Microservices with Micronaut

2021-01-28 - Moritz Kammerer

QAWARE
SOFTWARE ENGINEERING

About me

- 2013 M.Sc. Computer Science @ Munich University of Applied Sciences
- 2013 Software Engineer @ QAware
- …
- 2021 Expert Software Engineer @ QAware

Working on backend stuff, mostly Java.

LinkedIn | GitHub

Why Cloud?

What’s the problem?

```
: Started UserManagementApplication in 13.307 seconds (JVM running for 13.856)
: Git commit hash a3160f2d39df425a7f095fe4b07d412c6d2836e3, local changes: true
```

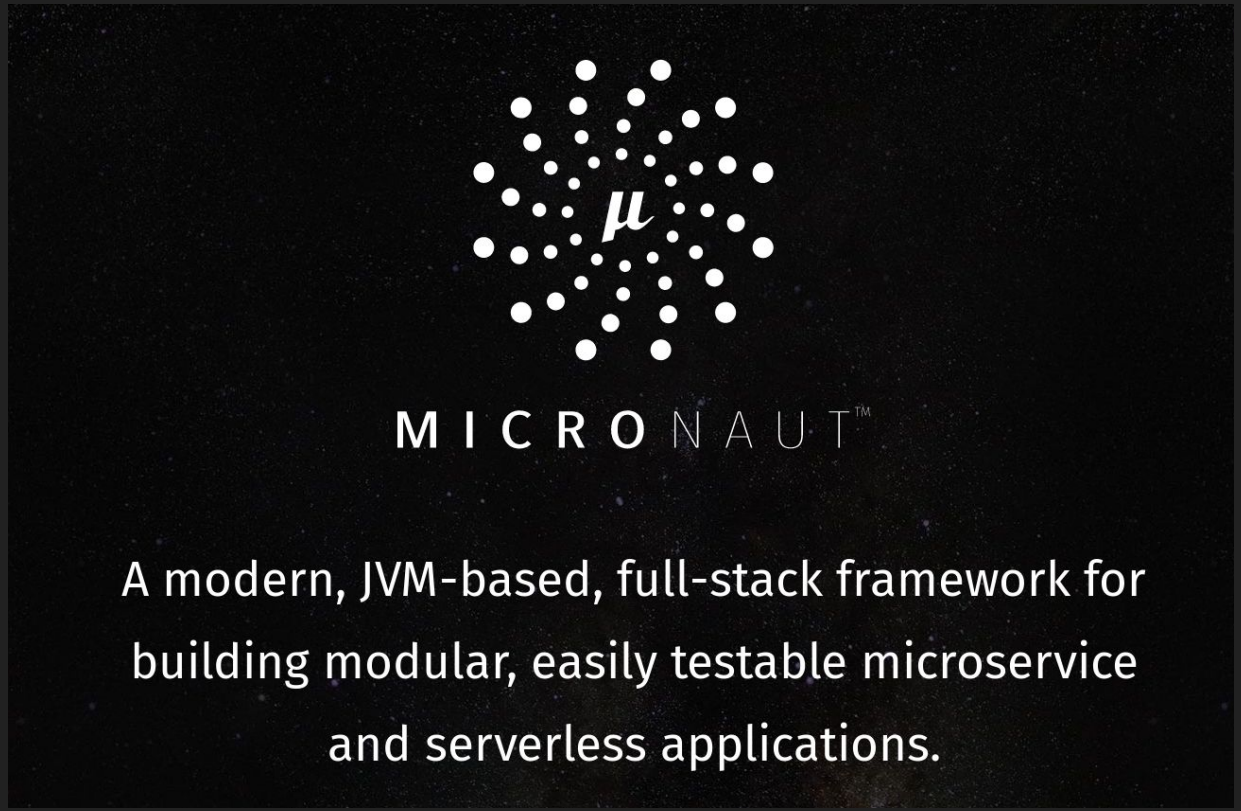
```
Tomcat started on port(s): 3122 (http) with context path '/cvi'
Started CviCoreApplication in 27.24 seconds (JVM running for 28.307)
```

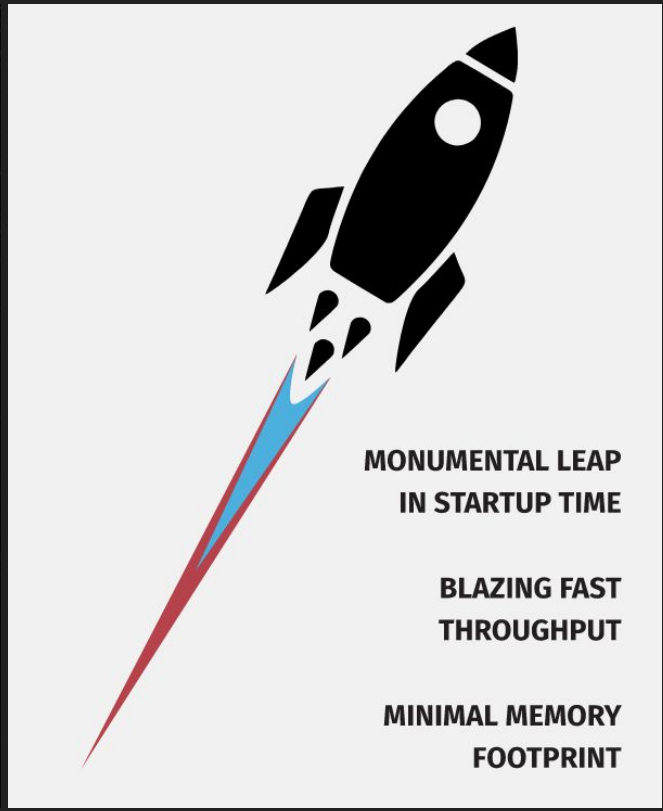
Why does it take so long?

