



Micronaut

Effiziente und performante
Microservices für die Cloud

Orientation in Objects GmbH

Weinheimer Str. 68
68309 Mannheim

www.oio.de
info@oio.de

Version: 1.0

Ihre Sprecher

Falk Sippach (@sippack)

Trainer, Berater, Entwickler



*Architektur
Agile Softwareentwicklung
Codequalität*

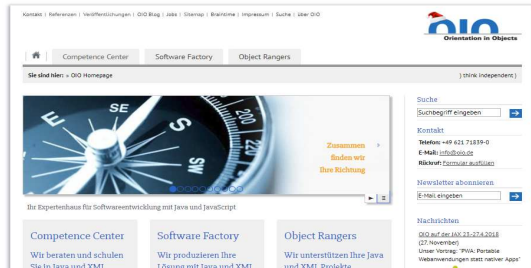


Co-Organisator



Committer DukeCon

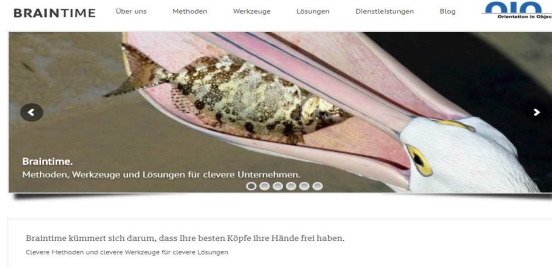
Zwei OIO Marken



<http://www.oio.de>

Ihr Expertenhaus für Softwareentwicklung mit Java und JavaScript

Schulung, Beratung und Programmierung



<http://www.braintime.de>

OIO Braintime
Ihr Atlassian Platinum Solution Partner

Methoden, Werkzeuge und Lösungen

Zusammenschluss Trivadis und OIO



- Im Mai 2018 haben sich Trivadis und Orientation in Objects (OIO) zusammengeschlossen. Gemeinsam stärken und erweitern wir unser Angebot im Bereich Java und agiler Softwareentwicklung.



Gemeinsam bieten wir ein **zwölfmonatiges Trainee-Programm** an, das Experten für Java- und Webtechnologien ausbildet.

Java Schulungen & Trainings



Neu finden Sie im Trivadis **Trainingsangebot** auch Kurse, die von der OIO entwickelt und durchgeführt werden.

Abstract



Den Chancen, die der Microservices-Ansatz bietet, stehen auch einige Herausforderungen gegenüber, die man aber gut mit Frameworks handhaben kann. Mit Micronaut hat nun ein ganz neuer Vertreter die Bühne mit dem Versprechen betreten, modulare, leicht testbare und sehr performante Anwendungen in Java, Kotlin oder Groovy entwickeln zu können.

Auch wenn Micronaut dem Platzhirsch aus dem Spring-Ökosystem ähnlich sieht, wurde es von Grund auf explizit für die Erstellung von Microservices im Cloud-Computing-Umfeld erstellt. Dank extrem kurzer Startzeiten, einem enorm niedrigen Speicherverbrauch und sehr kleinen JAR-Größen wird es die Microservices-Welt umkrempeln.

Ermöglicht wird das neuartige Programmiermodell mittels Compile-Zeit-Metaprogrammierung, wodurch die Metadaten für beispielsweise Dependency Injection und die aspektorientierte Programmierung bereits beim Kompilieren erzeugt werden. Reflection, Proxy Generierung und Data Caching zur Laufzeit entfallen dadurch. Zur Verwendung in der Cloud oder Serverless-Umgebungen gibt es zudem bereits zahlreiche fertig gestellte oder geplante Anbindungen an Service-Discovery-Dienste, Configuration Sharing, Load Balancing und Serverless Computing.



Einführung





MICRONAUT



Orientation in Objects

Natively **CLOUD NATIVE**



Cloud Native Computing Foundation



Orientation in Objects

... **skalierbare** Anwendungen
in modernen, **dynamischen Umgebungen** ...

... **lose gekoppelte Systeme**,
die **widerstandsfähig, kontrollierbar**
und **beobachtbar** sind.

... **robuste(n) Automatisierung** ... bei
minimalem Arbeitsaufwand ...

