23.05.2019 // JUG BB

GraalVM: Fast, Polyglot, Native

Jan Stępień
@janstepien

INNOQ

JAN STĘPIEŃ

Senior Consultant

jan.stepien@innoq.com

Functional Programming
Generative Testing
Tight Feedback Loops

INNOQ

Some background

*.java

*.clj

*.scala

*.kt

*.rb

*.java *.clj *.scala

*.kt *.rb

Ahead-of-time compilation

*.class

Ahead-of-time compilation

*.class

Just-in-time compilation

Native machine code

Just-in-time compilation

HotSpot

JIT compiling since 1999

JEP 243

JVM Compiler Interface

JDK 9

Graal

A compiler written in Java

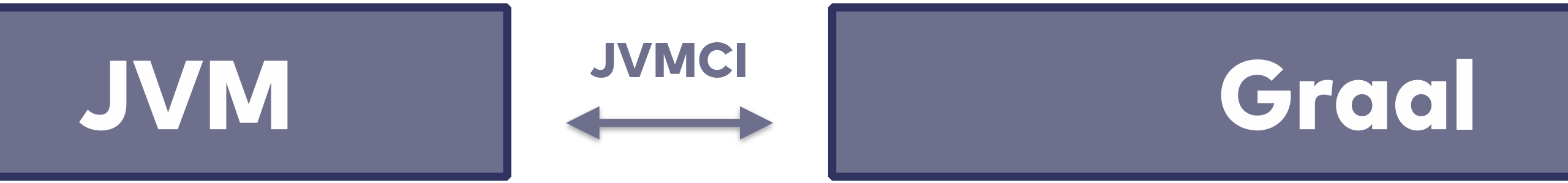

GraalVM™
Run Programs Faster Anywhere
WHY GRAALVM
GET STARTED
JAVA
Standalone
MySQL
GraalVM
OpenJDK
Oracle Database
Node