- Start time of Spring (Boot)
  - Depends on the number of beans in the context
  - Reading of bytecode [3]
  - Creating proxies at runtime
  - Runtime AOP
  - Reflection
  - Annotation Synthesizing [1] [2]
- Memory usage
  - Reflective Metadata Cache
- Debugging
  - Very very long stacktraces
  - … with code in dynamically generated proxy classes

[1] https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/core/annotation/AnnotationUtils.html
[2] https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/core/annotation/AliasFor.html
[3] https://github.com/spring-projects/spring-framework/blob/master/spring-core/src/main/java/org/springframework/core/type/classreading/AbstractRecursiveAnnotationVisitor.java

Micronaut to the rescue!





What’s the (promised) solution?

- Create stuff at compile time
  - Proxies
  - Dependency Injection
  - AOP
- Why?
  - Compiled once, run often
  - Tradeoff: Compile time vs. Runtime
- Goal:
  - No reflection - or as little as possible
  - No proxies
  - Minimize startup time
  - Minimize memory usage
  - Readable stacktraces

And how does that work?

- Annotation Processor
  - Reads bytecode & annotations of the classes
  - Creates new classes
  - Doesn’t touch your code, just extends it
- Gradle
  - `annotationProcessor "io.micronaut:micronaut-inject-java"`
  - `annotationProcessor "io.micronaut:micronaut-validation"`
- Also runs with Maven
  - < Insert 500 KB of XML here >

$CacheWarmerDefinition.class
$CacheWarmerDefinition$$exec1.class
$CacheWarmerDefinition$$exec1$$AnnotationMetadata.class
$CacheWarmerDefinitionClass.class
$CacheWarmerDefinitionClass$$AnnotationMetadata.class
$ProjectsControllerDefinition.class
$ProjectsControllerDefinition$$exec1.class
$ProjectsControllerDefinition$$exec1$$AnnotationMetadata.class
$ProjectsControllerDefinitionClass.class
$ProjectsControllerDefinitionClass$$AnnotationMetadata.class
Application.class
CacheWarmer.class
MicronautSubstitutions.class
ProjectDto.class
ProjectsController.class

And this is really working?

- Spring Boot 2.1.2 (starter-web), Java 11, @Component on empty class
  - 0 Beans: 1.236 seconds
  - 1000 Beans: 1.808 seconds
  - 2000 Beans: 2.356 seconds
  - 4000 Beans: 3.259 seconds
  - 8000 Beans: 6.498 seconds
  - 16000 Beans: 9.231 seconds
- Micronaut 1.0.4, Java 11, @Singleton on empty class
  - 0 Beans: 1085ms
  - 1000 Beans: 1870ms
  - 2000 Beans: 2757ms
  - 4000 Beans: 4529ms
  - 8000 Beans: 8038ms
  - 16000 Beans: 15861ms


Hmmmm...

Debunked: It’s not the classpath scanning

- Not so expensive:
  - Classpath scanning (Spring)
  - Injection with reflection (Spring)
- Expensive
  - Class Loading (Micronaut + Spring)
  - Runtime AOP Proxies (Spring)
  - Annotation Synthesizing (Spring)
  - Read bytecode on big classes (Spring)
  - Auto-Configuration Magic (Spring)
- Most of the expensive stuff is done at Micronaut compile time

Looks great, so not disadvantages?

The price you pay: Compile time (16k Beans)

```
Spring:
[INFO] Total time: 7.958 s

Micronaut:
BUILD SUCCESSFUL in 5m 3s
```

Benchmarking is hard!

- This was **NOT** a real benchmark
- Most time spent in the applications shown on the first slides: Spring autoconfiguration magic
- It really depends on the needed feature of your application
  - Actuator
  - JPA / Hibernate
  - Tracing
  - Metriken
  - @FeignClient
  - @Transactional
  - @Cacheable
  - …
- More in the live demo!

Okay... what else does Micronaut bring to the table?

- Dependency Injection
  - @Inject, @Named, @Singleton, @Prototype, ...
- REST Controller
  - @Controller, @GET, @POST, …
- Events
  - @PostConstruct
  - @EventListener
- AOP
  - Around Advice, Introduction Advice, …
- Validation
  - @Validated, @NotNull, ...

...

- Caching
  - @Cachable, @CacheInvalidate, …
- Retry
  - @Retryable
- Circuit Breaking
  - @CircuitBreaker
- Fallback / Recovery
  - @Recoverable / @Fallback
- Scheduling
  - @Scheduled

...

- Reactive & Blocking HTTP with Netty
  - Compile Time Route Validation
  - @Error Exception Handling ala Spring
- Websocket Server & Clients
- Server side views
  - Thymeleaf, Handlebars, Velocity
- OpenAPI / Swagger
  - Open API spec is created on compile time!
- HTTP Clients
  - @Client
  - Client-side load balancing
  - Service discovery (Consul, Eureka, K8S, Route 53, DNS based)
  - OpenTracing (Zipkin, Jaeger)