Quelle: <https://github.com/cncf/toc/blob/master/DEFINITION.md>

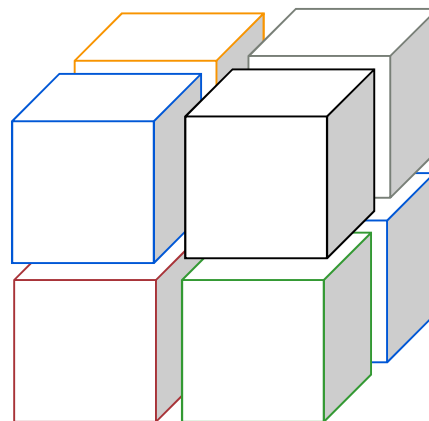
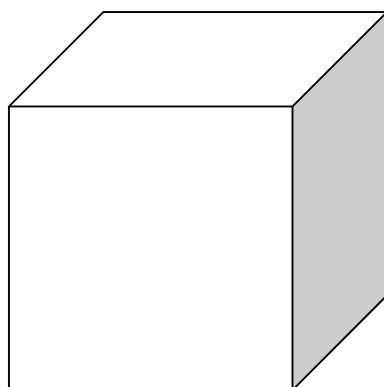
... Software **schneller, konsistent und zuverlässig** ... bereitstellen.

... **kontinuierliche** Bereitstellung,
DevOps und **Microservices**.

Quelle: <https://pivotal.io/de/cloud-native>



MICRONAUT





Herausforderungen

Wie sind die Microservices **konfiguriert**?

Welche Microservices sind **deployed** und wo?

Laufen alle Services?

Wie **kommunizieren** die Services?

Wie können **Fehlerketten verhindert** werden?

Wie kann man die Services **absichern**?

...



8 Irrtümer verteilter Systeme

Netzwerk ist **ausfallsicher**

Latenzzeit = \emptyset

Datendurchsatz ∞

Netzwerk ist **sicher**

Netzwerktopologie ist **stabil**

1 Netzwerkadministrator

Kosten des Datentransports = \emptyset

Netzwerk ist **homogen**



Herausforderungen in der Cloud



Teilen von Ressourcen (CPU/Speicher)

- Cloud-Laufzeitumgebung
- Elastische Anforderungen/Skalierung
- Lastverteilung, Ausfallsicherheit

Entkopplung durch verteiltes System

- Widerstandsfähigkeit/Robustheit
- Service Discovery, zentrale Konfiguration
- Tracing, verteiltes Logging, Monitoring



