring6:3.1.3.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:3.5.6:compile (4 KB)
        org.springframework.boot:spring-boot-starter-jdbc:3.5.6:compile (4 KB)
        org.hamcrest:hamcrest-core:3.0:test (2 KB)
POTENTIALLY UNUSED INHERITED DIRECT DEPENDENCIES [0]:
POTENTIALLY UNUSED INHERITED TRANSITIVE DEPENDENCIES [0]:
[INFO] Analysis done in 0min 24s
```

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 17.0.16-librca

mvn spring-boot:build-image
```

	Build	Image	Startup	Initial RAM	Requests/s	RAM	50%	75%	90%	99%	99.9%
⏱️ Java 17	105s	287MB	8.232s	260MB	86/s	352MB	499ms	770ms	1125ms	1861ms	2453ms
Java 17	102s	281MB	6.9s	233MB	100/s	325MB	433ms	669ms	969ms	1696ms	2430ms
Java 21	103s	309MB	7.04s	238MB	87/s	332MB	501ms	768ms	1136ms	1961ms	2633ms
Java 25	102s	377MB	7.052s	233MB	79/s	375MB	571ms	866ms	1191ms	1826ms	2233ms

More Application Optimisations

More Application Optimisations

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

Even More JVM Optimisations

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	105s	287MB	8.232s	260MB	86/s	352MB	499ms	770ms	1125ms	1861ms	2453ms
Java 17	120s	198MB	8.019s	268MB	92/s	371MB	496ms	728ms	966ms	1613ms	2268ms
Java 17	123s	206MB	7.854s	275MB	100/s	371MB	454ms	669ms	963ms	1661ms	2301ms
Java 25	131s	209MB	8.438s	266MB	72/s	360MB	683ms	942ms	1269ms	1905ms	2591ms

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-3.5.6-SNAPSHOT.jar

jdeps --ignore-missing-deps -q  \
    --recursive  \
    --multi-release 17  \
    --print-module-deps  \
    --class-path 'BOOT-INF/lib/*'  \
    target/spring-petclinic-3.5.6-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	105s	287MB	8.232s	260MB	86/s	352MB	499ms	770ms	1125ms	1861ms	2453ms
Java 17	117s	187MB	8.349s	265MB	85/s	359MB	504ms	790ms	1076ms	1888ms	2626ms
Java 21	118s	194MB	8.458s	277MB	89/s	364MB	499ms	733ms	1039ms	1931ms	2697ms
Java 25	128s	197MB	7.732s	258MB	73/s	359MB	662ms	941ms	1262ms	1866ms	2333ms

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	105s	287MB	8.232s	260MB	86/s	352MB	499ms	770ms	1125ms	1861ms	2453ms
Java 17	122s	449MB	4.986s	236MB	106/s	338MB	405ms	658ms	905ms	1563ms	2073ms
Java 21	122s	475MB	4.961s	233MB	106/s	347MB	400ms	657ms	901ms	1576ms	2265ms
Java 25	133s	542MB	4.661s	229MB	87/s	365MB	528ms	769ms	1068ms	1731ms	2298ms

I ❤️ Spring Boot 🍃 & Buildpacks 🚀

AOT Caching (Java 25+ Project Leyden)

AOT Caching

Project Leyden is an OpenJDK project that aims to improve the startup time and reduce the memory footprint of Java applications by introducing AOT (Ahead-of-Time) compilation and class data sharing (CDS) enhancements.