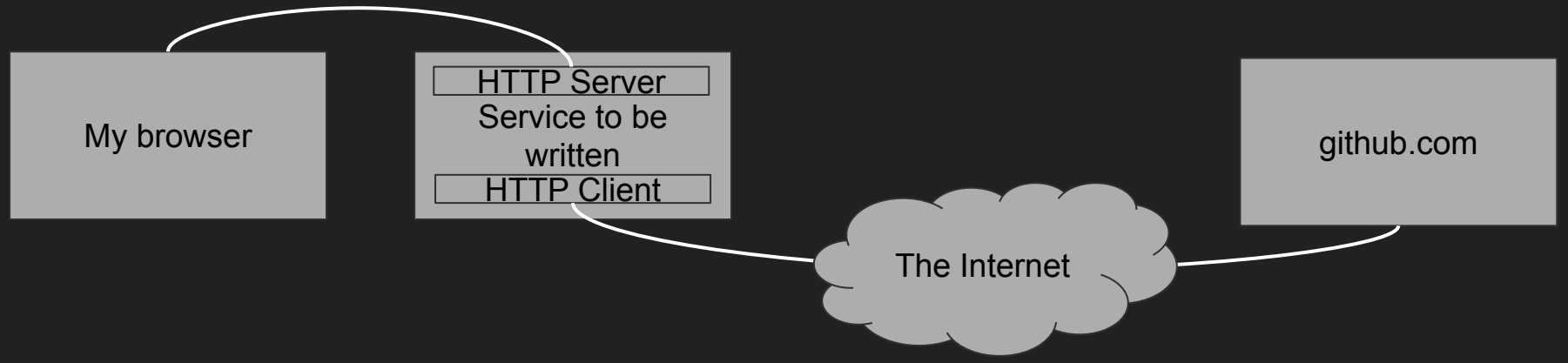
...

- Application Configuration
  - application.yml / .json / .groovy
  - Different Environments
  - Property Sources (ENV, System Properties, …)
  - @ConfigurationProperties, @Value
  - Distributed config with Consul, AWS Parameter Store
- Database support
  - JPA, Mongo, Neo4J, Postgres-Reactive, Redis, Cassandra
- Monitoring
  - @Endpoint (/beans, /info, /health, …)
  - Metriken with Micrometer
  - Change log level at runtime

...

- Security
  - Like Spring Security with users and roles (RBAC)
  - Basic Auth / Session based
  - JWT (JWS and JWE) incl. automatic token propagation
  - Auth with LDAP
- Function-as-a-Service (AWS Lambda, OpenFaaS)
- Message driven services (RabbitMQ / Kafka)
- CLI Applikationen
- Good documentation!
  - e.g. with tutorials for Let’s Encrypt, starting Consul, etc.

Enough bla bla, show code!


My browser
HTTP Server
Service to be written
HTTP Client
The Internet
github.com

Benchmarks

Spring vs. Micronaut - GitHub Scraper

Micronaut 2.2.1

BUILD SUCCESSFUL in 3s

0,89s user 0,07s system 22% cpu 4,199 total

Startup completed in 625ms

Spring vs. Micronaut - GitHub Scraper

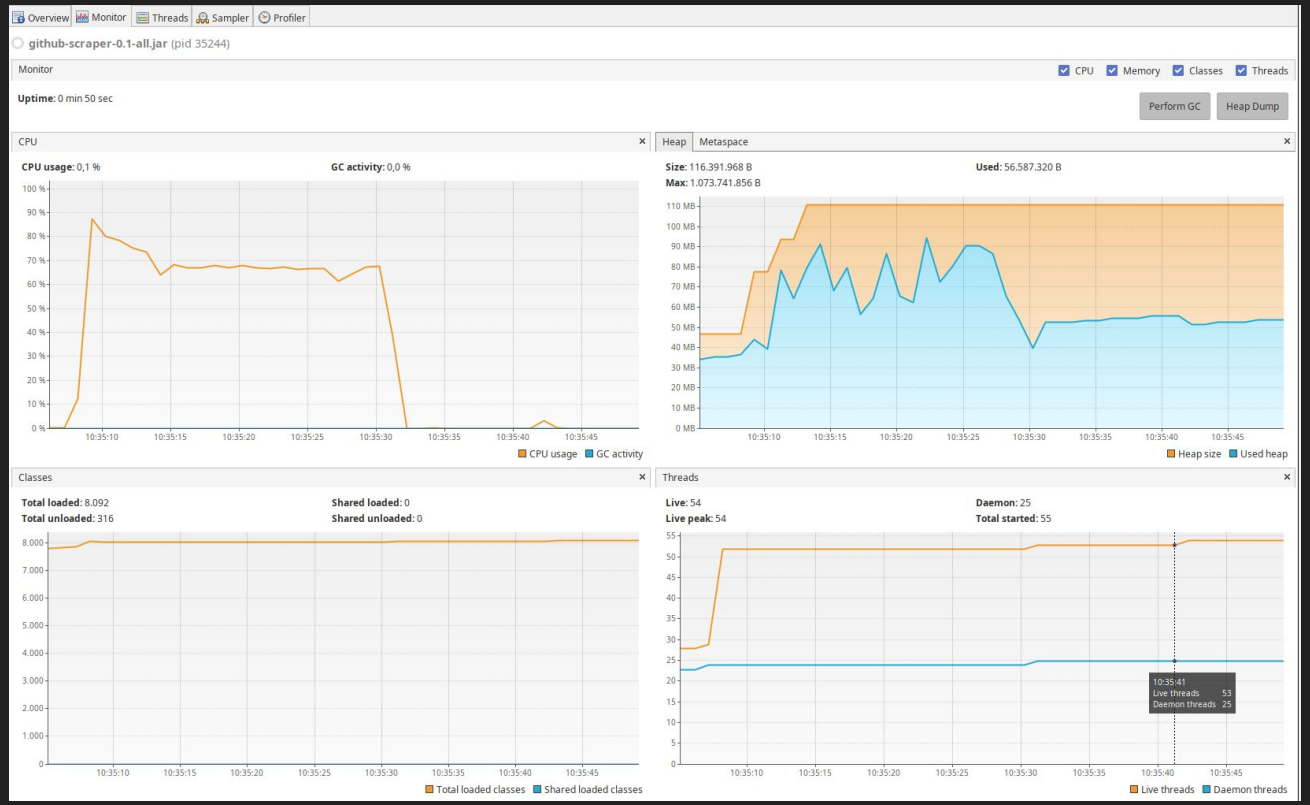
Spring 2.4.1

BUILD SUCCESSFUL in 1s

0,86s user 0,09s system 67% cpu 1,414 total

Started DemoApplication in 1.21 seconds (JVM running for 1.556)