graalvm.org

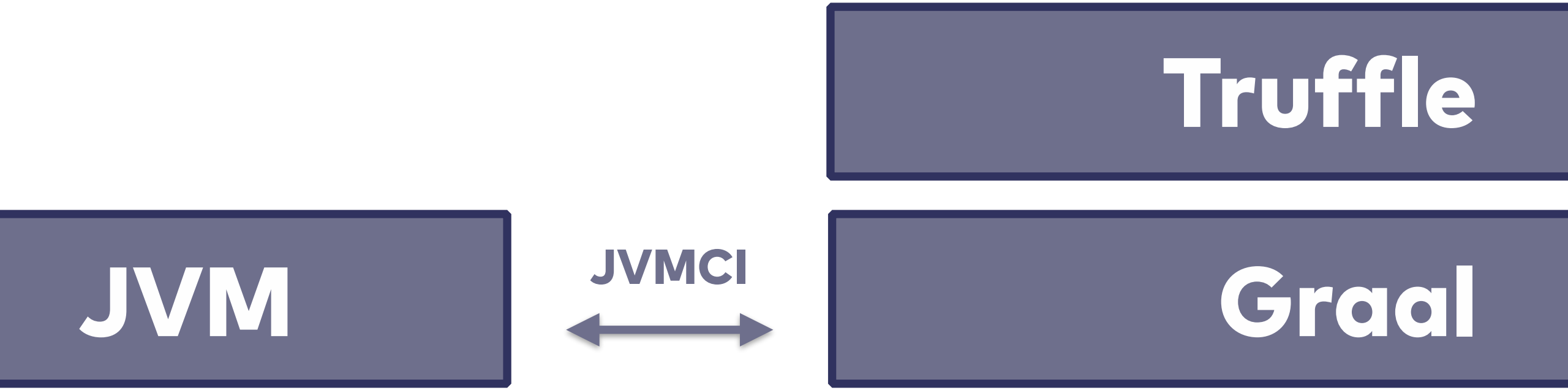
Architecture

GraalVM

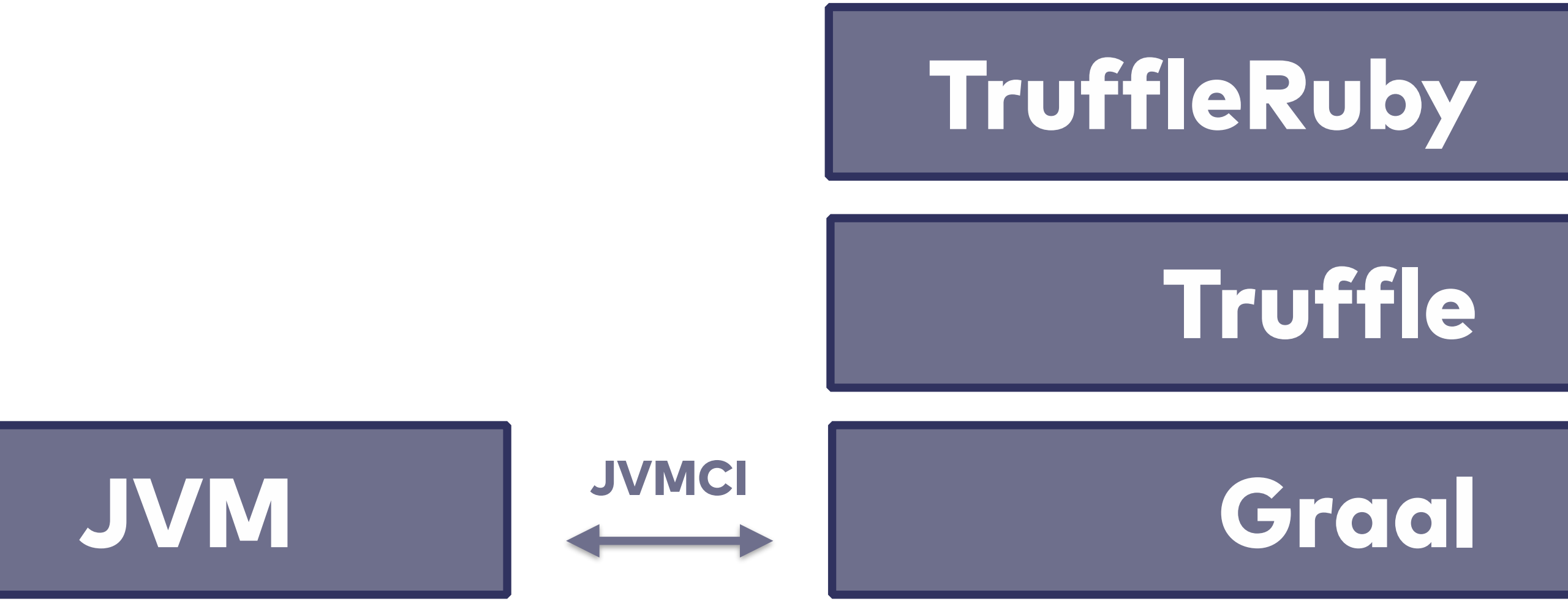
GraalVM



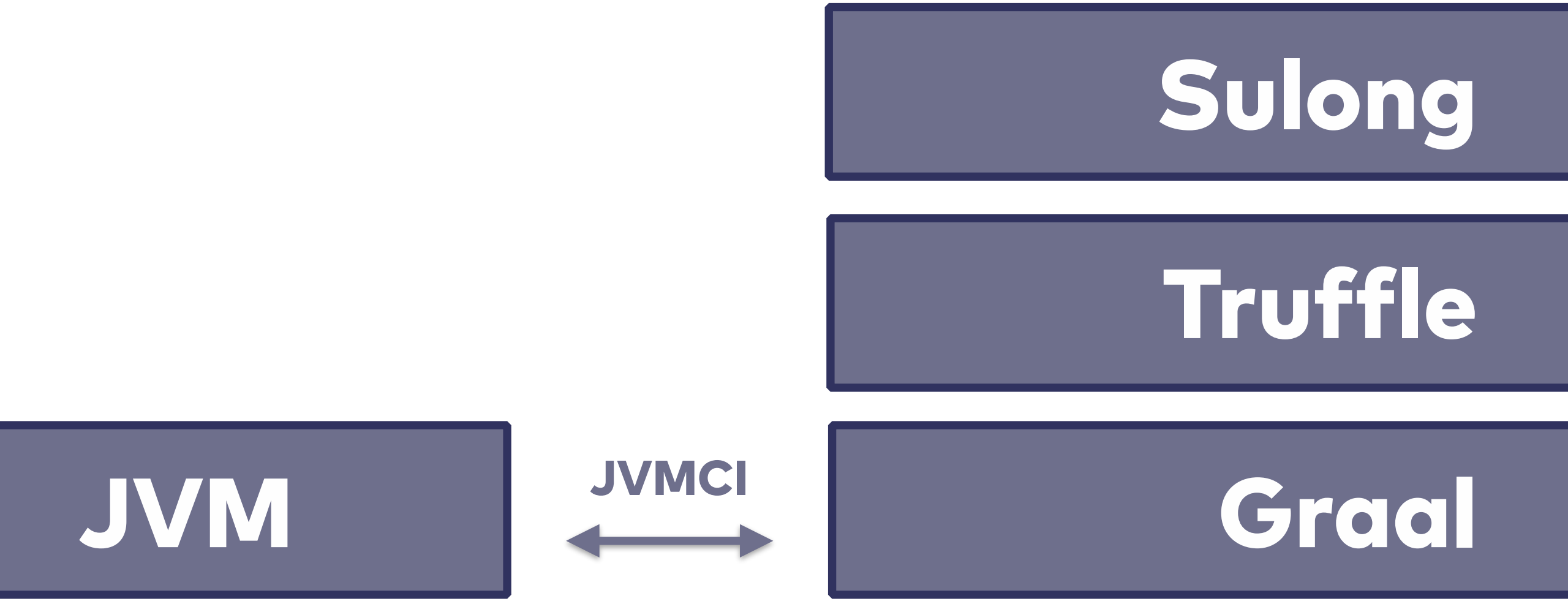
GraalVM



GraalVM



GraalVM



Three main target groups

JVM engineers
Authors of languages
Programmers and end users

Programmers and end users

Polyglot VM and tooling
Faster execution
Native compilation

Native Java

```
$ docker inspect ubuntu \
    | jq .[0].Size
```

```
{
   "a": {
      "b": {
         "c": "d"
      }
   }
}
```

```
$ java jq a b
```

```
{
   "c": "d"
}
```

```
public class jq {
    public static void main(String[] args) {




    }
}
```

```
public class jq {
    public static void main(String[] args) {
        var obj = new JSONObject(new JSONTokener(System.in));



    }
}
```

```
public class jq {
    public static void main(String[] args) {
        var obj = new JSONObject(new JSONTokener(System.in));
        for (String arg: args) {
            obj = obj.getJSONObject(arg);
        }

    }
}
```

```
public class jq {
    public static void main(String[] args) {
        var obj = new JSONObject(new JSONTokener(System.in));
        for (String arg: args) {
            obj = obj.getJSONObject(arg);
        }
        System.out.println(obj);
    }
}
```

```
$ time java jq -Xmx3m a b
{"c": "d"}
    0.14 real    0.13 user    0.03 sys
   31M  maximum resident set size
```

Latest release

vm-19.0.0

0721d34

GraalVM Community Edition 19.0.0

 **ansalond** released this 13 days ago · 577 commits to master since this release

GraalVM is a high-performance, embeddable, polyglot Virtual Machine for running applications written in JavaScript, Python, Ruby, R, JVM-based languages like Java, Scala, Kotlin, and LLVM-based languages such as C and C++.