Training the AOT cache requires two additional JVM flags:

```
-XX:AOTCacheOutput=app.aot -Dspring.context.exit=onRefresh
```

	Build	Image	Startup	Initial RAM	Requests/s	RAM	50%	75%	90%	99%	99.9%
⏱️ Java 17	105s	287MB	8.232s	260MB	86/s	352MB	499ms	770ms	1125ms	1861ms	2453ms
Java 25	135s	494MB	5.613s	233MB	108/s	351MB	450ms	603ms	767ms	1197ms	1618ms

Other Runtimes

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>paketobuildpacks/eclipse-openj9:latest</buildpack>
            <!-- Used to inherit all the other Java buildpacks -->
            <buildpack>paketobuildpacks/java:latest</buildpack>
        </buildpacks>
    </image>
  </configuration>
</plugin>
```

	Build	Image	Startup	Initial RAM	Requests/s	RAM	50%	75%	90%	99%	99.9%
⏱ Java 17	105s	287MB	8.232s	260MB	86/s	352MB	499ms	770ms	1125ms	1861ms	2453ms
Java 17	136s	360MB	8.709s	184MB	40/s	269MB	1236ms	1699ms	1972ms	2464ms	2867ms
Java 21	138s	379MB	9.05s	191MB	39/s	267MB	1257ms	1707ms	2054ms	2531ms	2881ms

GraalVM

GraalVM CE

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

	Build	Image	Startup	Initial RAM	Requests/s	RAM	50%	75%	90%	99%	99.9%
⏱️ Java 17	105s	287MB	8.232s	260MB	86/s	352MB	499ms	770ms	1125ms	1861ms	2453ms
Java 17	140s	685MB	8.427s	284MB	77/s	399MB	596ms	838ms	1144ms	1835ms	2428ms
Java 21	143s	670MB	8.117s	301MB	60/s	414MB	741ms	1101ms	1563ms	2606ms	3424ms
Java 25	144s	857MB	8.082s	313MB	83/s	419MB	538ms	827ms	1103ms	1729ms	2247ms

Oracle JDK

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

	Build	Image	Startup	Initial RAM	Requests/s	RAM	50%	75%	90%	99%	99.9%
⏱️ Java 17	105s	287MB	8.232s	260MB	86/s	352MB	499ms	770ms	1125ms	1861ms	2453ms
Java 21	127s	454MB	8.138s	265MB	89/s	371MB	498ms	709ms	1006ms	1800ms	2533ms
Java 25	126s	499MB	8.305s	258MB	79/s	407MB	600ms	832ms	1091ms	1634ms	2079ms

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	105s	287MB	8.232s	260MB	86/s	352MB	499ms	770ms	1125ms	1861ms	2453ms
Java 17 (m)	105s	194MB	7.732s	289MB	98/s	399MB	433ms	668ms	969ms	1837ms	2478ms
Java 21 (g)	103s	207MB	8.227s	279MB	94/s	368MB	464ms	676ms	1007ms	1868ms	2737ms
Java 25 (g)	119s	384MB	7.929s	255MB	69/s	393MB	697ms	1001ms	1333ms	2002ms	2635ms

Buildpack Jammy 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	105s	287MB	8.232s	260MB	86/s	352MB	499ms	770ms	1125ms	1861ms	2453ms
Java 17	104s	288MB	8.283s	260MB	94/s	351MB	488ms	724ms	990ms	1620ms	2389ms
Java 21	106s	315MB	7.873s	261MB	94/s	359MB	490ms	695ms	965ms	1729ms	2561ms
Java 25	116s	384MB	8.223s	255MB	75/s	396MB	630ms	927ms	1202ms	1859ms	2375ms

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
```

	Build	Image	Startup	Initial RAM	Requests/s	RAM	50%	75%	90%	99%	99.9%
⏱ Java 17	105s	287MB	8.232s	260MB	86/s	352MB	499ms	770ms	1125ms	1861ms	2453ms
CE 21	339s	248MB	0.474s	228MB	181/s	453MB	274ms	306ms	403ms	691ms	966ms
CE 25	268s	232MB	0.502s	219MB	176/s	436MB	285ms	318ms	427ms	707ms	1002ms
OG 21	395s	255MB	0.413s	206MB	231/s	410MB	199ms	280ms	310ms	596ms	790ms
OG 25	322s	232MB	0.344s	183MB	214/s	414MB	202ms	295ms	391ms	607ms	898ms