Micronaut (JAR: 14.8 MB)

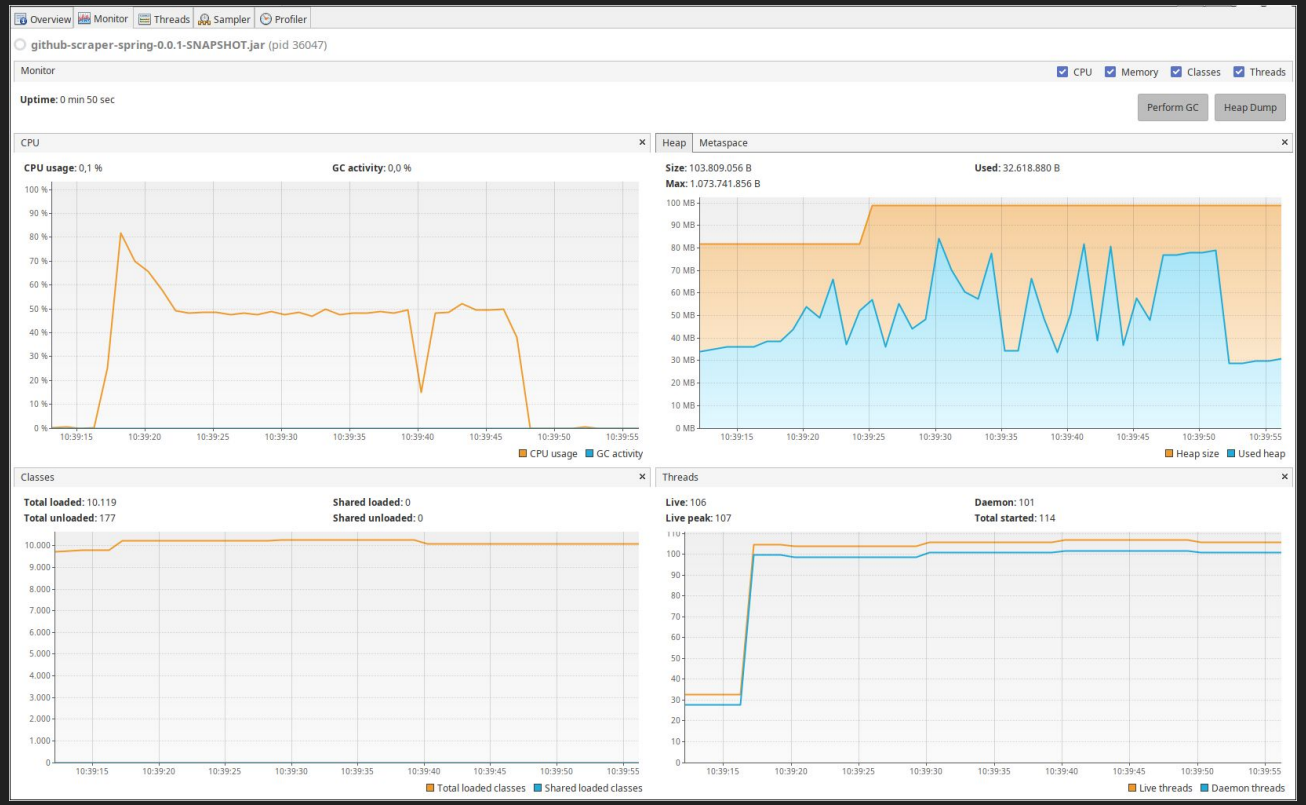


java -Xms16M -Xmx512M -jar github-scraper-0.1-all.jar / OpenJDK Runtime Environment AdoptOpenJDK (build 11.0.9+11)

Micronaut

```
wrk -t 8 -c 80 -d 30s --latency http://localhost:8080/github/stars
Running 30s test @ http://localhost:8080/github/stars
  8 threads and 80 connections
  Thread Stats   Avg      Stdev    Max   +/- Stdev
    Latency  2.21ms   5.33ms 139.33ms   91.82%
    Req/Sec  15.22k   5.34k   27.39k   72.74%
  Latency Distribution
    50%  286.00us
    75% 1.98ms
    90% 6.44ms
    99%   22.34ms
  2765210 requests in 30.07s, 1.38GB read
Requests/sec:  91956.68
Transfer/sec:    47.01MB
```

Spring Boot (JAR: 30.6 MB)



