19.03.2019

Javaland 2019

Java goes Native

Sascha Selzer

@tommy1199

INNOQ

Sascha Selzer arbeitet seit Juli 2018 als Senior Consultant bei der innoQ Deutschland GmbH.

Er hat langjährige Erfahrung in der Entwicklung mit JVM basierten Sprachen.

Sein aktueller Fokus liegt auf der Konzeption und Umsetzung von Backend Architekturen, sowie von Continuous Delivery/Deployment Strategien.

Sascha Selzer

Senior Consultant bei INNOQ Deutschland GmbH

Why do we use Java?

- Easy to learn
- JVM
- IDE support
- Libraries
- Tools
- Write once, run everywhere

New challenges

- Microservices
- Functions as a service
- Horizontal scaling
- Small memory footprints
- Fast startup times
- Containers are the new “runs everywhere”

Why not switch to another language?

Good reasons to stay...

- Most popular language
- Rich eco system
- Broad Know-How in companies
- Stable and improved over years
- Risk for a lot of organisations to introduce new languages

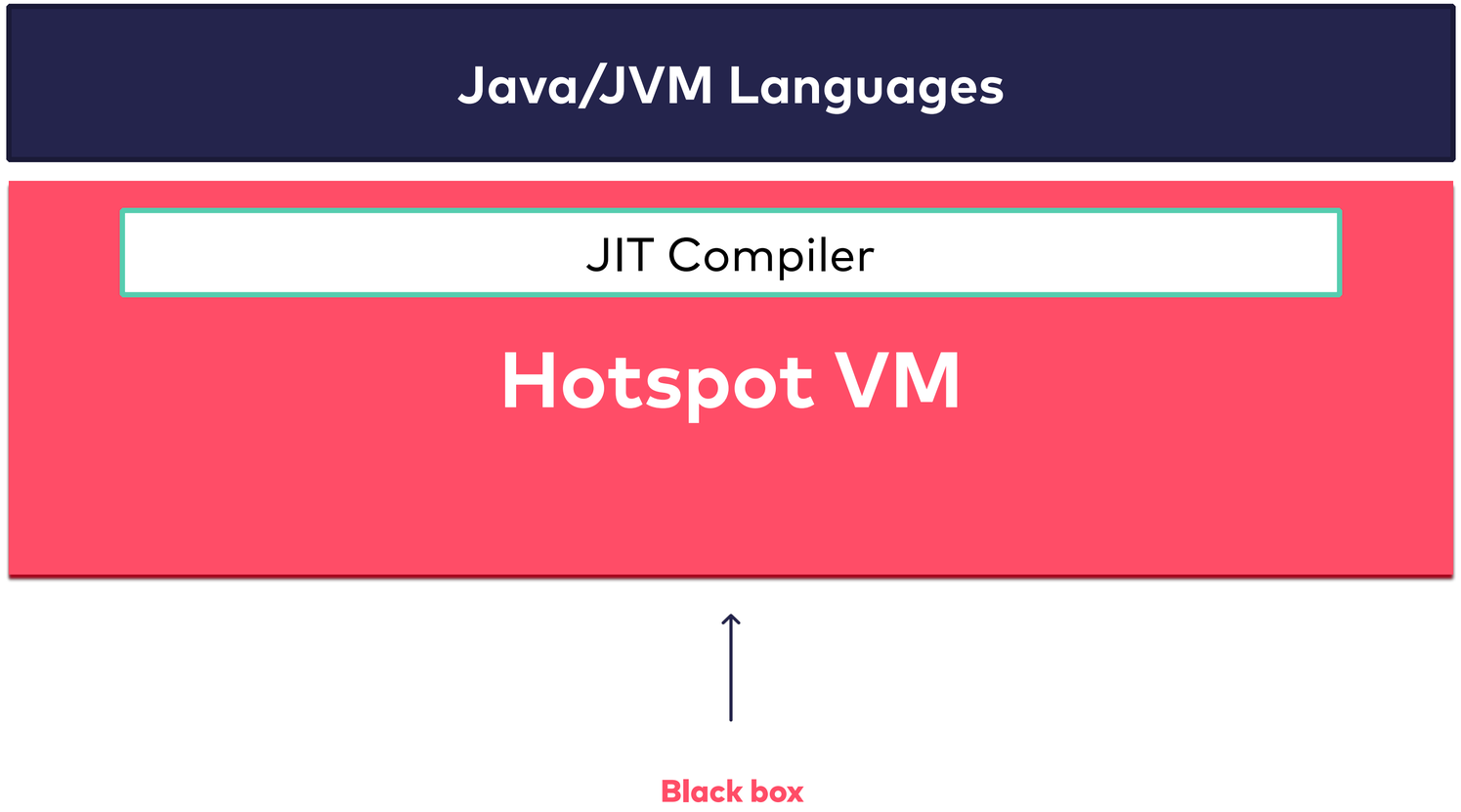
So what else can we do?

