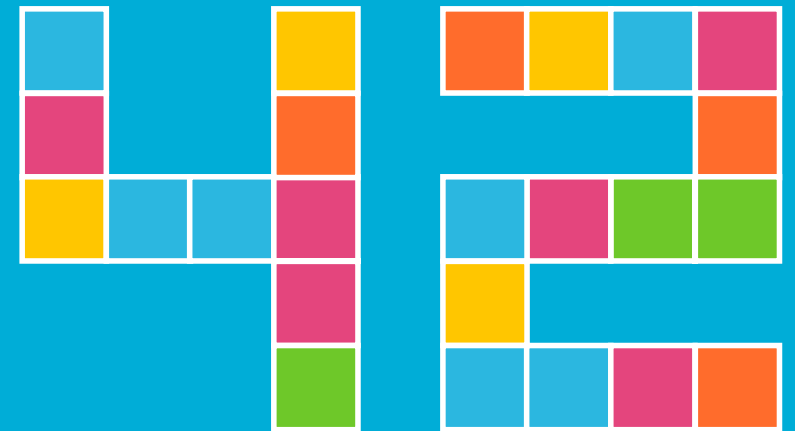


Lean Spring Boot

Applications for The Cloud

Patrick Baumgartner
42talents GmbH, Zürich, Switzerland

@patbaumgartner
patrick.baumgartner@42talents.com



TALENTS

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-context-indexer", classpath exclusions, lazy spring beans, actuator, JMX. In addition, we also look at switching to a different JVM and other tools.

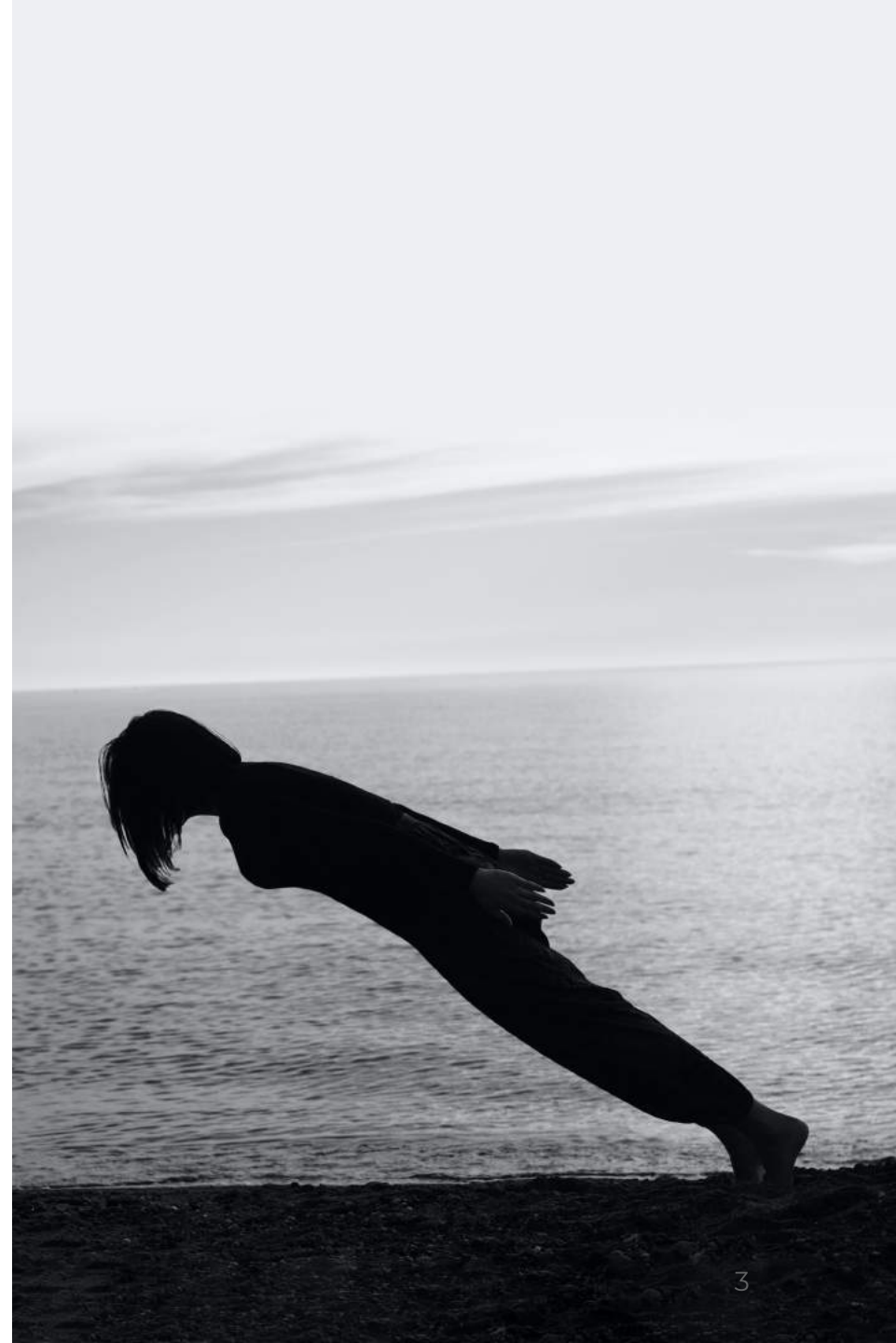
Let's make Spring Boot great again!

