GEDOPLAN

Vom Enterprise-Java-Monolithen zu Microservices in der Container-Welt

OWL Tech & Innovation Day

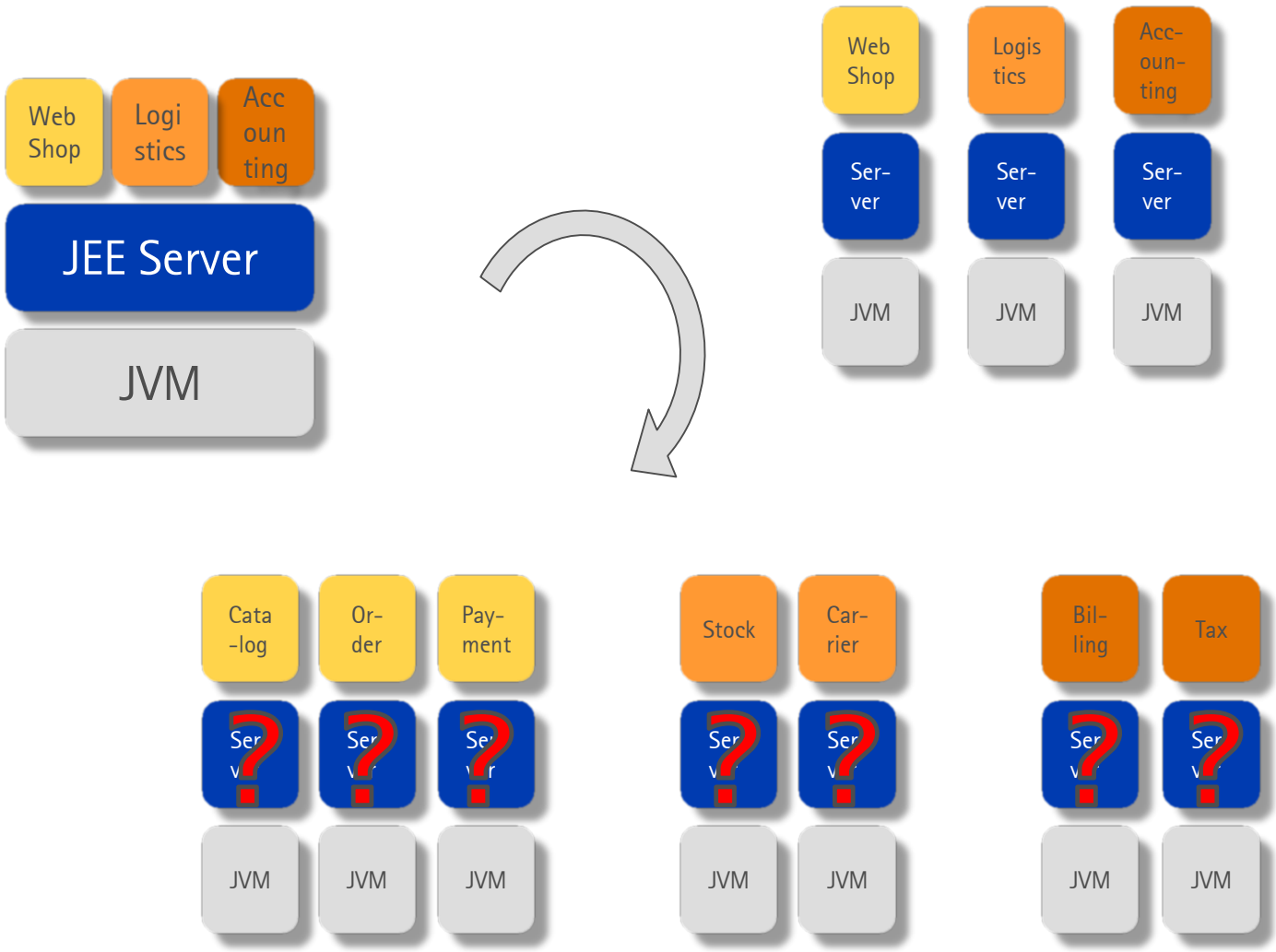
26.09.2019

Dirk Weil, GEDOPLAN GmbH

Dirk Weil

- GEDOPLAN GmbH, Bielefeld
  - GEDOPLAN IT Consulting
    Consulting, coaching, concepts, reviews, development
  - GEDOPLAN IT Training & partner
    Java, JEE, tools trainings in Berlin, Bielefeld, Köln, on-site
- JEE since 1998
- Speaker and author