Graal

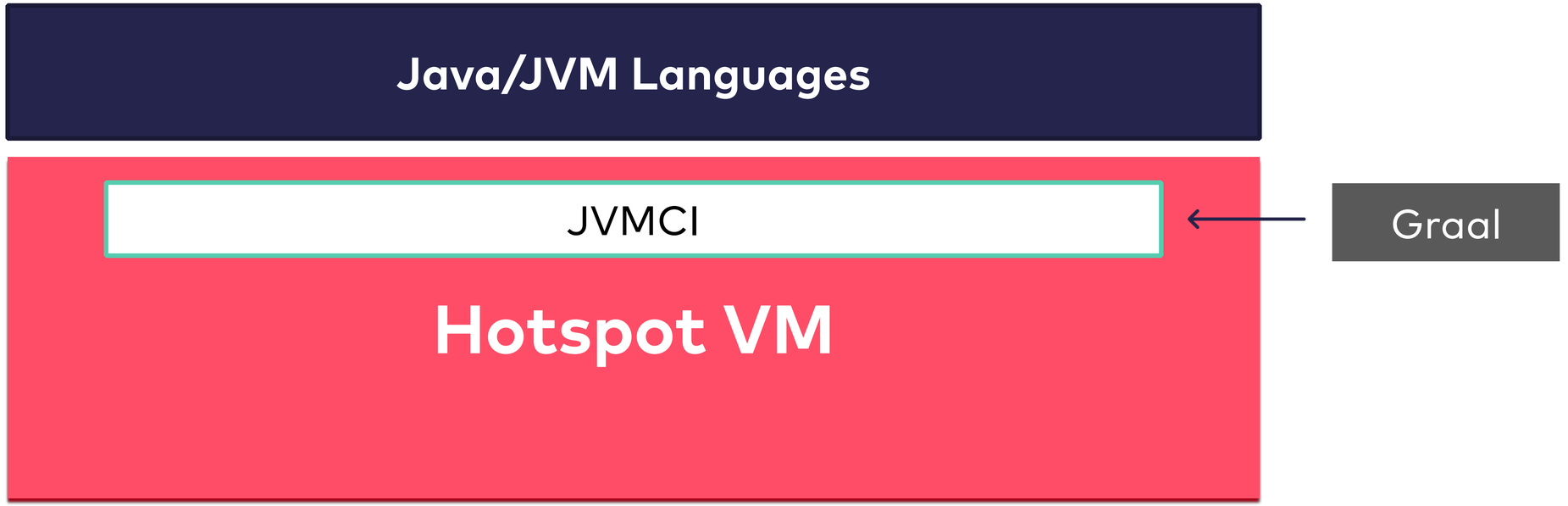
Graal (compiler)

- Modern JIT Compiler
- Implements the JVM compiler interface (JEP 243)
- Written in Java
- Planned to be integrated into official JVM (http://openjdk.java.net/projects/metropolis/)

Before Java 9



Introduced in Java 9



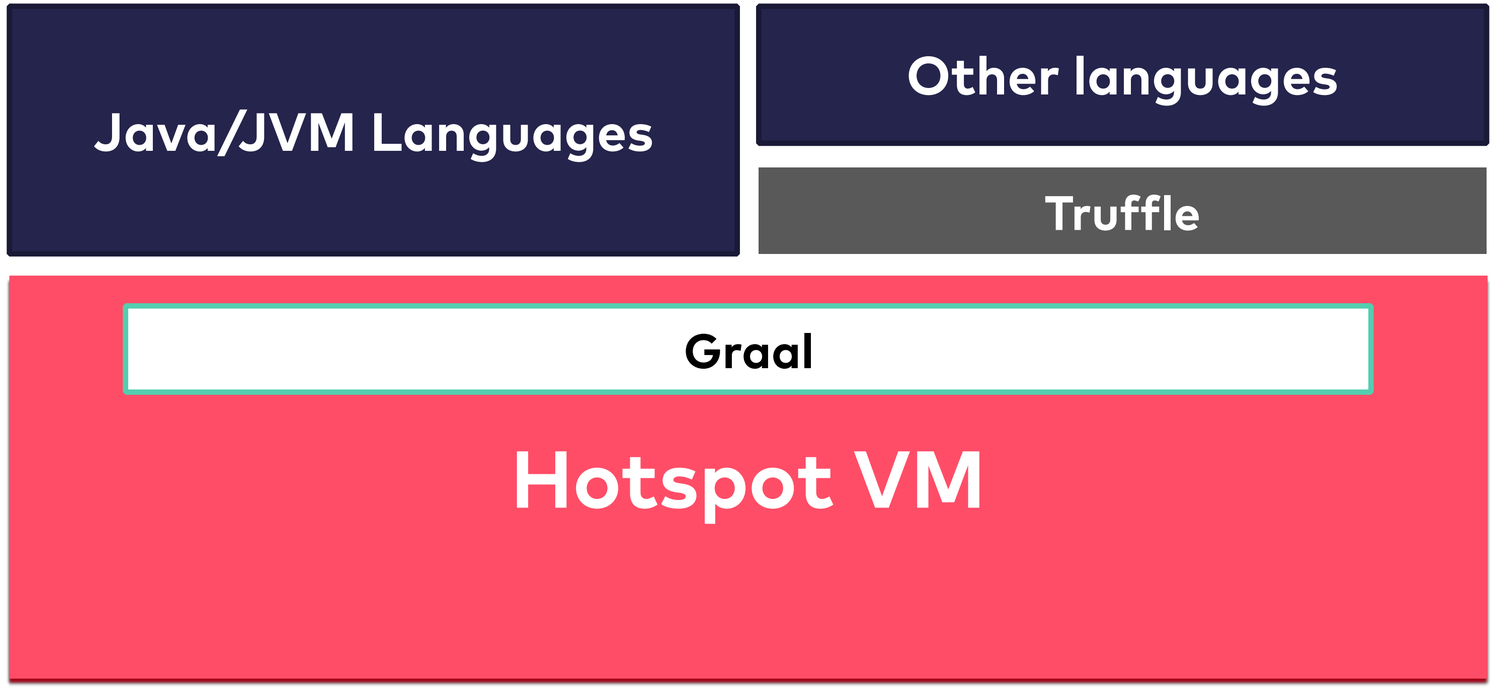
https://chrisseaton.com/truffleruby/jokerconf17/