Lean Spring Boot

Applications for The Cloud

Patrick Baumgartner
42talents GmbH, Zürich, Switzerland

@patbaumgartner
patrick.baumgartner@42talents.com





WARNING:

Numbers shown in is 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?

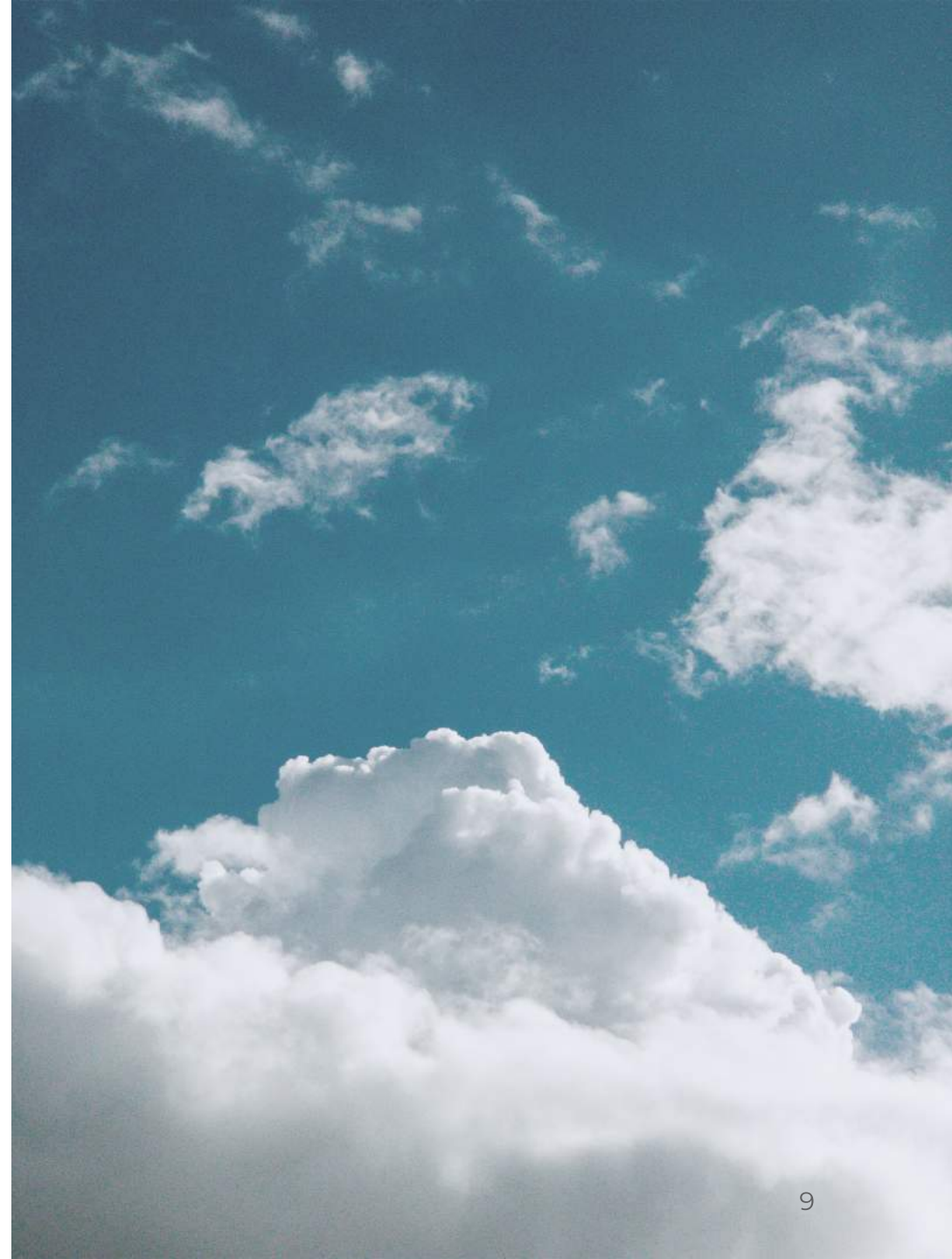
JAVA 🤔 & Spring Boot ❤️



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?