Monolith → Microservices



JEE

REST

JS UI

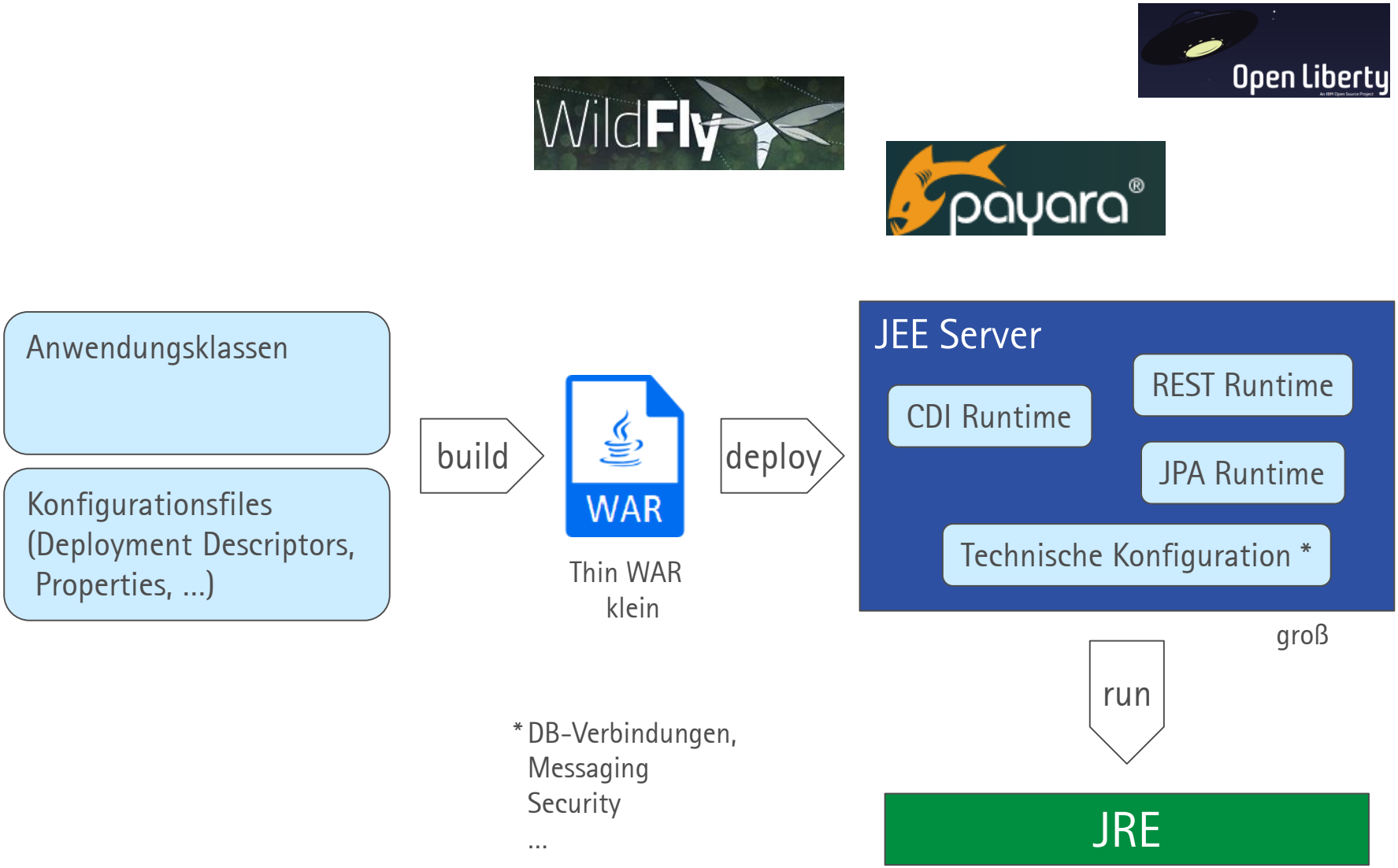
„So neue Sachen kannste mit Java EE nicht machen!“