GraalVM

GraalVM

- Universal virtual machine
- Full fletched JVM
- Contains Graal Compiler as JIT
- Available as CE and EE
- Bundled with additional tools (e.g. native-image)
- Current docker images based on custom Java 8 build

GraalVM Setup



How does native compilation works?

SubstrateVM

“Substrate VM is a framework that allows ahead-of-time (AOT) compilation of Java applications under closed-world assumption into executable images or shared objects (ELF-64 or 64-bit Mach-O).”

SubstrateVM

- Low-footprint VM
- Embeddable
- Written in and for Java
- AOT compiled by using Graal

Native image creation

Static Analysis → **Ahead-of-Time Compilation**

Own Application

JDK classes

Substrate VM

Machine Code

Image Heap

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
Threads	Supported
Identity Hash Code	Supported
Security Manager	Not supported
JVMTI, JMX, other native VM interfaces	Not supported
JCA Security Services	Supported

https://github.com/oracle/graal/blob/master/substratevm/LIMITATIONS.md

Let's try with an example

Most famous TODO app

Spring Boot

Current Status

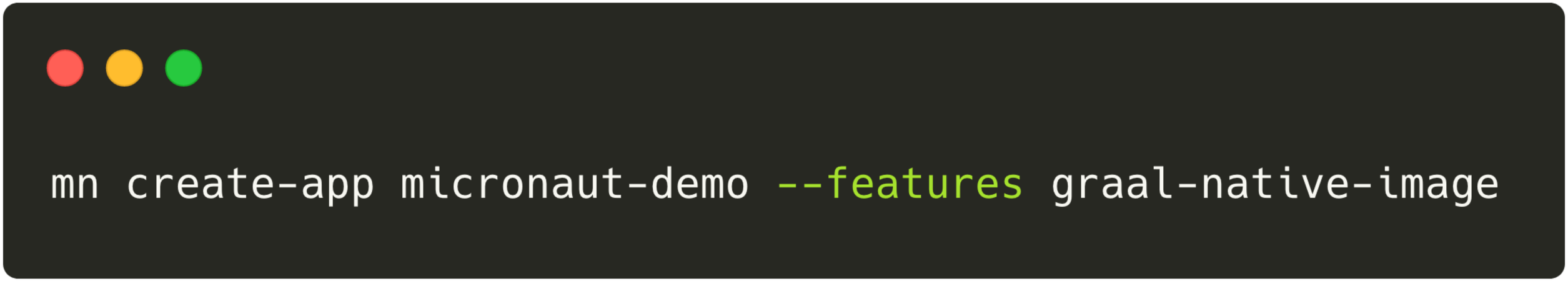
- Initial support finished in Spring Framework 5.1
- Collaborate with GraalVM team
- Spring Fu as incubator for new features (GraalVM support) https://github.com/spring-projects/spring-fu/tree/master/samples/jafu-reactive-minimal

Micronaut

Micronaut

- Annotation based Microservice Framework
- Polyglot
- Fast startup time and small memory footprint through annotation processing
- Graal VM support
- Own CLI

Create a new project




.
├── Dockerfile
├── DockerfileAllInOne
├── build-native-image.sh
├── build.gradle
├── docker-build.sh
├── gradle
│   └── wrapper
│   ├── gradle-wrapper.jar
│   └── gradle-wrapper.properties
├── gradlew
├── gradlew.bat
├── micronaut-cli.yml
└── src
    └── main
        ├── java
        │   └── micronaut
        │   └── demo
        │   ├── Application.java
        │   └── MicronautSubstitutions.java
        └── resources
            ├── application.yml
            └── logback.xml