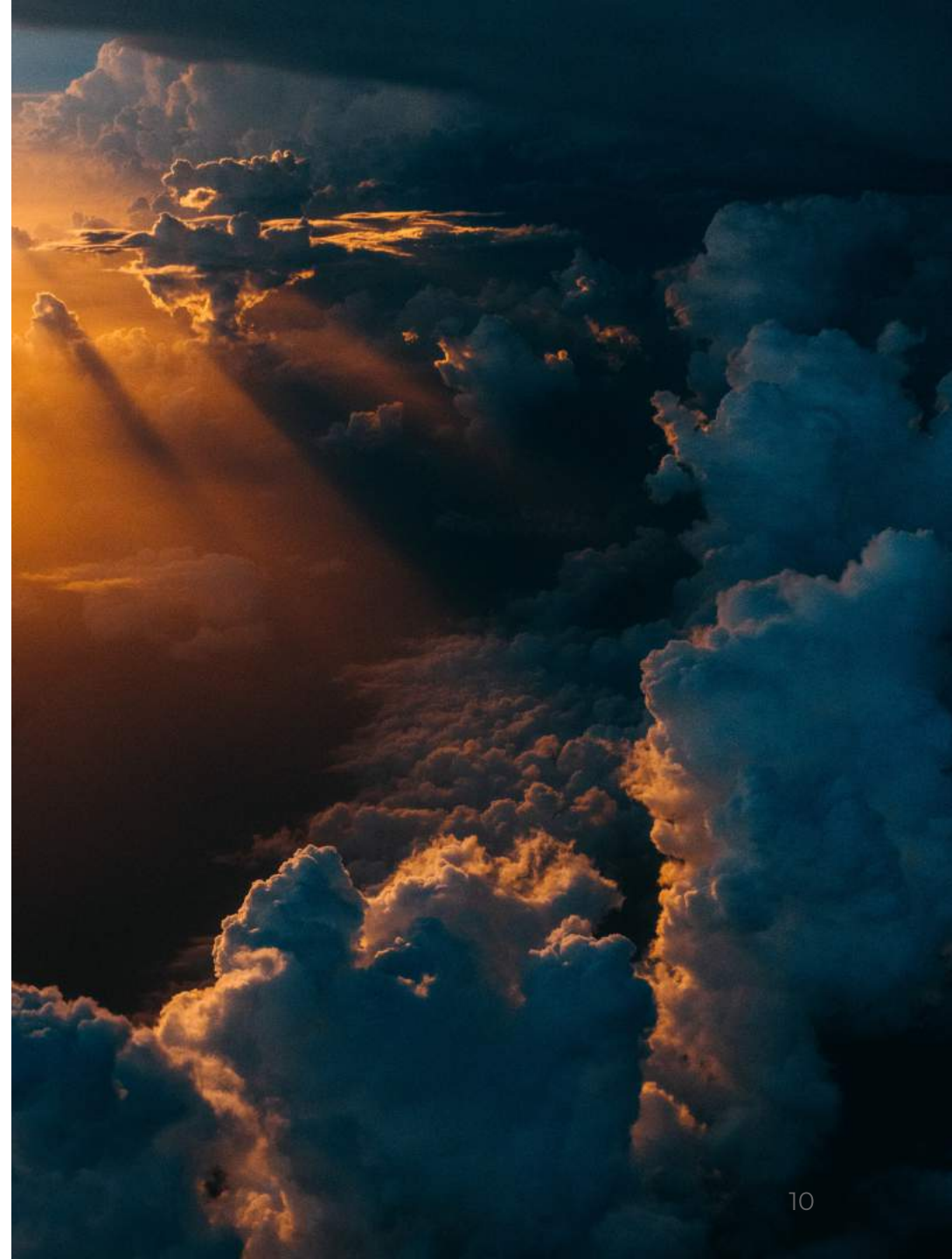
Considerations

Resources

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

Scaleability

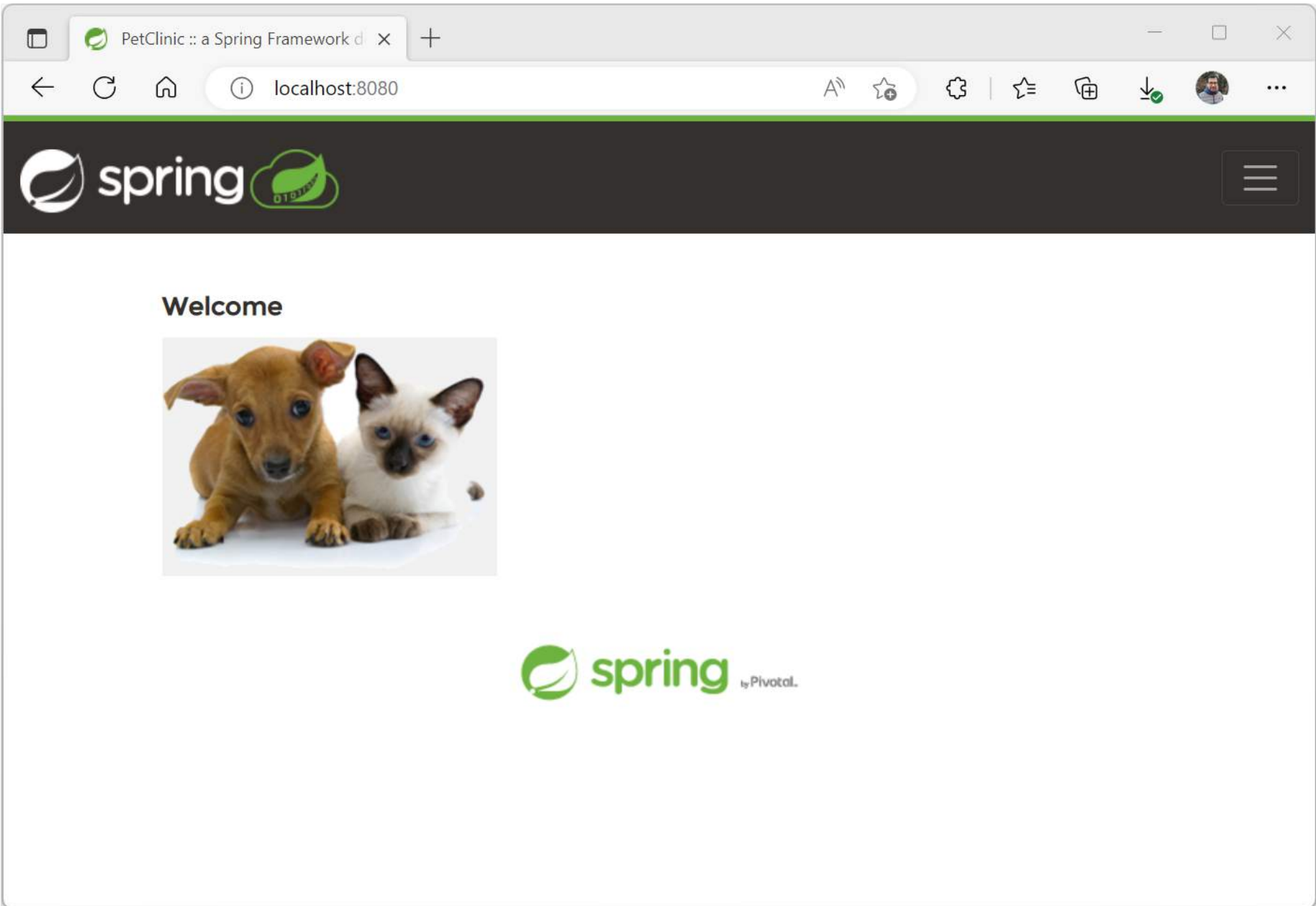
- Fast Startup
- Graceful Shutdown



Agenda

Agenda

- Spring PetClinic & Baseline
- Java Optimizations
- Spring Boot Optimizations
- Application Optimizations
- Other Runtimes
- Conclusions
- Everything Combined (OpenJDK Example)