Additionally, GraalVM allows efficient interoperability between programming languages and compiling Java applications ahead-of-time into native executables for faster startup time and lower memory overhead.

This download includes:

- JVM
- JavaScript Engine & Node.js Runtime
- LLVM Engine
- Developer Tools

The Native Image, Ruby, R and Python plugins are optionally available using the GraalVM `gu` utility.
The complete release notes can be found on the website: http://www.graalvm.org/docs/release-notes/.
The most notable changes since the previous release are:

Java

- We updated the base JDK to 8u212.

```
$ native-image jq
    classlist:   1,196.20 ms
                          ⋮
        image:     693.59 ms
        write:     222.40 ms
      [total]:  12,197.16 ms

$ du jq

3,5M
```

```
$ time java jq -Xmx3m a b
{"c": "d"}
    0.14 real    0.13 user    0.03 sys
   31M  maximum resident set size
```

```
$ time ./jq a b
{"c": "d"}
        0.00 real         0.00 user         0.00 sys
      2M   maximum resident set size
```

```
$ sqlite3 :memory: select 0.1 + 0.2
0.3
```

```
public class h2 {
    public static void main(String[] args) {
        String url = "jdbc:h2:mem:test";
        Connection conn = DriverManager.getConnection(url);
        Statement stmt = conn.createStatement();
        stmt.executeQuery(String.join(" ", args));
        System.out.println(stmt.getResultSet());
    }
}
```

```
$ java -Xmx5m h2 select 0.1 + 0.2

org.h2.result.LocalResultImpl@64c8
columns: 1 rows: 1 pos: -1

    0.42 real    0.68 user    0.06 sys
    54M  maximum resident set size
```

```
$ native-image h2
```

```
$ native-image --no-fallback \
    --rerun-class-init-at-runtime=... \
    --allow-incomplete-classpath \
    -H:ReflectionConfigurationFiles=reflect.json \
    --report-unsupported-elements-at-runtime \
    h2
                                ⋮
      image:   4,331.53 ms
      write:   1,093.98 ms
     [total]: 104,130.32 ms

$ du h2

17M
```

```
$ java -Xmx5m h2 select 0.1 + 0.2

org.h2.result.LocalResultImpl@64c8
columns: 1 rows: 1 pos: -1

    0.42 real    0.68 user    0.06 sys
    54M  maximum resident set size
```

```
$ ./h2 select 0.1 + 0.2

org.h2.result.LocalResultImpl@10c2
columns: 1 rows: 1 pos: -1

    0.01 real    0.00 user    0.00 sys
    7M    maximum resident set size
```

Just-in-time compilation

Slower startup

Higher top speed

High memory usage

Ahead-of-time compilation

Faster startup

Lower top speed

Low memory usage

Dynamically Ahead of Time

```
public class jq {
    public static void main(String[] args) {
        var obj = new JSONObject(new JSONTokener(System.in));
        for (String arg: args) {
            obj = obj.getJSONObject(arg);
        }
        System.out.println(obj);
    }
}
```

```clojure
(ns jq.main
  (:require [clojure.pprint :refer [pprint]])
  (:gen-class))

(defn -main [& path]
  (-> (read *in*)
      (get-in (mapv read-string path))
      pprint))
```

```
$ echo {:a [5, 3]} \
  | time java -jar jq.jar :a 1

3

1.51    real
112MB   maximum resident set size
```

```
$ echo {:a [5, 3]} \
   | java -XX:TieredStopAtLevel=1 \
     -jar jq.jar :a 1
3

1.06    real
96MB    maximum resident set size
```

```
$ echo {:a [5, 3]} \
  | time lumo jq.cljs :a 1

3

0.56    real
128MB   maximum resident set size
```

```
$ native-image -jar jq.jar
   classlist:   3,596.11 ms
                       (...)
       image:   4,671.96 ms
       write:  14,763.97 ms
     [total]: 122,614.21 ms
```

```
$ echo {:a [5, 3]} \
  | time ./pprint :a 1

3

0.01    real
12MB    maximum resident set size
```