MicronautSubstitutions.java

```java
@TargetClass(io.netty.util.internal.logging.InternalLoggerFactory.class)
final class Target_io_netty_util_internal_logging_InternalLoggerFactory {
    @Substitute
    private static InternalLoggerFactory newDefaultFactory(String name) {
        return JdkLoggerFactory.INSTANCE;
    }
}
```

```
private static InternalLoggerFactory newDefaultFactory(String name) {
    Object f;
    try {
        f = new Slf4JLoggerFactory(true);
        ((InternalLoggerFactory)f).newInstance(name).debug("Using SLF4J as the default logging framework");
    } catch (Throwable var7) {
        try {
            f = Log4JLoggerFactory.INSTANCE;
            ((InternalLoggerFactory)f).newInstance(name).debug("Using Log4J as the default logging framework");
        } catch (Throwable var6) {
            try {
                f = Log4J2LoggerFactory.INSTANCE;
                ((InternalLoggerFactory)f).newInstance(name).debug("Using Log4J2 ...");
            } catch (Throwable var5) {
                f = JdkLoggerFactory.INSTANCE;
                ((InternalLoggerFactory)f).newInstance(name).debug("Using java.util.logging ...");
            }
        }
    }
    return (InternalLoggerFactory)f;
}
```

MicronautSubstitutions.java

```java
@TargetClass(className = "io.micronaut.caffeine.cache.UnsafeRefArrayAccess")
final class Target_io_micronaut_caffeine_cache_UnsafeRefArrayAccess {
    @Alias @RecomputeFieldValue(kind = Kind.ArrayIndexShift, declClass = Object[].class)
    public static int REF_ELEMENT_SHIFT;
}
```

Usage of Unsafe

- API for low level programming
- Reads and writes data to memory addresses
- Intended to be only used by Java core classes

SubstituteVM limitations

Unsafe Memory Access

Support Status: Mostly supported

What: The memory access methods of `sun.misc.Unsafe`.

Fields that are accessed using `sun.misc.Unsafe` need to be marked as such for the static analysis. In most cases, that happens automatically: field offsets stored in `static final` fields are automatically rewritten from the hosted value (the field offset for the VM that the image generator is running on) to the Substrate VM value, and as part of that rewrite the field is marked as `Unsafe`-accessed. For non-standard patterns, field offsets can be recomputed manually using the annotation `RecomputeFieldValue`.

Supported Common Pattern

```
static final long ... = Unsafe.getUnsafe().objectFieldOffset(X.class.getDeclaredField("x"));

static final long ... = Unsafe.getUnsafe().arrayBaseOffset(byte[].class);

static final long ... = Unsafe.getUnsafe().arrayIndexScale(byte[].class);

...
```

Manually recompute

```
@TargetClass(className = "io.micronaut.caffeine.cache.UnsafeRefArrayAccess")
final class Target_io_micronaut_caffeine_cache_UnsafeRefArrayAccess {
    @Alias @RecomputeFieldValue(kind = Kind.ArrayIndexShift, declClass = Object[].class)
    public static int REF_ELEMENT_SHIFT;
}
```

Dockerfile

```
RUN java -cp micronaut-demo.jar io.micronaut.graal.reflect.GraalClassLoadingAnalyzer
RUN native-image --no-server \
                 --class-path micronaut-demo.jar
                 -H:ReflectionConfigurationFiles=build/reflect.json \
                 -H:EnableURLProtocols=http \
                 -H:IncludeResources='logback.xml|application.yml|META-INF/services/*.*' \
                 -H:+ReportUnsupportedElementsAtRuntime \
                 -H:+AllowVMInspection \
                 --rerun-class-initialization-at-runtime='sun.security.jca.JCAUtil$CachedSecureRandomHolder',... \
                 --delay-class-initialization-to-runtime=io.netty.handler.codec.http.HttpObjectEncoder,... \
                 -H:-UseServiceLoaderFeature \
                 -H:Name=micronaut-demo \
                 -H:Class=micronaut.demo.Application
```

reflect.json

```
[ {
  "name" : "com.fasterxml.jackson.databind.PropertyNamingStrategy$UpperCamelCaseStrategy",
  "allDeclaredConstructors" : true
}, {
  "name" : "com.fasterxml.jackson.datatype.jdk8.Jdk8Module",
  "allDeclaredConstructors" : true
}, {
  "name" : "com.fasterxml.jackson.datatype.jsr310.JSR310Module",
  "allDeclaredConstructors" : true
}, {
  "name" : "io.micronaut.buffer.netty.$NettyByteBufferFactoryDefinitionClass",
  "allDeclaredConstructors" : true
},
...
}]
```