PetClinic :: a Spring Framework d

×

Spring Petclinic community

×

+

←

→

↻

🏠

ⓘ

localhost:8080/vets.html

A

🌟

⚙️



🌟

📁

⬇️

👤


⋮


 **spring** 

☰

Veterinarians

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

Pages: [1 [2](#)] 

 **spring** by Pivotal

15

Spring Petclinic Community

- spring-framework-petclinic
- spring-petclinic-angular
- 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

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 ...

Optimizing Experiments

Baseline

Technology Stack

- OCI Container (Buildpacks)
- Java JRE 17 LTS
- Spring Boot 3.0.2
- Initialize SQL Scripts

Examination

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



paketo
buildpacks

Buildpacks FTW!

- Spring Boot plugin uses "Build Packs" during the `build-image` task. It detects the Spring Boot App and optimizes created container:
- Optimizes the runtime by:
 - Extracting the fat jar into exploded form.
 - Calculates and applies resource runtime tuning at container startup.
- Optimized the container image:
 - Adds layer from build pack, spring boot, ...
 - Subsequent builds are faster, they only build and add layers for the changed code.

See also: <https://buildpacks.io/>

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/>

No Optimizing - Baseline JRE 17

- Spring PetClinic (no adjustments)
- Bellsoft Liberica JRE 17.0.6
- Java Memory Calculator

```
sdk use java 17.0.6-tem  
  
mvn spring-boot:build-image  
  
docker run -p 8080:8080 -t spring-petclinic:3.0.0-SNAPSHOT
```

Image Size	Build	RAM	Startup #1	Startup #2	Startup #3
 313MB	43s	390MB	4.229s	3.947s	3.973s


No Optimizing - Baseline JRE 19

- Spring PetClinic (JDK 19 adjustments)
- Bellsoft Liberica JRE 19.0.2
- Java Memory Calculator

```
sdk use java 19.0.2-tem
```

```
mvn -Djava.version=19 spring-boot:build-image \  
    -Dspring-boot.build-image.imageName=spring-petclinic:3.0.0-SNAPSHOT-jdk19
```

```
docker run -p 8080:8080 -t spring-petclinic:3.0.0-SNAPSHOT-jdk19
```

Image Size	Build	RAM	Startup #1	Startup #2	Startup #3
 313MB	43s	390MB	4.229s	3.947s	3.973s
322MB	37s	314MB	3.663s	3.703s	3.676s


-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.

In order to disable C2 and only use 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:3.0.0-SNAPSHOT
```


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

Image Size	Build	RAM	Startup #1	Startup #2	Startup #3
 313MB	43s	390MB	4.229s	3.947s	3.973s
-	-	190MB	3.011s	3.042s	3.214s

Spring Context Indexer (1)

The `spring-context-indexer` artifact generates a `META-INF/spring.components` file that is included in the jar file. When the `ApplicationContext` detects such an index, it automatically uses it rather than scanning the classpath.

```
<dependency>  
  <groupId>org.springframework</groupId>  
  <artifactId>spring-context-indexer</artifactId>  
  <scope>optional</scope>  
</dependency>
```

Image Size	Build	RAM	Startup #1	Startup #2	Startup #3
 313MB	43s	390MB	4.229s	3.947s	3.973s
313MB	47s	352MB	4.039s	3.936s	4.045s

```
sdk use java 17.0.6-tem
```

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

```
docker run -p 8080:8080 -t spring-petclinic-indexer:3.0.0-SNAPSHOT
```

Spring Context Indexer (2)


META-INF/spring.components