	Time	Memory
JVM	1.10 s	100 MB
JS	0.60 s	130 MB
Native	0.01 s	12 MB

```
$ curl -i http://localhost:8080/kv/city -X PUT -d val=Berlin

HTTP/1.1 201 Created
Content-Type: application/octet-stream
Content-Length: 8
Server: http-kit
Date: Mon, 03 Sep 2018 16:04:22 GMT

Berlin

$ curl -i http://localhost:8080/kv/city

HTTP/1.1 200 OK
Content-Type: application/octet-stream
Content-Length: 8
Server: http-kit
Date: Mon, 03 Sep 2018 16:04:27 GMT

Berlin
```

```clojure
(ns webkv.main
  (:require [org.httpkit.server :as http]
            [ring.middleware.defaults
              :refer [wrap-defaults
                      api-defaults]]
            [bidi.ring :refer [make-handler]])
  (:gen-class))

;; We want to be sure none of our calls relies
;; on reflection. Graal does not support them.
(set! *warn-on-reflection* 1)

;; This is where we store our data.
(def ^String tmpdir
  (System/getProperty "java.io.tmpdir"))

;; That's how we find a file given a key.
;; Keys must match the given pattern.
(defn file [^String key]
  {:pre [(re-matches #"^[A-Za-z-]+$" key)]}
  (java.io.File. tmpdir key))

;; Here we handle GET requests. We just
;; read from a file.
(defn get-handler
  [{:keys [params]}]
  {:body (str (slurp (file (params :key))) "\n")})

;; This is our PUT request handler. Given
;; a key and a value we write to a file.
(defn put-handler
  [{:keys [params]}]
  (let [val (params :val)]
    (spit (file (params :key)) val)
    {:body (str val "\n") , :status 201}))

;; Here's the routing tree of our application.
;; We pick the handler depending on the HTTP
;; verb. On top of that we add an extra middle-
;; ware to parse data sent in requests.
(def handler
  (-> ["/kv/" {[:key] {:get #'get-handler
                       :put #'put-handler}}]
      (make-handler)
      (wrap-defaults api-defaults)))

;; Finally, we've got all we need to expose
;; our handler over HTTP.
(defn -main []
  (http/run-server handler
                   {:port 8080})
  (println "🔥 http://localhost:8080"))
```

```
FROM ubuntu AS BASE
RUN apt-get update
RUN apt-get install -yy curl leiningen build-essential zlib1g-dev
RUN cd /opt && curl -sL https://github.com/.../graalvm.tar.gz \
  | tar -xzf -
ADD project.clj .
RUN lein deps
ADD src src
RUN lein uberjar
RUN /opt/graalvm-ce-19.0.0/bin/native-image \
  -H:EnableURLProtocols=http --static --no-server \
  -cp target/webkv-0.0.0-standalone.jar webkv.main
```

```
FROM ubuntu AS BASE
RUN apt-get update
RUN apt-get install -yy curl leiningen build-essential zlib1g-dev
RUN cd /opt && curl -sL https://github.com/.../graalvm.tar.gz \
  | tar -xzf -
ADD project.clj .
RUN lein deps
ADD src src
RUN lein uberjar
RUN /opt/graalvm-ce-19.0.0/bin/native-image \
  -H:EnableURLProtocols=http --static --no-server \
  -cp target/webkv-0.0.0-standalone.jar webkv.main

FROM scratch
COPY --from=BASE /webkv.main /
CMD ["/webkv.main"]
```