12 Factor App



Service Discovery / Orchestrierung

Konfiguration

Unveränderbare Deployments

Effiziente Service Interaktion

Elastische Skalierung

Cloud Bewusstsein

Monitoring

Tracing

Security

Widerstandsfähigkeit

Cloud Funktionen

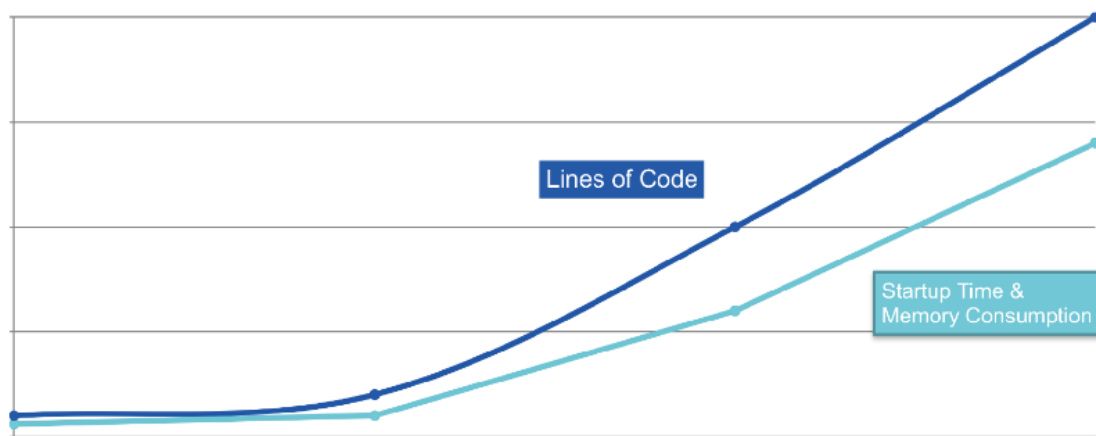
Quelle: <https://www.12factor.net/>



Werkzeug wählen



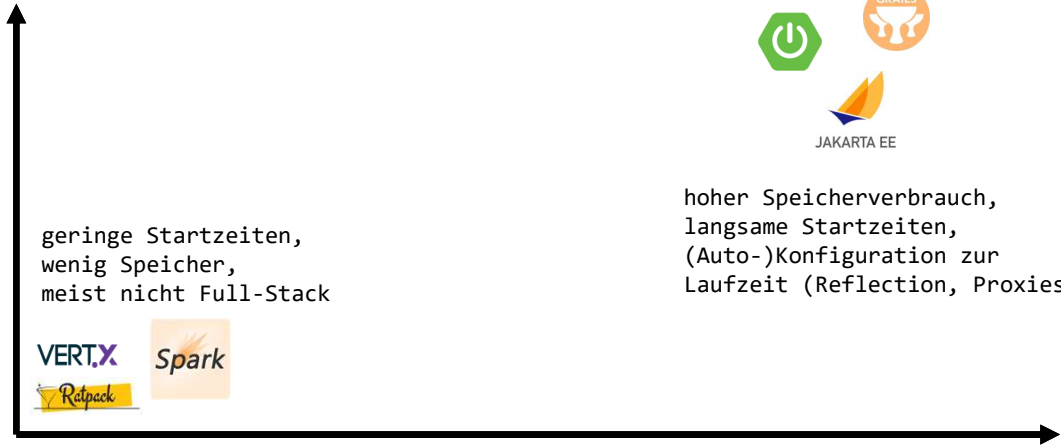
Probleme klassische Frameworks



Quelle: <https://github.com/micronaut-projects/presentations/blob/master/Greach-2018.pdf>



Alternative Frameworks



Echte Microframeworks

helidon.io

QUARKUS

Retpack

MICRONAUT

javalin

Spark

VERT.X

Ktor



Micronaut's **Cloud support is built right in**, including support for common **discovery services**, **distributed tracing tools**, and **cloud runtimes**.

Quelle: <https://micronaut.io/>



Hello Micronaut





Ziele

Konzipiert für Microservices und Cloud/Serverless

Extrem schnelle Startzeiten

Geringer Speicherverbrauch

Kleine Deployment-Artefakte

Wenig externe Abhängigkeiten



Polyglott





Installation CLI



```
$ sdk install micronaut 1.1.0
```

```
$ mn
```

```
| Starting interactive mode...
```

```
| Enter a command name to run. Use TAB for completion:
```

```
mn>
```



Projekt erstellen und bauen



```
$ mn create-app hello-world
```

oder

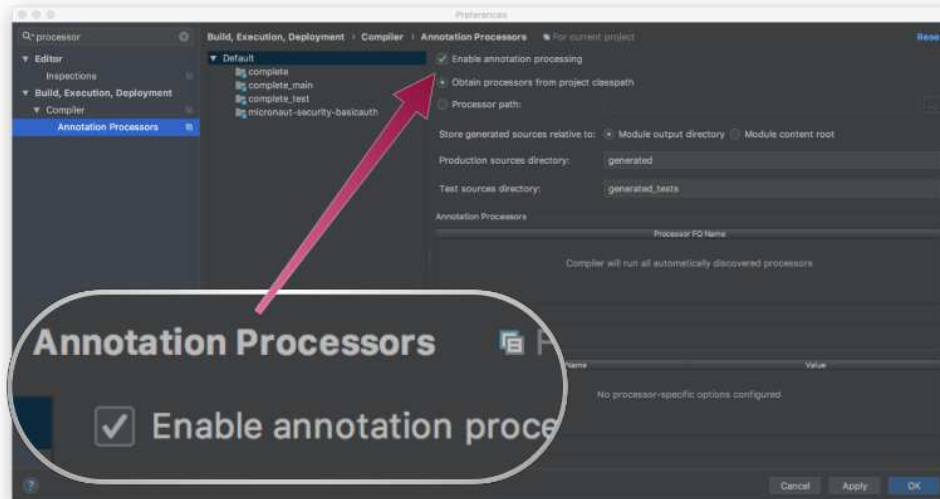
```
$ mn create-app de.oio.demo --lang kotlin --features http-client
```