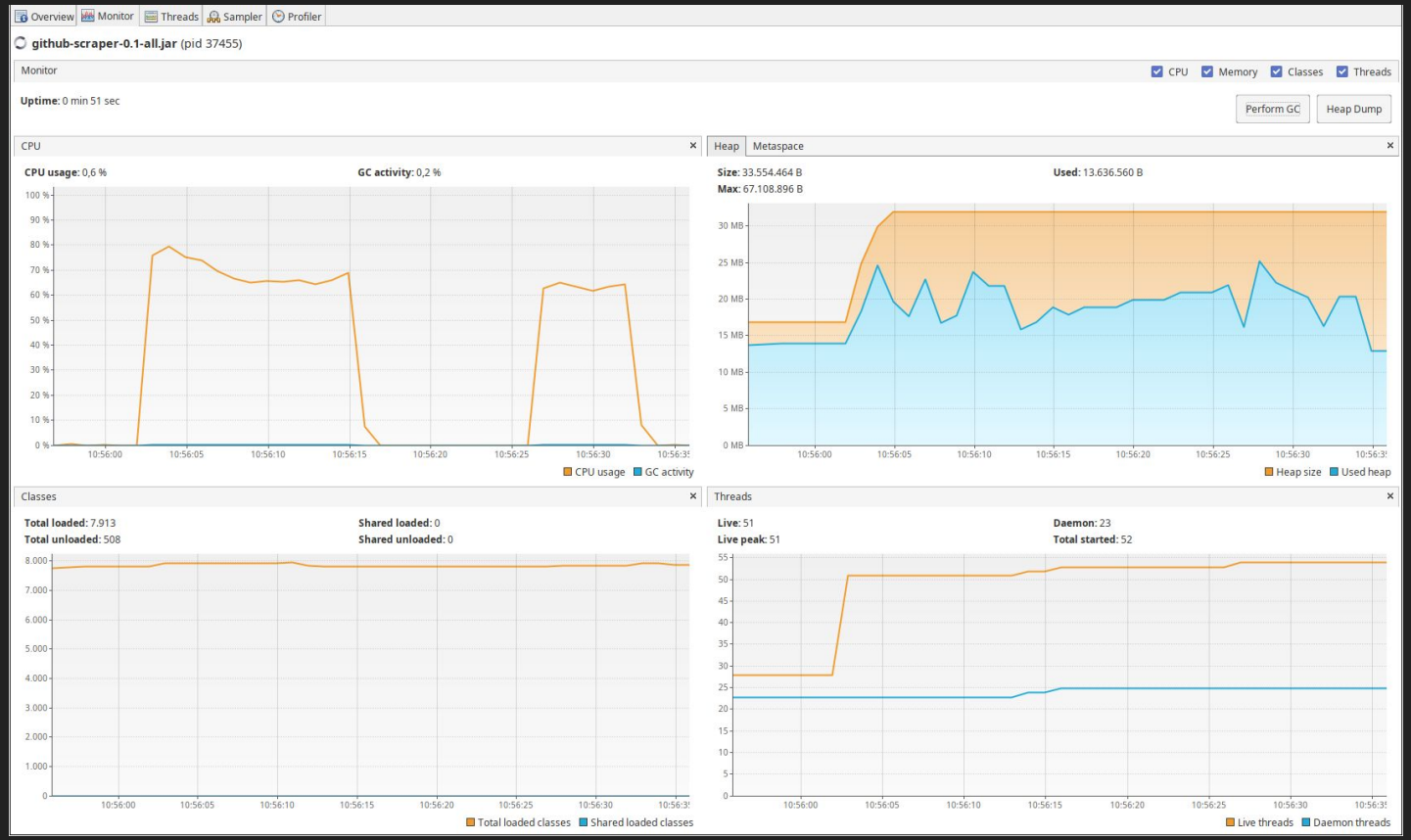
java -Xms16M -Xmx512M -jar github-scraper-spring-0.0.1-SNAPSHOT.jar / OpenJDK Runtime Environment AdoptOpenJDK (build 11.0.9+11)

Spring Boot

```
wrk -t 8 -c 80 -d 30s --latency http://localhost:8080/github/stars
Running 30s test @ http://localhost:8080/github/stars
  8 threads and 80 connections
  Thread Stats   Avg      Stdev    Max   +/- Stdev
    Latency  10.23ms   65.15ms 778.07ms   97.82%
    Req/Sec  9.29k    2.69k   14.24k   78.93%
  Latency Distribution
    50%  0.87ms
    75%  1.55ms
    90%  3.36ms
    99%  414.50ms
  2169903 requests in 30.10s, 1.07GB read
Requests/sec:  72091.42
Transfer/sec:    36.38MB
```

How much RAM do I have to use?