How Is That Even Possible

	Time	Memory
JVM	1.10 s	100 MB
JS	0.60 s	130 MB
Native	0.01 s	12 MB

```
static {
    ...
}
```

```
public static void __init0() {
  const__0 = (Var)RT.var("clojure.core", "in-ns");
  const__1 = (AFn)Symbol.intern(null, "hello.core");
  const__2 = (AFn)Symbol.intern(null, "clojure.core");
  const__3 = (Var)RT.var("hello.core", "-main");
  const__4 = (Keyword)RT.keyword(null, "file");
  const__5 = (Keyword)RT.keyword(null, "column");
  const__6 = Integer.valueOf(1);
  const__7 = (Keyword)RT.keyword(null, "line");
  const__8 = Integer.valueOf(3);
  const__9 = (Keyword)RT.keyword(null, "arglists");
  const__10 = PersistentList.create(Arrays.asList(new Object[] {
    RT.vector(new Object[] {
      Symbol.intern(null, "&"),
      Symbol.intern(null, "args")
    })
  }));
```

https://blog.ndk.io/clojure-compilation

Christian Wimmer Follow
VM and compiler researcher at Oracle Labs. Project lead for GraalVM native image generation (Substrate VM). Opinions are my own.
Sep 6 · 9 min read

Understanding Class Initialization in GraalVM Native Image Generation

tl;dr: Classes reachable for a GraalVM native image are initialized at image build time. Objects allocated by class initializers are in the image heap that is part of the executable. The new option `--delay-class-initialization-to-runtime=` *delays initialization of listed classes to image run time.*

Limitations

Substrate VM Java Limitations

Substrate VM does not support all features of Java to keep the implementation small and concise, and also to allow aggressive ahead-of-time optimizations. This page documents the limitations.

What	Support Status
Dynamic Class Loading / Unloading	Not supported
Reflection	Mostly supported
Dynamic Proxy	Mostly supported
Java Native Interface (JNI)	Mostly supported
Unsafe Memory Access	Mostly supported
Static Initializers	Partially supported
InvokeDynamic Bytecode and Method Handles	Not supported
Lambda Expressions	Supported
Synchronized, wait, and notify	Supported
Finalizers	Not supported
References	Mostly supported

Reflection on Substrate VM

Java reflection support (the `java.lang.reflect.*` API) enables Java code to examine its own classes, methods and fields and their properties at runtime.