Monitoring

Leichtgewichtig

Health

Der schwergewichtige (?) Application Server



Die leichtgewichtige (?) Alternative: „Micro“ Framework

Anwendungsklassen

Konfigurationsfiles *

REST Runtime

CDI Runtime

JPA Runtime

* Anwendungsparameter,
DB-Verbindungen,
Messaging
Security
...

build

Fat JAR
JAR + Dependencies
groß

run

JRE

Vergleichanwendung

- `https://github.com/GEDOPLAN/micro-comparison`
- REST-Service mit Persistenz
  - JPA + CDI/Spring + JAX-RS/Spring Web
- lauffähig als WAR bzw. Fat JAR oder Docker Image
- Implementierungsvarianten
  - klassisches WAR für WildFly (und andere JEE-Server)
  - Spring Boot
  - Quarkus
  - KumuluzEE

Kandidaten

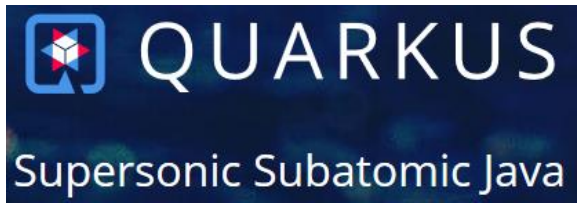
- WildFly 16.0.0
  - klassischer Application Server
  - JEE 8 + MicroProfile 2.2
  - mittels Galleon provisionierbar

- Spring Boot 1.5.16
  - populäre Alternative zu Java EE
  - Runtime wird durch Dependencies kombiniert
  - große Vielfalt an 3rd-Party-Anbindungen
  - Autokonfiguration
  - nutzt viele JEE-Anteile

Kandidaten

- KumuluzEE
  - kombiniert Runtime aus JEE-Dependencies
    - Jetty, Weld, Jersey, Hibernate, ...
  - Anwendungscode ist 100% Standard-JEE

- Quarkus
  - kombiniert Runtime aus JEE-Dependencies
  - spezieller CDI-Container ARC
    - Subset von CDI 2.0
    - Optimierung schon zur Build-Zeit
  - Hot Reloading im Dev-Modus



„Gewicht“: Anwendungscode

- z. B. Persistenz

```
@ApplicationScoped
@Transactional(rollbackOn = Exception.class)
public class PersonRepository extends SingleIdEntityRepository<Integer, Person> {

  public List<Person> findByName(String name) {
    return findMultiByProperty(Person_.name, name);
  }
```


```
@Transactional(propagation = Propagation.REQUIRED, rollbackFor = Exception.class)
public interface PersonRepository extends CrudRepository<Person, Integer> {

  public List<Person> findByName(String name);
}
```