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-3.5.6-SNAPSHOT.jar
```

```
jcmd target/spring-petclinic-crac-3.5.6-SNAPSHOT.jar JDK.checkpoint
```

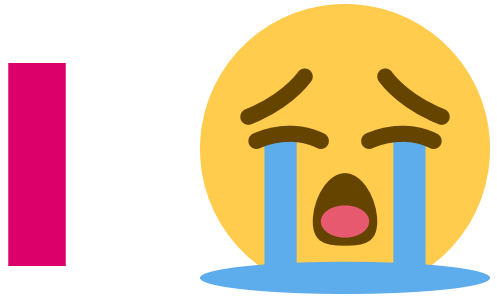
```
java -XX:CRaCRestoreFrom=crac-files
```

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 21.0.8-librca

mvn spring-boot:build-image

docker run -e spring.threads.virtual.enabled=true \
```

	Build	Image	Startup	Initial RAM	Requests/s	RAM	50%	75%	90%	99%	99.9%
⏱️ Java 17	105s	287MB	8.232s	260MB	86/s	352MB	499ms	770ms	1125ms	1861ms	2453ms
Java 21	-	-	8.425s	263MB	172/s	346MB	290ms	306ms	364ms	603ms	2105ms
Java 24	-	-	8.311s	256MB	666/s	349MB	59ms	88ms	111ms	242ms	2004ms
Java 25	-	-	8.276s	254MB	657/s	362MB	54ms	86ms	109ms	427ms	3593ms

Virtual Threads Java 21

CPU	VT	Startup	Initial RAM	Request/s	RAM	50%	75%	90%	99%	99.9%	99.99%
4 vCPU		6.543s	278MB	293/s	392MB	133ms	207ms	309ms	637ms	941ms	1275ms
4 vCPU		6.479s	288MB	390/s	390MB	114ms	150ms	181ms	194ms	621ms	2346ms
2 vCPU		8.574s	262MB	66/s	352MB	636ms	1007ms	1469ms	2452ms	3312ms	4098ms
2 vCPU		8.359s	262MB	164/s	347MB	296ms	332ms	400ms	572ms	868ms	2204ms
1 vCPU		23.083s	263MB	22/s	348MB	2133ms	3034ms	3967ms	6200ms	8011ms	8361ms
1 vCPU		23.159s	262MB	50/s	350MB	962ms	1196ms	1366ms	1665ms	2878ms	6868ms
0.5 vCPU		49.063s	258MB	10/s	337MB	4299ms	6701ms	9344ms	16138ms	21606ms	21606ms
0.5 vCPU		50.463s	258MB	15/s	329MB	3302ms	3865ms	4414ms	7873ms	14262ms	14262ms
0.25 vCPU		108.793s	259MB	2/s	335MB	30324ms	43535ms	48836ms	56330ms	56330ms	56330ms
0.25 vCPU		107.764s	259MB	5/s	317MB	10337ms	11799ms	15141ms	20259ms	20991ms	20991ms

Virtual Threads Java 24

CPU	VT	Startup	Initial RAM	Request/s	RAM	50%	75%	90%	99%	99.9%	99.99%
4 vCPU		6.142s	275MB	326/s	523MB	133ms	204ms	288ms	459ms	603ms	734ms
4 vCPU		6.509s	275MB	802/s	483MB	56ms	87ms	120ms	190ms	262ms	341ms
2 vCPU		8.335s	251MB	87/s	394MB	497ms	797ms	1136ms	1994ms	2742ms	3369ms
2 vCPU		8.528s	255MB	702/s	349MB	53ms	80ms	100ms	420ms	2055ms	2976ms
1 vCPU		23.021s	253MB	24/s	343MB	1967ms	2834ms	3768ms	5804ms	7654ms	8270ms
1 vCPU		24.312s	253MB	188/s	342MB	222ms	304ms	493ms	875ms	1186ms	1221ms
0.5 vCPU		49.831s	248MB	10/s	316MB	4467ms	6894ms	9695ms	15111ms	19059ms	19059ms
0.5 vCPU		50.501s	251MB	20/s	317MB	2272ms	2769ms	3471ms	5870ms	9721ms	12095ms
0.25 vCPU		112.161s	250MB	2/s	312MB	48899ms	53993ms	56430ms	58705ms	58705ms	58705ms
0.25 vCPU		108.235s	246MB	6/s	295MB	9114ms	10166ms	12534ms	17828ms	21275ms	21275ms