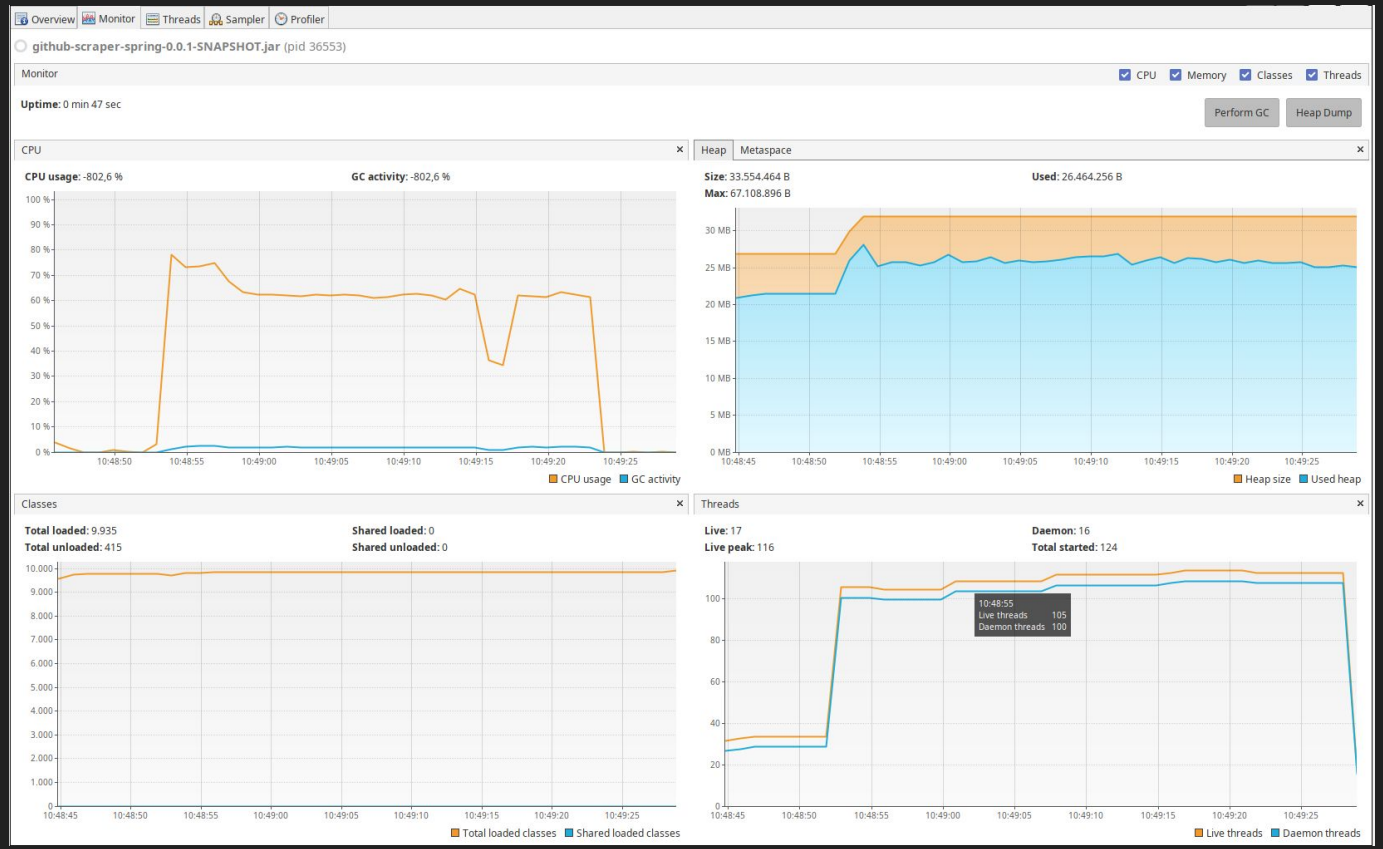
Minimal Heap Size - Micronaut (-Xmx32M)



Minimal Heap Size - Micronaut (-Xmx32M)

```
wrk -t 8 -c 80 -d 30s --latency http://localhost:8080/github/stars
Running 30s test @ http://localhost:8080/github/stars
  8 threads and 80 connections
  Thread Stats   Avg      Stdev    Max   +/- Stdev
    Latency  2.68ms   6.40ms 135.54ms   91.54%
    Req/Sec  11.34k   4.58k   23.37k   68.22%
  Latency Distribution
    50%  270.00us
    75% 2.58ms
    90% 8.02ms
    99%   27.00ms
  1717518 requests in 30.06s, 631.95MB read
  Socket errors: connect 0, read 0, write 0, timeout 80
Requests/sec:  57134.45
Transfer/sec:    21.02MB
```

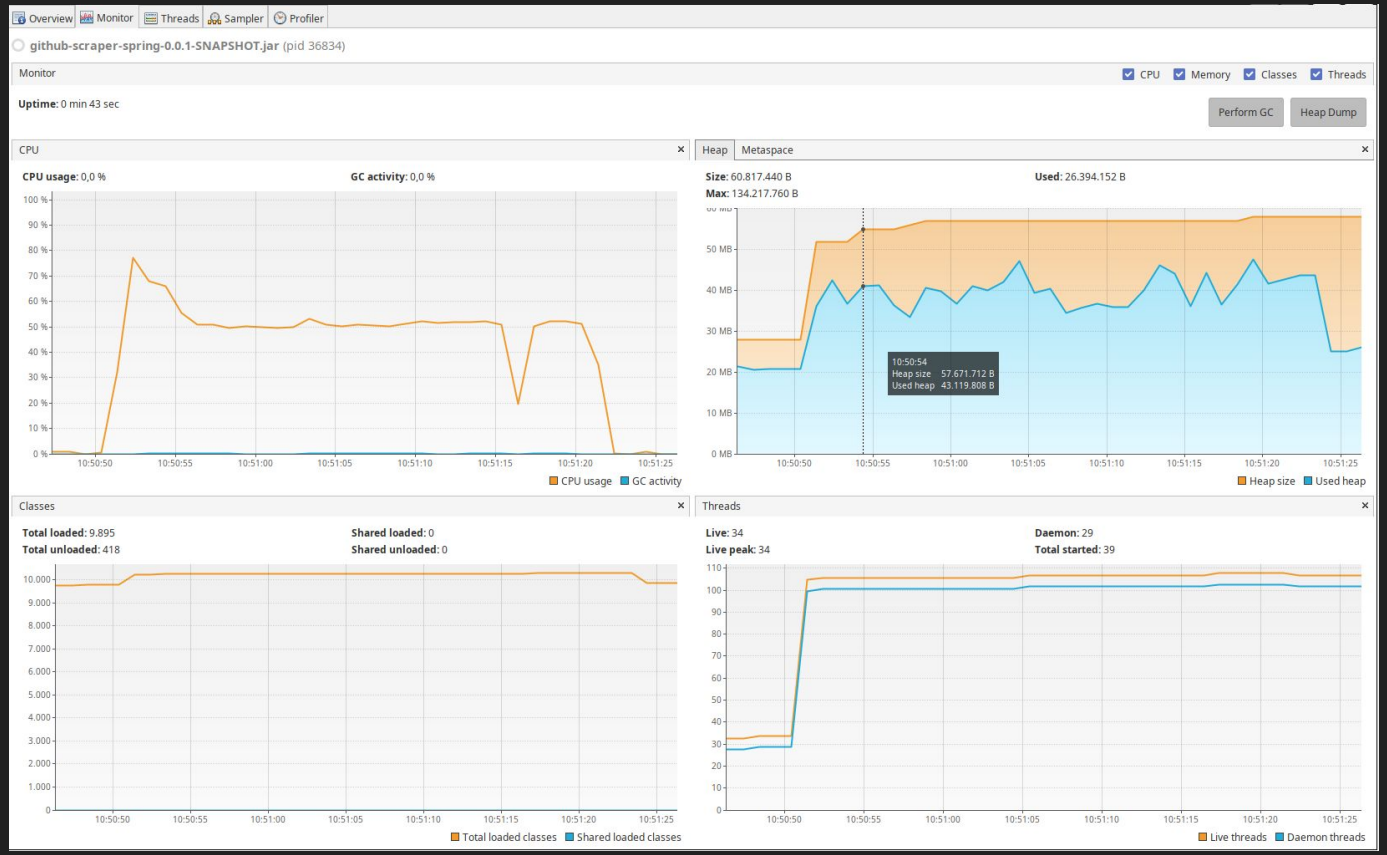
Minimal Heap Size - Spring (-Xmx32M)