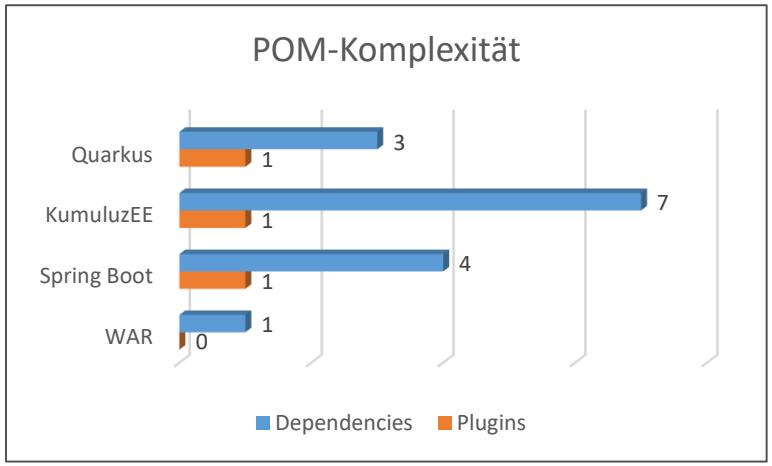
„Gewicht“: Anwendungscode

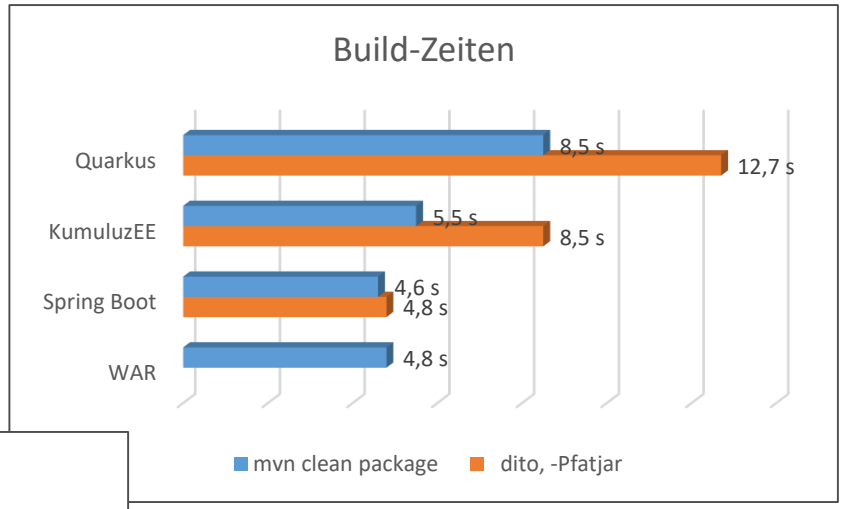
- z. B. REST-Endpunkt

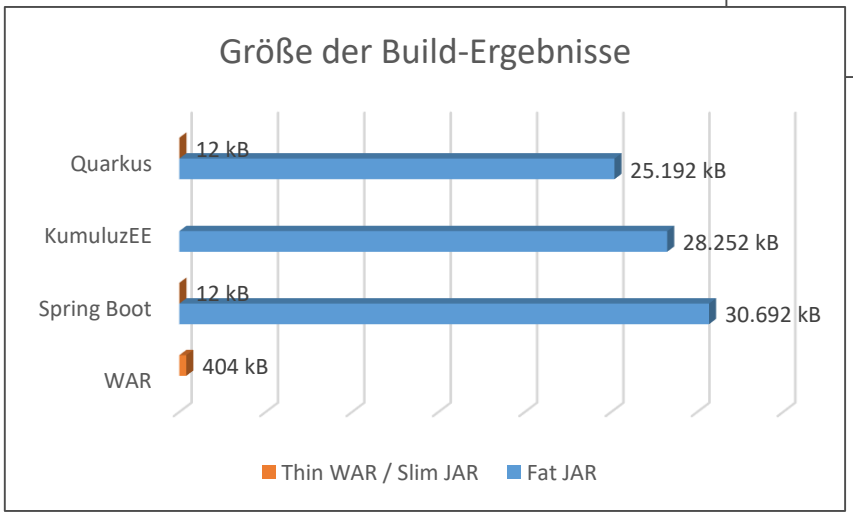
```jakarta
@ApplicationScoped
@Path("person")
public class PersonResource {

  @Inject
  PersonRepository personRepository;

  @GET
  @Produces(MediaType.APPLICATION_JSON
  public List<Person> getAll() {
    return personRepository.findAll();
  }
```

```spring
@RestController
@RequestMapping("/person")
public class PersonResource {

  @Autowired
  private PersonRepository personRepository;

  @RequestMapping
  public Iterable<Person> getAll() {
    return personRepository.findAll();
  }
```