Virtual Threads Java 25

CPU	VT	Startup	Initial RAM	Request/s	RAM	50%	75%	90%	99%	99.9%	99.99%
4 vCPU		6.002s	288MB	390/s	390MB	114ms	150ms	181ms	194ms	621ms	2346ms
4 vCPU		6.236s	277MB	627/s	669MB	64ms	116ms	172ms	283ms	416ms	587ms
2 vCPU		8.359s	262MB	164/s	347MB	296ms	332ms	400ms	572ms	868ms	2204ms
2 vCPU		8.198s	258MB	727/s	362MB	53ms	63ms	95ms	190ms	2753ms	3985ms
1 vCPU		23.159s	262MB	50/s	350MB	962ms	1196ms	1366ms	1665ms	2878ms	6868ms
1 vCPU		22.67s	255MB	195/s	342MB	219ms	301ms	411ms	896ms	1203ms	1279ms
0.5 vCPU		50.463s	258MB	15/s	329MB	3302ms	3865ms	4414ms	7873ms	14262ms	14262ms
0.5 vCPU		49.13s	251MB	21/s	317MB	1936ms	2533ms	3282ms	4849ms	20245ms	21156ms
0.25 vCPU		107.764s	259MB	5/s	317MB	10337ms	11799ms	15141ms	20259ms	20991ms	20991ms
0.25 vCPU		108.601s	253MB	6/s	303MB	8801ms	9901ms	11161ms	18049ms	28540ms	28540ms

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 21.0.8-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:3.5.6-SNAPSHOT
```

	VT	Build	Image	Startup	Initial RAM	Requests/s	RAM	50%	75%	90%	99%	99.9%
⏱️ Java 17		105s	287MB	8.232s	260MB	86/s	352MB	499ms	770ms	1125ms	1861ms	2453ms
Java 21	🧵	129s	220MB	5.453s	234MB	117/s	369MB	367ms	572ms	862ms	1490ms	2031ms
Java 25	🧵	106s	195MB	5.771s	244MB	74/s	440MB	703ms	1005ms	1306ms	1937ms	2434ms

Java 21 & Base Builder Tiny

```
sdk use java 21.0.8-librca

mvn spring-boot:build-image

docker run -p 8080:8080 \
  -e spring.threads.virtual.enabled=true \
  -e spring.main.lazy-initialization=true \
```

	VT	Build	Image	Startup	Initial RAM	Requests/s	RAM	50%	75%	90%	99%	99.9%
Java 17		105s	287MB	8.232s	260MB	86/s	352MB	499ms	770ms	1125ms	1861ms	2453ms
Java 21		135s	334MB	5.708s	226MB	83/s	348MB	531ms	831ms	1168ms	1936ms	2560ms
Java 25		148s	335MB	5.756s	222MB	99/s	435MB	483ms	696ms	934ms	1496ms	1942ms
Java 25	AOTC	149s	291MB	6.433s	230MB	107/s	443MB	439ms	605ms	798ms	1140ms	1561ms

Did I miss something? 🧐

Let me/us know! 🙋

… or get in touch later! 🙊

Spring Boot in the Cloud

Advanced Optimization Deep Dive

Patrick Baumgartner
42talents GmbH, Zürich, Switzerland

@patbaumgartner
patrick.baumgartner@42talents.com
bit.ly/48i5U9x