Minimal Heap Size - Spring (-Xmx32M)

```
wrk -t 8 -c 80 -d 30s --latency http://localhost:8080/github/stars
Running 30s test @ http://localhost:8080/github/stars
  8 threads and 80 connections
  Thread Stats   Avg      Stdev    Max   +/- Stdev
    Latency  16.38ms   87.16ms 955.97ms   97.33%
    Req/Sec  3.92k    1.19k    6.21k    75.93%
  Latency Distribution
    50% 2.11ms
    75% 4.21ms
    90% 8.59ms
    99%  578.98ms
  905844 requests in 30.05s, 457.16MB read
Requests/sec:  30148.82
Transfer/sec:    15.22MB
```

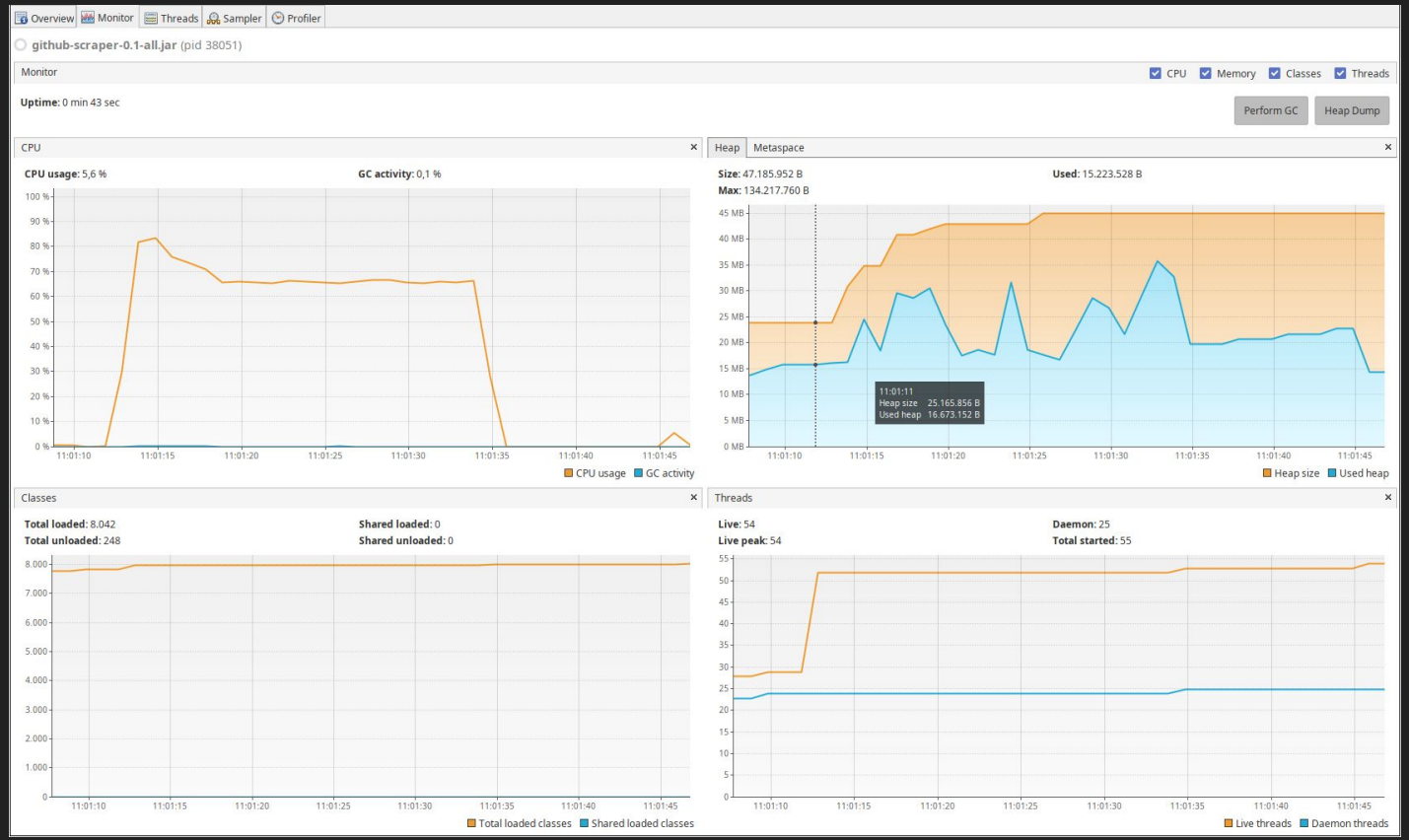
Minimal Heap Size - Spring (-Xmx64M)