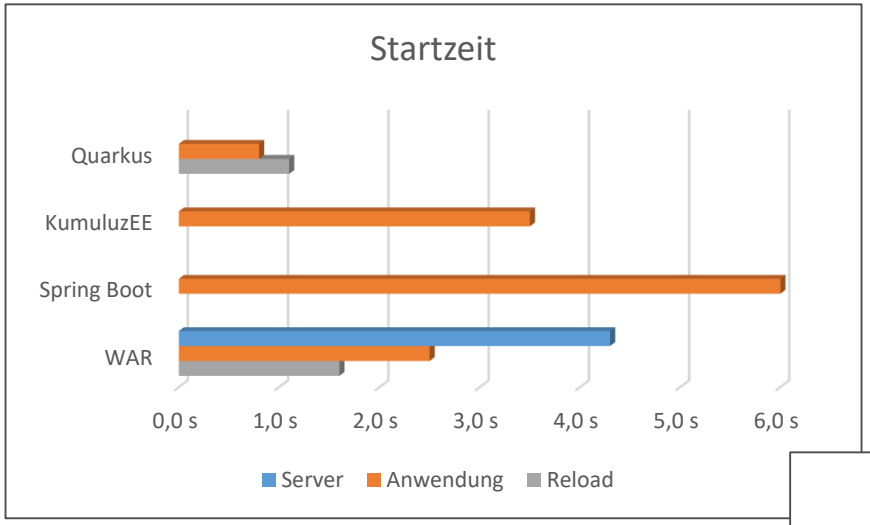
„Gewicht“: Build

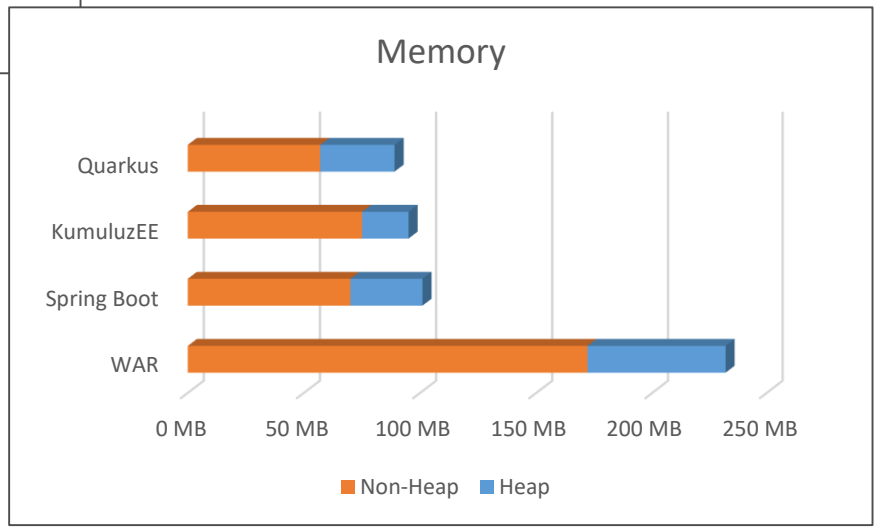






„Gewicht“: Laufzeit





Test

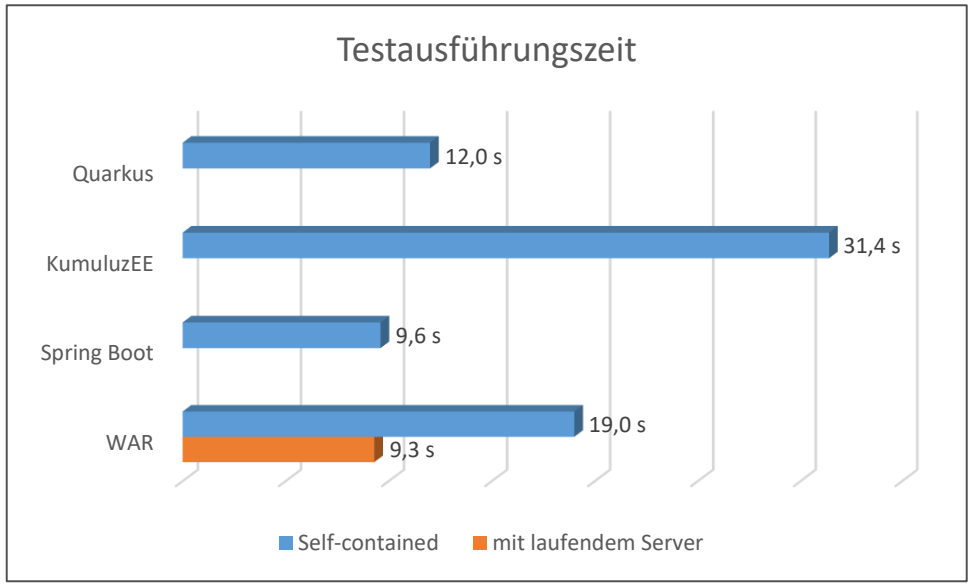
- Arquillian

  - De-facto-Standard für JEE-Tests
  - Server Lifecycle, Deployment, Test
  - komplex
    - Auswahl und Konfiguration der Zielumgebung
    - Zusammenstellung des Deployments
  - Im Demo-Projekt für WildFly und KumuluzEE genutzt