Dockerfile

```
RUN java -cp micronaut-demo.jar io.micronaut.graal.reflect.GraalClassLoadingAnalyzer
RUN native-image --no-server \
                 --class-path micronaut-demo.jar
                 -H:ReflectionConfigurationFiles=build/reflect.json \
                 -H:EnableURLProtocols=http \
                 -H:IncludeResources='logback.xml|application.yml|META-INF/services/*.*' \
                 -H:+ReportUnsupportedElementsAtRuntime \
                 -H:+AllowVMInspection \
                 --rerun-class-initialization-at-runtime='sun.security.jca.JCAUtil$CachedSecureRandomHolder',... \
                 --delay-class-initialization-to-runtime=io.netty.handler.codec.http.HttpObjectEncoder,... \
                 -H:-UseServiceLoaderFeature \
                 -H:Name=micronaut-demo \
                 -H:Class=micronaut.demo.Application
```

Let's run some code

TodoController.java

```java
@Controller("/todos")
public class TodoController {
    ...

    @Post(value = "/{id}", consumes = MediaType.TEXT_PLAIN)
    public HttpResponse update(Integer id, @Body String title) {
        return todoService.update(id, title)
                .map(HttpResponse::ok)
                .orElse(HttpResponse.badRequest());
    }

    @Get("/{id}")
    public HttpResponse todo(Integer id) {
        return todoService.findById(id)
                .map(HttpResponse::ok)
                .orElse(HttpResponse.badRequest());
    }
}
```

Compile with gradle and run

```
./gradlew assemble

java -jar build/libs/micronaut-demo-0.1-all.jar
20:52:42.676 [main] INFO  io.micronaut.runtime.Micronaut - Startup completed in 846ms. Server Running:
http://localhost:8080
```

Compile with gradle and run

```
curl localhost:8080/todos

[{"id":1,"title":"todo 1"},{"id":2,"title":"todo 2"},{"id":3,"title":"todo 3"},{"id":4,"title":"todo 4"}]
```

Build docker image and run

```
docker build -t micronaut-demo .
...
 ---> Running in a26abc58b1cf
[micronaut-demo:7]    classlist:  12,348.47 ms
[micronaut-demo:7]        (cap):   1,087.24 ms
[micronaut-demo:7]        setup:   2,948.21 ms
[micronaut-demo:7]   (typeflow):  35,894.29 ms
[micronaut-demo:7]    (objects):  23,445.56 ms
[micronaut-demo:7]   (features):   1,254.32 ms
[micronaut-demo:7]     analysis:  63,022.78 ms
[micronaut-demo:7]     universe:   1,669.20 ms
[micronaut-demo:7]      (parse):   5,389.90 ms
[micronaut-demo:7]     (inline):   8,082.00 ms
[micronaut-demo:7]    (compile):  39,502.49 ms
[micronaut-demo:7]      compile:  55,234.13 ms
[micronaut-demo:7]        image:   3,761.47 ms
[micronaut-demo:7]        write:     623.53 ms
[micronaut-demo:7]      [total]: 139,990.63 ms
...
docker run -it --rm -p 8080:8080 micronaut-demo
20:04:32.654 [main] INFO  io.micronaut.runtime.Micronaut - Startup completed in 28ms. Server Running:
http://e76ab27cfd0b:8080
```

Build docker image and run

```
curl localhost:8080/todos
{"message":"Internal Server Error: Error encoding object [[micronaut.demo.model.Todo@7f1f8cca5368,
micronaut.demo.model.Todo@7f1f8cca5400, micronaut.demo.model.Todo@7f1f8cca5410,
micronaut.demo.model.Todo@7f1f8cca5420]] to JSON: No serializer found for class micronaut.demo.model.Todo and no
properties discovered to create BeanSerializer (to avoid exception, disable
SerializationFeature.FAIL_ON_EMPTY_BEANS) (through reference chain: java.util.ArrayList[0])"}
```

TodoController.java

```java
@Controller("/todos")
public class TodoController {
    ...

    @Post(value = "/{id}", consumes = MediaType.TEXT_PLAIN)
    public HttpResponse update(Integer id, @Body String title) {
        return todoService.update(id, title)
                .map(HttpResponse::ok)
                .orElse(HttpResponse.badRequest());
    }

    @Get("/{id}")
    public HttpResponse todo(Integer id) {
        return todoService.findById(id)
                .map(HttpResponse::ok)
                .orElse(HttpResponse.badRequest());
    }
}
```

reflect.json

```
[{
...
}, {
  "name": "micronaut.demo.model.Todo",
  "allDeclaredConstructors" : true,
  "allDeclaredMethods" : true
} ]
```

Rerun

```
curl localhost:8080/todos

[{"id":1,"title":"todo 1"},{"id":2,"title":"todo 2"},{"id":3,"title":"todo 3"},{"id":4,"title":"todo 4"}]
```