Minimal Heap Size - Spring (-Xmx64M)

```
wrk -t 8 -c 80 -d 30s --latency http://localhost:8080/github/stars
Running 30s test @ http://localhost:8080/github/stars
  8 threads and 80 connections
  Thread Stats   Avg      Stdev    Max   +/- Stdev
    Latency  10.50ms   66.17ms 786.18ms   97.73%
    Req/Sec  8.60k    2.27k   13.25k   79.23%
  Latency Distribution
    50%  0.95ms
    75%  1.68ms
    90%  3.30ms
    99%  423.31ms
  2006502 requests in 30.09s, 0.99GB read
Requests/sec:  66689.38
Transfer/sec:    33.66MB
```

Minimal Heap Size - Micronaut (-Xmx64M)



Minimal Heap Size - Micronaut (-Xmx64M)

```
wrk -t 8 -c 80 -d 30s --latency http://localhost:8080/github/stars
Running 30s test @ http://localhost:8080/github/stars
  8 threads and 80 connections
  Thread Stats   Avg      Stdev    Max   +/- Stdev
    Latency  2.28ms   5.05ms  95.77ms   90.92%
    Req/Sec  14.11k   5.15k   25.09k   73.51%
  Latency Distribution
    50%  283.00us
    75% 2.12ms
    90% 6.78ms
    99%   23.54ms
  2451595 requests in 30.06s, 1.22GB read
Requests/sec:  81556.51
Transfer/sec:    41.69MB
```

Graal VM

“High-performance polyglot VM”
(and AOT compiler)

Enough bla bla, show code!

Graal Substrate VM

```
$ ./gradlew nativeImage (or ./mvnw package -Dpackaging=native-image)

… 2 minutes later …

$ file application
application: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked,
interpreter /lib64/ld-linux-x86-64.so.2 for GNU/Linux 3.2.0, with debug_info, not stripped

$ ll application
-rwxr-xr-x. 1 moe moe 62M 18. Dez 11:29 application

$ ldd application
     linux-vdso.so.1 (0x00007ffd7cbbe000)
     libpthread.so.0 => /lib64/libpthread.so.0 (0x00007fc717414000)
     libdl.so.2 => /lib64/libdl.so.2 (0x00007fc71740d000)
     libz.so.1 => /lib64/libz.so.1 (0x00007fc7173f3000)
     librt.so.1 => /lib64/librt.so.1 (0x00007fc7173e8000)
     libgcc_s.so.1 => /lib64/libgcc_s.so.1 (0x00007fc7173cd000)
     libc.so.6 => /lib64/libc.so.6 (0x00007fc717202000)
     /lib64/ld-linux-x86-64.so.2 (0x00007fc717456000)
```

Graal (no -Xmx set)

```
Startup completed in 67ms.
```

Idle:

PID	USER	PRI	NI	VIRT	RES	SHR	S	CPU%	MEM%	TIME+	START	Command
57535	moe	20	0	1390M	94524	46008	S	0.0	0.3	0:00.08	13:33	./application

KB = 92 MB

Under load:

PID	USER	PRI	NI	VIRT	RES	SHR	S	CPU%	MEM%	TIME+	START	Command
57535	moe	20	0	4135M	764M	49276	S	732.	2.4	1:42.99	13:33	./application

Graal - Benchmark

```
wrk -t 8 -c 80 -d 30s --latency http://localhost:8080/github/stars
Running 30s test @ http://localhost:8080/github/stars
  8 threads and 80 connections
  Thread Stats   Avg      Stdev    Max   +/- Stdev
    Latency  1.60ms   2.28ms  41.76ms   86.51%
    Req/Sec  8.88k    1.21k   12.09k   79.71%
  Latency Distribution
    50%  416.00us
    75% 2.34ms
    90% 4.61ms
    99%   10.16ms
  1355757 requests in 30.04s, 305.29MB read
  Socket errors: connect 0, read 0, write 0, timeout 80
Requests/sec:  45130.28
Transfer/sec:    10.16MB
```

Conclusion

Experience from the trenches

Pro:

- Services start faster than Spring Boot
- Smaller JARs
- Tests are faster, as the embedded server starts fast
  - Spring mitigates this with application context caching!

Contra:

- No separation of management and API port
- Spring Boot is more common (more documentation, blog posts, Stack Overflow answers)
- Extension story from Spring is better (e.g. OAuth2)

Conclusion

- Good documentation
- Recommendation for writing small services
  - I have no experience with bigger services in Micronaut. If you have, please contact me :)
- Development is fun
- If the features are sufficient: Try it!

- GraalVM Substrate VM, if you get it to run, is really cool
  - This is NOT a JVM, so profilers / metrics / thread dumps etc. won't work
  - Build times like it's C++ (> 30s)
  - Not all libraries are compatible and workarounds must be employed
    - I ran into trouble with Caffeine Cache (GitHub Issue)

Literature

- GitHub Repo: https://github.com/qaware/microservices-with-micronaut
- Micronaut documentation: https://docs.micronaut.io/latest/guide/index.html
- GraalVM: https://www.graalvm.org/

Questions?

Contact: moritz.kammerer@qaware.de

We are hiring.

In Munich. And in Mainz.

https://www.qaware.de/karriere/#jobs

QAWARE
SOFTWARE ENGINEERING