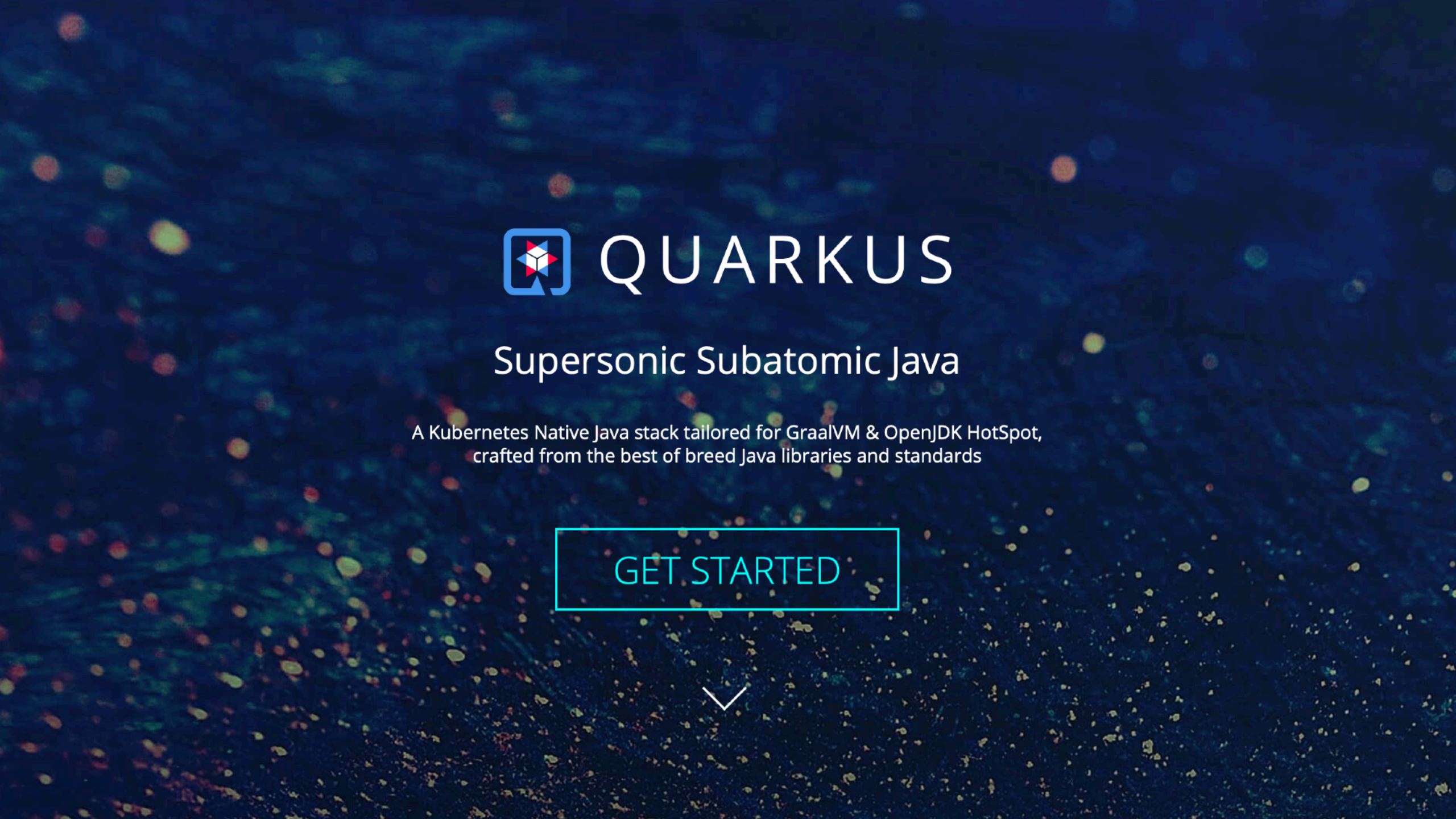
Substrate VM has partial support for reflection and it needs to know ahead of time the reflectively accessed program elements. Examining and accessing program elements through `java.lang.reflect.*` or loading classes with `Class.forName(String)` at run time requires preparing additional metadata for those program elements. (Note: We include here loading classes with `Class.forName(String)` since it is closely related to reflection.)

SubstrateVM tries to resolve the target elements through a static analysis that detects calls to the reflection API. Where the analysis fails the program elements reflectively accessed at run time must be specified using a manual configuration.

Automatic detection

The analysis intercepts calls to `Class.forName(String)`, `Class.forName(String, ClassLoader)`, `Class.getDeclaredField(String)`, `Class.getField(String)`, `Class.getDeclaredMethod(String, Class[])`, `Class.getMethod(String, Class[])`, `Class.getDeclaredConstructor(Class[])` and `Class.getConstructor(Class[])`. If the arguments to these calls can be reduced to a constant we try to resolve the target elements. If the target elements can be resolved the calls are removed and instead the target elements are embedded in the code. If the target elements cannot be resolved, e.g., a class is not on the classpath or it doesn't declare a field/method/constructor, then the calls are replaced with a snippet that throws the appropriate exception at run time. The benefits are two fold. First, at run time there are no calls to the reflection API. Second, Graal can employ constant folding and optimize the code further.

Outlook


QUARKUS
Supersonic Subatomic Java
A Kubernetes Native Java stack tailored for GraalVM & OpenJDK HotSpot,
crafted from the best of breed Java libraries and standards
GET STARTED

MICRONAUT™

A modern, JVM-based, full-stack framework for building modular, easily testable microservice and serverless applications.

GraalVM native image support

Sébastien Deleuze edited this page 5 days ago · 7 revisions

This wiki page is intended to provide an up-to-date status of Spring Framework support for GraalVM native images.