Bauen:

```
$ ./gradlew build
```



IDE einrichten



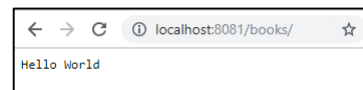
Implementierung Controller + main

```
import io.micronaut.http.MediaType;
import io.micronaut.http.annotation.*;
```

```
@Controller("/hello")
public class HelloController {
    @Get(produces = MediaType.TEXT_PLAIN)
    public String index() { return "Hello World"; }
}
```

```
import io.micronaut.runtime.Micronaut;
```

```
public class Application {
    public static void main(String[] args) {
        Micronaut.run(Application.class);
    }
}
```





Module/Feature hinzufügen

```
$ mn create-app book -feature bolt-neo4j

dependencies {
  annotationProcessor "io.micronaut:inject-java"
  annotationProcessor "io.micronaut:validation"
  compile "io.micronaut:inject"
  compile "io.micronaut:validation"
  compile "io.micronaut:runtime"
  compile "io.micronaut:http-client"
  compile "io.micronaut:http-server-netty"
  compile "io.micronaut.configuration:neo4j-bolt"
  compileOnly "io.micronaut:inject-java"
  runtime "ch.qos.logback:logback-classic:1.2.3"
  testCompile "junit:junit:4.12"
  testCompile "io.micronaut:inject-java"
  testCompile "org.hamcrest:hamcrest-all:1.3"
}
```



Verfügbare Features

```
$ mn profile-info service
Profile: service
-----
The service profile
...
Provided Features:
-----
annotation-api  Adds Java annotation API
config-consul   Adds support for Distributed Configuration with Consul (https://www.consul.io)
discovery-consul Adds support for Service Discovery with Consul (https://www.consul.io)
discovery-eureka Adds support for Service Discovery with Eureka
groovy          Creates a Groovy application
hibernate-gorm  Adds support for GORM persistence framework
hibernate-jpa   Adds support for Hibernate/JPA
http-client     Adds support for creating HTTP clients
http-server     Adds support for running a Netty server
java            Creates a Java application
jdbc-dbcps     Configures SQL DataSource instances using Commons DBCP
jdbc-hikari     Configures SQL DataSource instances using Hikari Connection Pool
jdbc-tomcat     Configures SQL DataSource instances using Tomcat Connection Pool
jrebel         Adds support for class reloading with JRebel (requires separate JRebel installation)
junit          Adds support for the JUnit testing framework
kafka          Adds support for Kafka
kotlin         Creates a Kotlin application
mongo-gorm     Configures GORM for MongoDB for Groovy applications
mongo-reactive Adds support for the Mongo Reactive Streams Driver
neo4j-bolt     Adds support for the Neo4j Bolt Driver
neo4j-gorm     Configures GORM for Neo4j for Groovy applications
...
```



Ausführen/Deployment

```
$ ./gradlew run
> Task :run
 [main] INFO io.micronaut.runtime.Micronaut -
Startup completed in 972ms. Server Running:
http://localhost:8080

$ java -jar build/libs/hello-world-all.jar

$ docker build -t mn-hello-world .
$ docker run --rm -d -p 8080:8080 mn-hello-world
```



Details





Interne Funktionsweise

Verschieben der Magie von der Lauf- zur **Compile-Zeit** für:

- Konfiguration
- Dependency Injection
- Profile Aktivierung
- AOP
- HTTP-Routing
- Finder Method Resolution



Dependency Injection

IoC-Container basiert auf **Code-Generation**.

Alle annotierten Beans werden beim Build-Prozess entsprechend hart verlinkt.

JVM kann generierten Code **inlinen** (optimieren).

Kein Classpath-Scan beim Start der Anwendung.

Kein Laufzeit-Cache von Reflection-Infos.



Dependency Injection

```
interface Engine {
    int getCylinders()
    String start()
}

@Singleton class V8Engine implements Engine {
    int cylinders = 8
    String start() { "Starting V8" }
}

@Singleton class Vehicle {
    final Engine engine
    Vehicle(Engine engine) { this.engine = engine }
    String start() { engine.start() }
}
```



Dependency Injection

```
import io.micronaut.context.*;

Vehicle vehicle =
    BeanContext.run().getBean(Vehicle.class);

System.out.println(vehicle.start()); // "Starting V8"
```



BeanContext

Typ	Beschreibung	Beispiel
java.util.Optional	An Optional of a bean. If the bean doesn't exist empty() is injected	Optional<Engine>
java.lang.Iterable	An Iterable or subtype of Iterable (example List, Collection etc.)	Iterable<Engine>
java.util.stream.Stream	A lazy Stream of beans	Stream<Engine>
Array	A native array of beans of a given type	Engine[]
Provider	A javax.inject.Provider if a circular dependency requires it	Provider<Engine>