Improvements

```
FROM oracle/graalvm-ce:1.0.0-rc13 as builder
COPY build/libs/*-all.jar micronaut-demo.jar
ADD . build
RUN native-image --no-server \
             --class-path micronaut-demo.jar \
             -H:ReflectionConfigurationFiles=build/reflect.json \
             -H:DynamicProxyConfigurationFiles=build/proxy.json \
             -H:IncludeResources="logback.xml|application.yml|META-INF/services/*.*" \
             -H:Name=micronaut-demo \
             -H:Class=micronaut.demo.Application \
             -H:+ReportUnsupportedElementsAtRuntime \
             -H:+AllowVMInspection \
             -H:+ReportExceptionStackTraces \
             -H:-UseServiceLoaderFeature \
             --enable-http \
             --allow-incomplete-classpath \
             --static \
             --rerun-class-initialization-at-runtime='sun.security.jca.JCAUtil$CachedSecureRandomHolder',... \
             --delay-class-initialization-to-runtime=io.netty.handler.codec.http.HttpObjectEncoder,...

FROM scratch

EXPOSE 8080

COPY --from=builder /micronaut-demo .

ENTRYPOINT ["./micronaut-demo"]
```

Or use latest mn (1.1.0.M2)

```
FROM oracle/graalvm-ce:1.0.0-rc11 as graalvm
COPY . /home/app/micronaut-demo
WORKDIR /home/app/micronaut-demo
RUN ./build-native-image.sh

FROM frolvlad/alpine-glibc
EXPOSE 8080
COPY --from=graalvm /home/app/micronaut-demo .
ENTRYPOINT ["./micronaut-demo"]
```

Persistence

Candidates

- ~~JAsync~~
- ~~Reactive Postgres~~
- ~~Hibernate~~
- JDBC + jOOQ

build.gradle

```
dependencies {
    compile "io.micronaut.configuration:micronaut-jdbc-tomcat"
    compile "org.jooq:jooq:3.11.10"
    runtime "org.postgresql:postgresql:42.2.5"
}
```

reflect.json

```
[{
...
, {
  "name" : "org.postgresql.Driver",
  "allPublicMethods" : true,
  "allDeclaredConstructors" : true
}]
```

application.yml

```yaml
datasources:
    default:
        url: jdbc:postgresql://${DB_HOST:localhost}:5432/${DB_NAME:postgres}
        username: postgres
        password: ""
        driverClassName: org.postgresql.Driver
```

Configure jOOQ

```java
@Factory
public class DSLFactory {

    @Bean
    @Singleton
    public DSLContext dslContext(DataSource dataSource) {
        return DSL.using(dataSource, SQLDialect.POSTGRES);
    }
}
```

jOOQ Usage

```
public List<Todo> findAll() {
    return context.select(ID_COLUMN, CONTENT_COLUMN)
                  .from(TODOS_TABLE)
                  .orderBy(ID_COLUMN)
                  .fetch()
                  .map(this::toTodo);
}
```

Docker build

```
Error: com.oracle.graal.pointsto.constraints.UnsupportedFeatureException: Invoke with MethodHandle argument could
not be reduced to at most a single call: java.lang.invoke.MethodHandle.bindTo(Object)
Trace:
    at parsing java.lang.invoke.MethodHandleImpl.makePairwiseConvertByEditor(MethodHandleImpl.java:221)
Call path from entry point to java.lang.invoke.MethodHandleImpl.makePairwiseConvertByEditor(MethodHandle,
MethodType, boolean, boolean):
    at java.lang.invoke.MethodHandleImpl.makePairwiseConvertByEditor(MethodHandleImpl.java:207)
    at java.lang.invoke.MethodHandleImpl.makePairwiseConvert(MethodHandleImpl.java:194)
    at java.lang.invoke.MethodHandleImpl.makePairwiseConvert(MethodHandleImpl.java:380)
    at java.lang.invoke.MethodHandle.asTypeUncached(MethodHandle.java:776)
    at java.lang.invoke.MethodHandle.asType(MethodHandle.java:761)
    at java.lang.invoke.MethodHandle.invokeWithArguments(MethodHandle.java:627)
    at org.jooq.impl.DefaultRecordMapper$ProxyMapper$2.invoke(DefaultRecordMapper.java:601)
    ...
```

Method Handles

InvokeDynamic Bytecode and Method Handles

Support Status: Not supported

What: The `invokedynamic` bytecode and the method handle classes introduced with Java 7.

The static analysis and our closed-world assumption require that we know all methods that are called and their call sites, while `invokedynamic` can introduce calls at runtime or change the method that is invoked.

Only special use cases of `invokedynamic` are supported: when the `invokedynamic` can be reduced to a single virtual call or field access during native image generation. This is sufficient for full support of Java 8 Lambda expressions.

build.gradle

```
dependencies {
    compile "io.micronaut.configuration:micronaut-jdbc-tomcat"
    compile "org.jooq:jooq:3.7.4"
    runtime "org.postgresql:postgresql:42.2.5"
}
```

Docker run

```
com.oracle.svm.core.jdk.UnsupportedFeatureError: Proxy class defined by interfaces [interface
java.sql.PreparedStatement] not found. Generating proxy classes at runtime is not supported. Proxy classes need to
be defined at image build time by specifying the list of interfaces that they implement. To define proxy classes
use -H:DynamicProxyConfigurationFiles=...
```

Dynamic Proxy

Dynamic Proxy

Support Status: Mostly supported

What: Generating dynamic proxy classes and allocating instances of dynamic proxy classes using the `java.lang.reflect.Proxy` API.