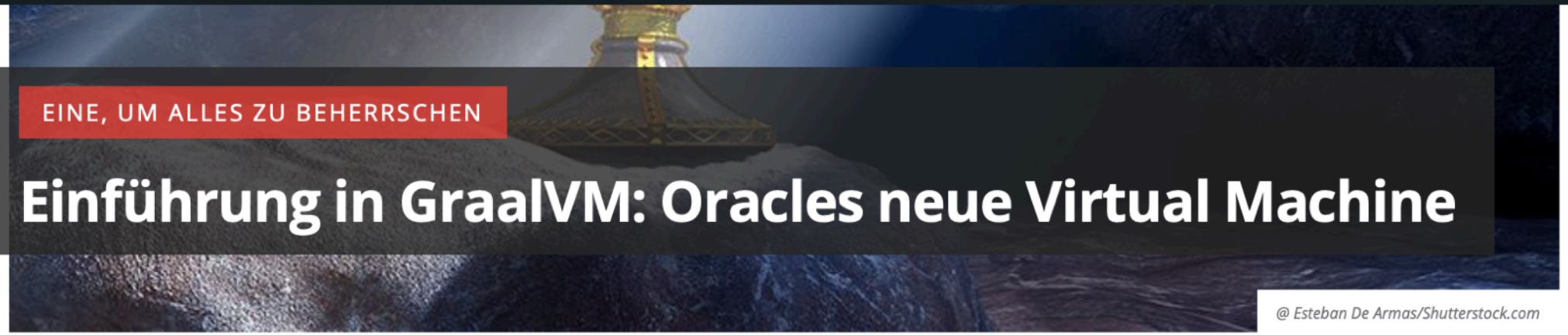
GraalVM

One important thing to have in mind regarding to GraalVM is that it is an umbrella project with a wide scope and multiple components. **While GraalVM is now GA, GraalVM native image feature which allows ahead-of-time compilation of Java applications into executable images is only available as an early adopter plugin, so we don't consider it production ready yet.** 2 versions of GraalVM are provided: the community version which is free and the EE version which is not.

GraalVM native image allows to compile Spring applications to native executable with very fast startup (less than 100ms) with low memory consumption (usually 5x less than its regular JVM equivalent) at the price of lower throughput and various limitations. **Reflection and dynamic proxies are supported but need to be configured manually or via dedicated support.** It also allows to produce small container images.

Support of native images at Spring Framework


GraalVM Native Images (experimental)
Prepared for GraalVM since Spring Framework 5.1
avoiding unnecessary internal reflection
skipping parameter name discovery
As of Spring Framework 5.2: custom setup for GraalVM 19 GA
explicit reflection configuration and command line args necessary
SubstrateVM is still an early adopter plugin in 19 GA
ring Framework 5.3: out-of-the-box setup
stom Graal configuration integration
framework/wiki/GraalVM-native-image-support
SPRING
spring
Sébastien Deleuze @sdeleuze · May 16
More details in @springjuergen talk at #springio19 about @GraalVM native image support in upcoming Spring Framework 5.3, see our new dedicated wiki page.
github.com/spring-project…



EINE, UM ALLES ZU BEHERRSCHEN

Einführung in GraalVM: Oracles neue Virtual Machine

@ Esteban De Armas/Shutterstock.com

Oliver B. Fischer E-Post Development GmbH

„Zusammen mit der Möglichkeit, sie auf Grundlage des Truffle Frameworks als Plattform für Projekte mit den bereits mehrfach genannten Sprachen zu nutzen, selbst domainspezifische Sprachen mit vergleichsweise geringem Aufwand implementieren und polyglotte Anwendungen entwickeln zu können, stellt die GraalVM einen Durchbruch dar, dessen Auswirkung auf die Softwareentwicklung im Moment noch nicht abschätzbar ist.“

23.05.2019 // JUG BB

GraalVM: Fast, Polyglot, Native

Jan Stępień

@janstepien
janstepien.com

INNOQ