```
org.springframework.samples.petclinic.PetClinicApplication=org.springframework.stereotype.Component,org.springframework.boot.SpringBootConfiguration
org.springframework.samples.petclinic.model=package-info
org.springframework.samples.petclinic.model.BaseEntity=jakarta.persistence.MappedSuperclass
org.springframework.samples.petclinic.model.NamedEntity=jakarta.persistence.MappedSuperclass
org.springframework.samples.petclinic.model.Person=jakarta.persistence.MappedSuperclass
org.springframework.samples.petclinic.owner.Owner=jakarta.persistence.Entity,jakarta.persistence.Table
org.springframework.samples.petclinic.owner.OwnerController=org.springframework.stereotype.Component
org.springframework.samples.petclinic.owner.OwnerRepository=org.springframework.data.repository.Repository
org.springframework.samples.petclinic.owner.Pet=jakarta.persistence.Entity,jakarta.persistence.Table
org.springframework.samples.petclinic.owner.PetController=org.springframework.stereotype.Component
org.springframework.samples.petclinic.owner.PetType=jakarta.persistence.Entity,jakarta.persistence.Table
org.springframework.samples.petclinic.owner.PetTypeFormatter=org.springframework.stereotype.Component
org.springframework.samples.petclinic.owner.Visit=jakarta.persistence.Entity,jakarta.persistence.Table
org.springframework.samples.petclinic.owner.VisitController=org.springframework.stereotype.Component
org.springframework.samples.petclinic.system.CacheConfiguration=org.springframework.stereotype.Component
org.springframework.samples.petclinic.system.CrashController=org.springframework.stereotype.Component
org.springframework.samples.petclinic.system.WelcomeController=org.springframework.stereotype.Component
org.springframework.samples.petclinic.vet.Specialty=jakarta.persistence.Entity,jakarta.persistence.Table
org.springframework.samples.petclinic.vet.Vet=jakarta.persistence.Entity,jakarta.persistence.Table
org.springframework.samples.petclinic.vet.VetController=org.springframework.stereotype.Component
org.springframework.samples.petclinic.vet.VetRepository=org.springframework.data.repository.Repository
org.springframework.samples.petclinic.vet.Vets=jakarta.xml.bind.annotation.XmlRootElement
```


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 \  
-t spring-petclinic:3.0.0-SNAPSHOT
```

Image Size	Build	RAM	Startup #1	Startup #2	Startup #3
 313MB	43s	390MB	4.229s	3.947s	3.973s
	-	365MB	3.623s	3.659s	3.801s

Lazy Spring Beans (2)

Pros

- Faster startup usefull 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 missconfigurations unnoticed at startup
- Beans creation errors only be found at the time of loading the bean

No Spring Boot Actuators


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

- No. of Spring Beans
 - Spring Pet Clinic with Actuators: 426
 - Spring Pet Clinic no Actuators: 264 🔥

```
sdk use java 17.0.6-tem
```

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

```
docker run -p 8080:8080 -t spring-petclinic-no-actuator:3.0.0-SNAPSHOT
```

Image Size	Build	RAM	Startup #1	Startup #2	Startup #3
 313MB	43s	390MB	4.229s	3.947s	3.973s
310MB	44s	345MB	3.467s	3.704s	3.536s

Fixing Spring Boot Config Location


Fix the location of the Spring Boot config file(s).

Considered in 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

See also: <https://docs.spring.io/spring-boot/docs/current/reference/html/features.html#features.external-config>

```
docker run -p 8080:8080 -e spring.config.location=classpath:application.properties \
-t spring-petclinic:3.0.0-SNAPSHOT
```


Image Size	Build	RAM	Startup #1	Startup #2	Startup #3
 313MB	43s	390MB	4.229s	3.947s	3.973s
-	-	390MB	4.082s	4.109s	4.148s

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:3.0.0-SNAPSHOT
```

Image Size	Build	RAM	Startup #1	Startup #2	Startup #3
 313MB	43s	390MB	4.229s	3.947s	3.973s
-	-	-	-	-	-



Spring Boot & Buildpacks

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>
```

Dependency Cleanup (2)

But there are some challenges:

- Spring uses reflection to load classes
- Spring Boot uses META-INF/spring-boot/org.springframework.boot.autoconfigure.AutoConfiguration to load classes
- Spring Context Indexer uses META-INF/spring.components
- Component & Entity Scanning through Classpath Scanning

<pre>sdk use java 17.0.6-tem mvn spring-boot:build-image docker run -p 8080:8080 -t spring-petclinic-depclean:3.0.0-SNAPSHOT</pre>	Image Size	Build	RAM	Startup #1	Startup #2	Startup #3
	 313MB	43s	390MB	4.229s	3.947s	3.973s
	303MB	54s	316MB	3.501s	3.407s	3.387s