DI-Container standalone

```
// Annotation Processing Plugin
plugins { id "net.ltgt.apt" version "0.18" }

...

dependencies {
    annotationProcessor "io.micronaut:micronaut-inject-java:1.0.4"
    compile "io.micronaut:micronaut-inject:1.0.4"
    ...
}
```



Mehrdeutige Beans auflösen

@Qualifier

```
@Retention(RUNTIME)
```

```
@interface V8 { }
```

```
@Inject Vehicle(@V8 Engine engine) {  
    this.engine = engine  
}
```



Eingebaute Scopes

Typ	Beschreibung
@Singleton	Singleton scope indicates only one instance of the bean should exist
@Context	Context scope indicates that the bean should be created at the same time as the ApplicationContext (eager initialization)
@Prototype	Prototype scope indicates that a new instance of the bean is created each time it is injected
@Infrastructure	Infrastructure is a @Context scope stereotype that indicates the bean cannot be replaced
@ThreadLocal	@ThreadLocal scope is a custom scope that associates a bean per thread via a ThreadLocal
@Refreshable	@Refreshable scope is a custom scope that allows a bean's state to be refreshed via the /refresh endpoint.



Conditional Beans

```
@Singleton
@Requires(beans = DataSource.class)
@Requires(property = "datasource.url")
public class JdbcBookService implements BookService {
    DataSource dataSource;

    public JdbcBookService(DataSource dataSource) {
        this.dataSource = dataSource;
    }
}
```



Lebenszyklus Methoden

```
import javax.annotation.PostConstruct
import javax.inject.Singleton

@Singleton
class V8Engine implements Engine {
    int cylinders = 8
    boolean initialized = false

    String start() {
        if(!initialized) throw new IllegalStateException("Engine not initialized!")

        return "Starting V8"
    }

    @PostConstruct
    void initialize() {
        this.initialized = true
    }
}
```



Events

```
@Singleton
public class MyBean {
    @Inject ApplicationEventPublisher eventPublisher;
    void doSomething() {
        eventPublisher.publishEvent(...);
    }
}

@Singleton
public class OnStartup implements ApplicationEventListener<ServiceStartedEvent> {
    @Override
    void onApplicationEvent(ServiceStartedEvent event) {
        ...
    }
}
```



Konfiguration

- ähnliches Vorgehen wie bei Spring Boot mit externen Properties-/JSON-/YAML-Dateien
- Überschreibbar durch Umgebungs- und Systemvariablen
- Injectpoints:

```
@Value("${http.port}")
private int port;

@Controller("${api.version}/list")
public class ...
```
- zusätzlich Verwendung von Profilen/Environments für spezifische Konfigurationen oder bedingte Bean-Instanzierungen



Persistenz

- relationale Datenbanken via JDBC/JPA/Hibernate
- MongoDB
- Neo4J
- Redis
- Cassandra
- ORM (objektrelationales Mapping) über GORM (auch für MongoDB und Neo4J)



Neo4jRepository

```
public interface BookRepository {
    void save(Book b);
    Stream<Book> findByTitle(String title);
}

@Singleton
public class BookNeo4jRepository implements BookRepository {
    @Inject Driver driver;

    public void save(Book book) {
        try (Session s = driver.session()) {
            String stmt = "MERGE (b:Book {id:$book.id}) ON CREATE SET b+=$book";
            s.writeTransaction(tx -> tx.run(stmt, singletonMap("book", book.asMap())));
        }
    }
}
```



Verwendung Repository

```
@Controller("/book")
public class BookController {
    @Inject BookRepository repo;
    @Get("/titled/{title}")
    public Stream<Book> books(String title) {
        return repo.findByTitle(title);
    }
}

dependencies {
    ...
    compile "io.micronaut.configuration:neo4j-bolt"
}
```



AOP

Querschnittsbelange (Logging, Trx, Monitoring, ...)