Dynamic class proxies are supported as long as the bytecodes are generated ahead-of-time. This means that the list of interfaces that define dynamic proxies needs to be known at image build time. SubstrateVM employs a simple static analysis that intercepts calls to `java.lang.reflect.Proxy.newProxyInstance(ClassLoader, Class<?>[], InvocationHandler)` and `java.lang.reflect.Proxy.getProxyClass(ClassLoader, Class<?>[])` and tries to determine the list of interfaces automatically. Where the analysis fails the lists of interfaces can be specified via configuration files. For more details, read our documentation on dynamic proxies.

proxy.json

```
[
  ["java.sql.PreparedStatement"]
]
```

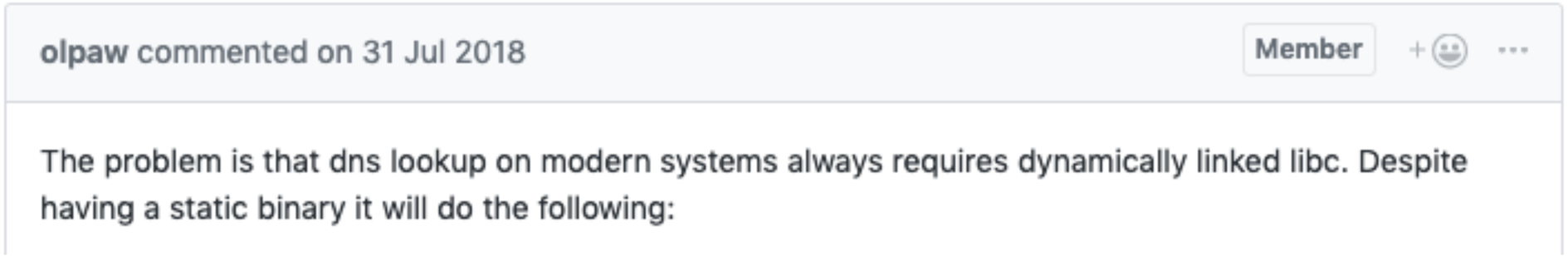
Dockerfile

```
RUN native-image --no-server \
             --class-path micronaut-demo.jar \
             -H:ReflectionConfigurationFiles=build/reflect.json \
             -H:DynamicProxyConfigurationFiles=build/proxy.json \
             -H:IncludeResources="logback.xml|application.yml|META-INF/services/*.*" \
             ...
```