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>
```

Image Size	Build	RAM	Startup #1	Startup #2	Startup #3
 313MB	43s	390MB	4.229s	3.947s	3.973s
246MB	94s	375MB	3.889s	4.125s	4.129s


```
sdk use java 17.0.6-tem
```

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

```
docker run -p 8080:8080 -t spring-petclinic-jlink:3.0.0-SNAPSHOT
```

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 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>
```

Image Size	Build	RAM	Startup #1	Startup #2	Startup #3
 313MB	43s	390MB	4.229s	3.947s	3.973s
233MB	88s	382MB	4.235s	4.372s	4.293s



Spring Boot & Buildpacks



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

```
<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>
```

Image Size	Build	RAM	Startup #1	Startup #2	Startup #3
 313MB	43s	390MB	4.229s	3.947s	3.973s
301MB	53s	188MB	5.748s	5.565s	5.610s

```
sdk use java 17.0.6-tem
```

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


```
docker run -p 8080:8080 -t spring-petclinic-custom-jvm:3.0.0-SNAPSHOT
```


Eclipse OpenJ9 Optimized

-Xquickstart causes the JIT compiler to run with a subset of optimizations, which can improve the performance of short-running applications.

Use the -Xshareclasses option to enable, disable, or modify class sharing behavior. Class data sharing is enabled by default for bootstrap classes only, *unless your application is running in a container.*

```
docker run -p 8080:8080 -e "JAVA_TOOL_OPTIONS=-Xshareclasses -Xquickstart" \
-t spring-petclinic-custom-jvm:3.0.0-SNAPSHOT
```


Image Size	Build	RAM	Startup #1	Startup #2	Startup #3
 313MB	43s	390MB	4.229s	3.947s	3.973s
301MB	53s	199MB	4.549s	4.424s	4.573s

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:3.0.0-SNAPSHOT
```

Image Size	Build	RAM	Startup #1	Startup #2	Startup #3
 313MB	43s	390MB	4.229s	3.947s	3.973s
199MB	418s	245MB	0.207s	0.207s	0.199s

CRaC - OpenJDK (1)

CRaC (Checkpoint and Restart in Java) is a feature that allows to checkpoint 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+3_linux-x64/  
export PATH=$JAVA_HOME/bin:$PATH
```

```
mvn clean verify
```

```
java -XX:CRaCCheckpointTo=crac-files -jar target/spring-petclinic-crac-2.7.6.jar
```

```
jcmd target/spring-petclinic-crac-2.7.6.jar JDK.checkpoint
```

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

CRaC - OpenJDK (3)

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

- Does not work with Spring Boot 3
 - Only patched Tomcat 9.0.58 available
- Does not work on Windows or on macOS
 - But Ubuntu 20.04 LTS and also WSL2)
- Does not work in Docker containers

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

Conclusions

Conclusions (1)

CPUs

- Your application might not need a full CPU at runtime
- It will need 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)

Bytecode Obfuscating and Shrinking

- Bytecode obfuscation and shrinking can be used to reduce the size of the application
- Unfortunately, I wasn't successful with Proguard and YGuard to get it working with Spring Boot 3

Conclusions (3)

Throughput

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

Conclusions (4)


Other Runtimes

- CRaC for OpenJDK and CRIU for OpenJ9 are promising
- GraalVM Native Image is a great option for Java applications
 - But build times are long
 - Result is different from what you run in your IDE
- Eclipse OpenJ9 is a great option for running apps with less memory
 - But startup times are longer than with HotSpot
- Depending on the distribution, you might have other features
 - Oracle GraalVM Enterprise Edition, Azul Platform Prime, IBM Semeru Runtime, ...

Everything Combined

Everything Combined

- Dependency Cleanup
 - DB Drivers, Spring Boot Actuator, Jackson
- JVM Parameters
- JLink

Image Size	Build	RAM	Startup #1	Startup #2	Startup #3
 313MB	43s	390MB	4.229s	3.947s	3.973s
233MB	85s	221MB	2.648s	2.574s	2.739s

```
sdk use java 17.0.6-tem
```

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

```
docker run -p 8080:8080 \  
-e "JAVA_TOOL_OPTIONS=-XX:TieredStopAtLevel=1" \  
-e spring.main.lazy-initialization=true \  
-e spring.config.location=classpath:application.properties \  
-t spring-petclinic-optimized:3.0.0-SNAPSHOT
```

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