Generieren von konkreten Aspekt-Klassen zur Compilezeit

```
@Singleton
class CacheInterceptor implements
MethodInterceptor<Object, Object> {
    @Inject Cache<MethodCall, Object> cache;
    public Object intercept(
        MethodInvocationContext<Object, Object> context) {
        return cache.computeIfAbsent(methodCall(context),
            call -> context.proceed);
    }
}
```



Anwendung Interceptor

@Around

@Type(CacheInterceptor.class)

@Target(ElementType.METHOD)

public @interface Memoized {}

@Memoized

BigInteger fibonacci(BigInteger number) { ... }



Eingebaute AOP-Mechanismen

Validation (JSR-303 über Hibernate Validator)

Caching (synchronous and asynchronous), (Redis) **@Cacheable**

Retry with **@Retryable** also on asynchronous methods

Circuit Breaker mit **@CircuitBreaker**

Zeitversetzte Ausführung mit **@Scheduled**

Transaktionale Steuerung mit **@Transactional**



Timer/Scheduler

```
$ mn create-job Book
```

```
@Singleton
public class BookJob {
    @Inject BookClient client;
    @Inject BookRepository repo;

    @Scheduled(fixedRate = "5m")
    public void process() {
        client.books(10).forEach(repo::save);
    }
}
```



HTTP-Server

- asynchron mit reaktiven Ergebnistypen als Event-Loop ist Default (basiert auf Netty)
- application/json ist Default
- @Blocking bei synchronen Service-Aufrufen (z. B. JDBC)
- synchrone Controller sind möglich und nutzen dann den klassischen IO-Thread-Pool



HTTP-Server

- zustandlos bei Default, Redis-basierte In-Memory-Session ist konfigurierbar
- Fehlerbehandlung über selbst definierte Methoden

```
@Error
```

```
void catchMyException(MyException e) {...}
```



HTTP-Server

```
mn create-controller Book
```

```
@Controller ("/book")
```

```
public class BookController {  
    @Get ("/{isbn}")  
    public Single<Book> get (String isbn) {  
        return Single.just(repo.findByIsbn(isbn));  
    }  
}
```



HTTP-Client (deklarativ)

```
mn create-client Book
```

```
@Client("/book")
public interface BookClient {
    @Get("/{isbn}")
    public Book get(String isbn);
}

@Controller("/library")
public class LibraryController {
    @Inject BookClient client;
    ...
}
```



Integrations-Tests

```
public class BookControllerTest {

    @Test
    public void testIndex() throws Exception {
        EmbeddedServer server =
            ApplicationContext.run(EmbeddedServer.class);

        RxHttpClient client =
            server.getApplicationContext()
                .createBean(RxHttpClient.class, server.getURL());

        assertEquals(client.toBlocking().exchange("/books").status(),
            HttpStatus.OK);
        server.stop();
    }
}
```



Cloud Unterstützung



Distributed Configuration

- Consul, AWS Parameter Store

Service Discovery

- Consul, Eureka, Kubernetes, AWS Route 53

Client Side Load Balancing mit Netflix Ribbon

Distributed Tracing mit Zipkin oder Jaeger

Serverless Functions für AWS Lambda

Message Driven Microservices mit Kafka



Service Discovery mit Consul



```
$ docker run -p 8500:8500 consul
```

```
src/main/resources/application.yml:
```

```
micronaut:  
  application:  
    name: book  
  consul:  
    client:  
      registration:  
        enabled: true  
        defaultZone: "${CONSUL_HOST:localhost}:${CONSUL_PORT:8500}"
```

```
@Client(id = "book")  
public interface BookClient { //... }
```



Load Balancing

Eigene Client Implementierung (Round-Robin)

Netflix Ribbon hinzukonfigurierbar:

```
ribbon:  
  VipAddress: test  
  ServerListRefreshInterval: 2000
```



Retry

```
@Retryable( attempts = "${retry.attempts:3}",  
           delay = "${retry.delay:1s}" )  
@Client("book")  
public interface BookClient { ... }
```



Fallback

@Fallback

```
public class PetFallback implements PetOperations {  
    @Override  
    public Single<Pet> save(String name, int age) {  
        Pet pet = new Pet();  
        pet.setAge(age);  
        pet.setName(name);  
        return Single.just(pet);  
    }  
}
```



Circuit Breaker

```
@CircuitBreaker(reset = "30s")  
public List<Book> findBooks() {  
    ...  
}
```



Performance



Performance-Vorteile



Annotation Processor (Java, Kotlin), **AST Transformations** (Groovy)

-> Generieren von Klassen (implements BeanDefinition)