Unknown Host

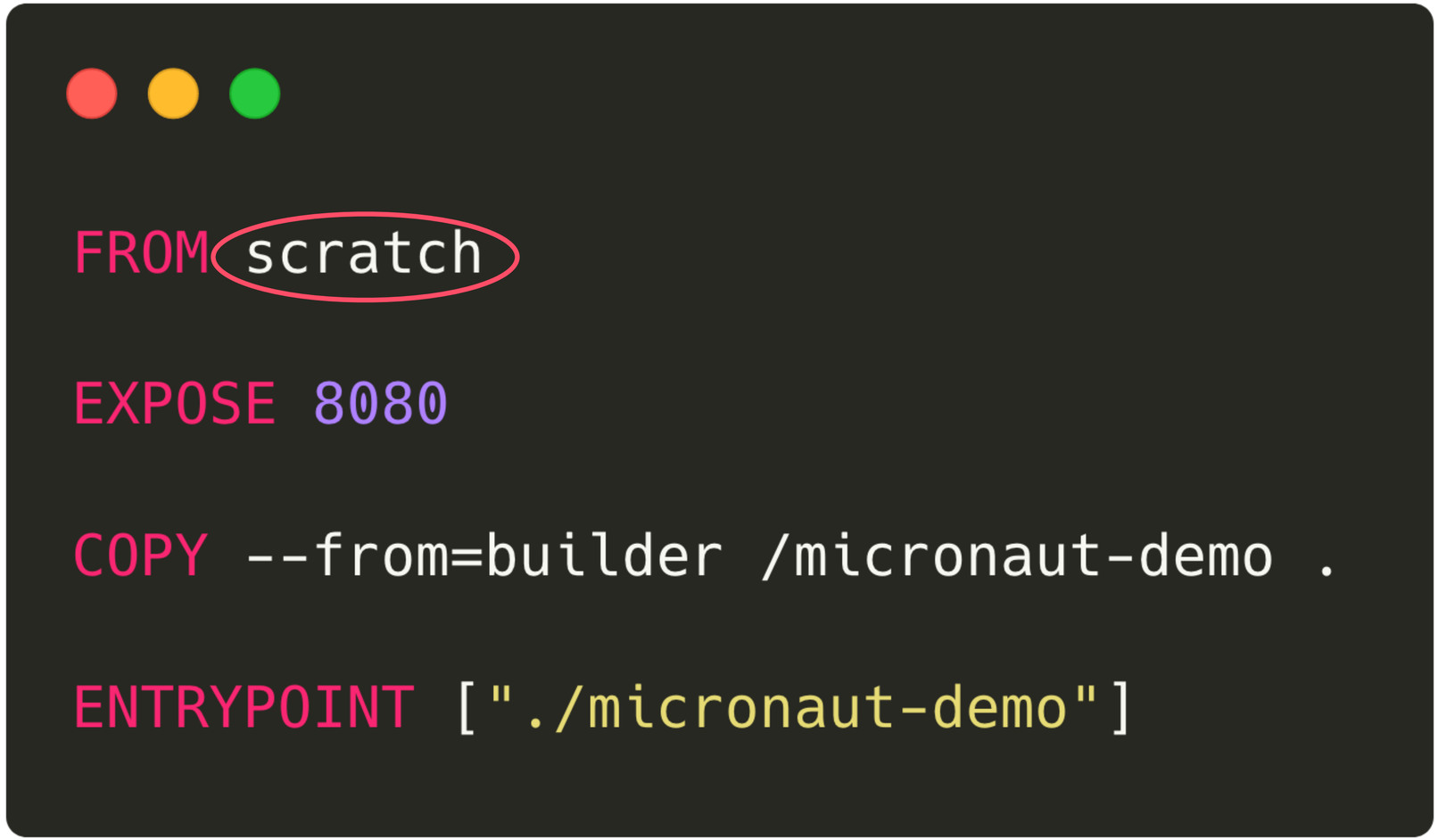
```
Caused by: java.net.UnknownHostException: db
       at java.net.AbstractPlainSocketImpl.connect(AbstractPlainSocketImpl.java:184)
       at java.net.SocksSocketImpl.connect(SocksSocketImpl.java:392)
       at java.net.Socket.connect(Socket.java:589)
       at org.postgresql.core.PGStream.<init>(PGStream.java:70)
       at org.postgresql.core.v3.ConnectionFactoryImpl.tryConnect(ConnectionFactoryImpl.java:91)
       at org.postgresql.core.v3.ConnectionFactoryImpl.openConnectionImpl(ConnectionFactoryImpl.java:192)
       ... 47 common frames omitted
```

DNS



https://github.com/oracle/graal/issues/571

Dockerfile


FROM scratch
EXPOSE 8080
COPY --from=builder /micronaut-demo .
ENTRYPOINT ["./micronaut-demo"]

Dockerfile

```
FROM oraclelinux:7-slim AS base

FROM scratch

EXPOSE 8080

COPY --from=base /lib64/ld-linux-x86-64.so.2 /lib64/ld-linux-x86-64.so.2
COPY --from=base /lib64/libc.so.6 /lib64/libc.so.6
COPY --from=base /lib64/libcrypt.so.1 /lib64/libcrypt.so.1
COPY --from=base /lib64/libdl.so.2 /lib64/libdl.so.2
COPY --from=base /lib64/libfreebl3.so /lib64/libfreebl3.so
COPY --from=base /lib64/libnss_dns.so.2 /lib64/libnss_dns.so.2
COPY --from=base /lib64/libnss_files.so.2 /lib64/libnss_files.so.2
COPY --from=base /lib64/libpthread.so.0 /lib64/libpthread.so.0
COPY --from=base /lib64/libresolv.so.2 /lib64/libresolv.so.2
COPY --from=base /lib64/librt.so.1 /lib64/librt.so.1
COPY --from=base /lib64/libz.so.1 /lib64/libz.so.1

COPY --from=builder /micronaut-demo .

ENTRYPOINT ["./micronaut-demo"]
```

Run

```
curl localhost:8080/todos

[{"id":1,"title":"todo 1"},{"id":2,"title":"todo 2"},{"id":3,"title":"todo 3"},{"id":4,"title":"todo 4"}]
```

Result

- Startup 28ms (~30 times faster)
- Image size ~48MB (~9 times smaller)
- Memory usage ~15MB (~7 times smaller)

What else?

Kotlin/Native

- LLVM backend for the Kotlin compiler
- Creates native images
- Native Implementation of the Kotlin standard library
- Early stage
- Rudimentary support in Ktor

Additional resources

- https://github.com/tommy1199/native-jvm-showcase
- https://royvanrijn.com/blog/2018/09/part-1-java-to-native-using-graalvm/
- https://www.youtube.com/watch?v=9oHpAhgkNAY&feature=youtu.be
- https://chrisseaton.com/truffleruby/jokerconf17/

19.03.2019

Javaland 2019

Java goes Native

Sascha Selzer

@tommy1199

INNOQ