-> zur Compile-Zeit mit ASM

Micronaut kennt **Injection-Points vorher**

-> kein Scannen der Methoden, Felder, Konstruktoren zur Laufzeit

JVM kann optimieren (Inline), weil **kein Reflection**

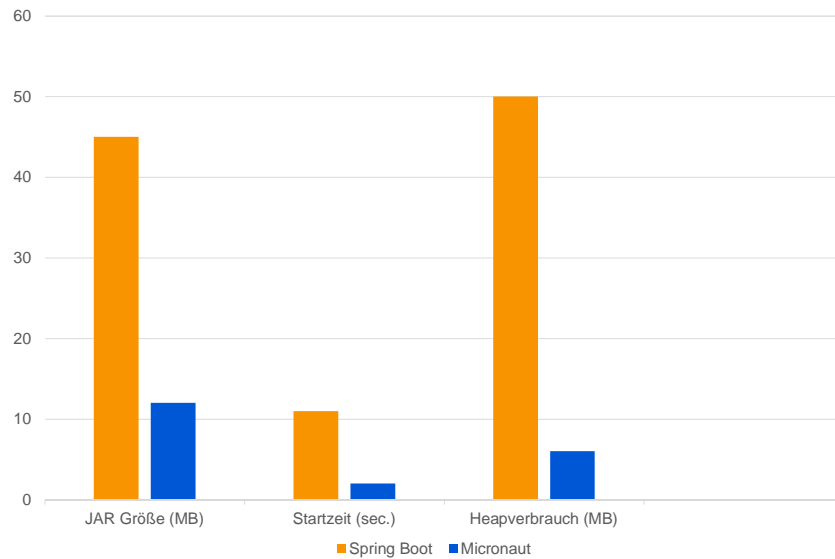
-> Laufzeit verbessert und geringerer Speicherverbrauch

Startzeit und Speicherverbrauch ist **nicht abhängig von der Projektgröße**

-> wird nicht langsamer bei steigender Anzahl Klassen



Spring Boot vs. Micronaut



Spring Boot vs. Micronaut



	Spring Boot	Micronaut JVM	Micronaut Graal Native
Compile-Zeit	2 sec	10 sec	150 sec
Größe Deployment-Artefakt	15 MB	11 MB	41 MB
Startzeit	5 sec	3 sec	16 ms
RAM-Verbrauch	290 MB	194 MB	23 MB

Quelle: <https://www.adesso.de/de/news/blog/micronaut-eine-alternative-zu-spring-4.jsp>



GraalVM Native Image

Polyglotte VM von Oracle (JavaScript, Ruby, ...)

GraalVM Native erzeugt Binaries, idealerweise:

- keine Reflection
- kein dynamische Class-Loading
- wenig Drittbibliotheken



Fazit





MICRONAUT



Orientation in Objects

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

Quelle: <https://micronaut.io/>



Alternativen



Orientation in Objects



MICRONAUT

QUARKUS





Fazit

beeindruckende **Geschwindigkeit**, geringer **Memory-Footprint**

bereits viel **Funktionalität out-of-the-box**

wenige Beispiele, aber **gute Doku**

Ähnlichkeit zu Spring/CDI erleichtert Einstieg

auch für Android geeignet (keine Reflection)



Vorteile

Kaum Verwendung von Reflection

Kaum Verwendung von Laufzeit-Proxy-Klassen

Full-Stack-Framework

Polyglott

Einfaches Unit-/Integrations-Testen



Kompromisse



langsames Kompilieren, keine inkrementellen Builds möglich

Build-Zeiten gehen hoch

Debugging durch die vielen generierten Klassen erschwert: A, B -> A, A\$1, A\$2, B, B\$1, B\$1\$1



Micronaut Framework
@micronautfw

Micronaut is officially on the @thoughtworks Technology Radar under Assess! This is a great milestone and we are very excited! thoughtworks.com/radar/language...
#TWTechRadar #micronautfw

[Tweet übersetzen](#)



Quelle: <https://twitter.com/micronautfw/status/1121058081830526976?s=03>



Fragen ?

Orientation in Objects GmbH

Weinheimer Str. 68
68309 Mannheim

www.oio.de
info@oio.de



**Vielen Dank für Ihre
Aufmerksamkeit!**

Orientation in Objects GmbH

Weinheimer Str. 68
68309 Mannheim

www.oio.de
info@oio.de