- `@SpringBootTest`: Spring-eigener Test Runner
  - Server Lifecycle, Test
  - einfach

- `@QuarkusTest`: Quarkus-Pendent zu `@SpringBootTest`

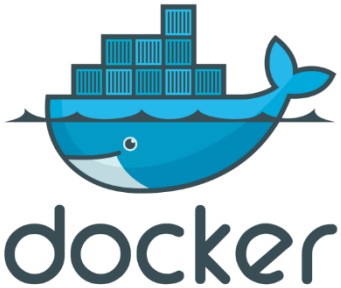
Test

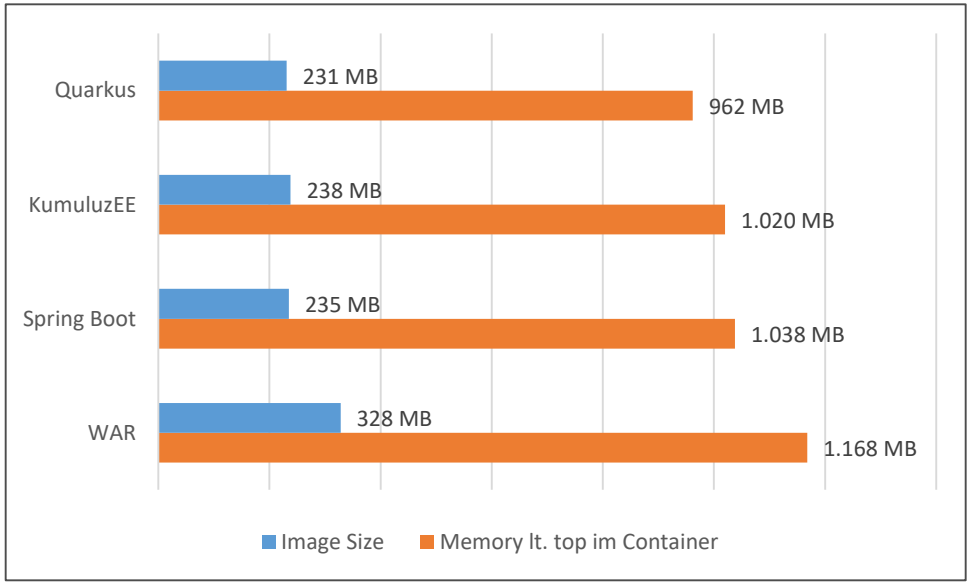

Testausführungszeit
Quarkus
12,0 s
KumuluzEE
31,4 s
Spring Boot
9,6 s
WAR
19,0 s
9,3 s
Self-contained
mit laufendem Server

Betriebsmodell

- „Server mit Deployment, Server eingebettet – ist das nicht zukünftig eh egal?“

- Container
  - OS
  - \+ JVM
  - \+ Server?
  - \+ Anwendung





Health Monitoring, Metrics

- MicroProfile Health
  - https://javaeeblog.wordpress.com/2019/01/27/alles-gesund-health-checking-mit-microprofile-health/
- MicroProfile Metrics
  - https://javaeeblog.wordpress.com/2019/03/15/wie-laeufts-denn-so-monitoring-mit-microprofile-metrics/

Fazit

- Micro-Frameworks auch für JEE verfügbar
- Anwendungscode von JEE und Spring Boot sehr ähnlich
- Projekt-Setup für klassische JEE-Anwendung einfacher
  - aber: Server notwendig
- Entwicklungs-Zeiten für JEE und Micro vergleichbar kurz
  - klassische Deployzeiten vergleichbar mit Hot Reload
- Footprint von Micro-Frameworks kleiner
  - insb. Quarkus durch Build-Time-Optimierung
- Testen im klassischen JEE-Umfeld zu kompliziert
  - Spring Boot und Quarkus richtungsweisend
- Server separat oder embedded tritt in den Hintergrund

Fazit

„... kannste sehr wohl machen!“

Ausblick

- Quarkus (u. a.) bietet
  - Compile in Maschinencode (GraalVM)
    ➔ Startzeiten im ms-Bereich
    ➔ Cloud, ggf. serverless
  - Reactive Programming
    - non-blocking
    - asychron
    - abstraktes Producer/Consumer-Konzept

More

- `github.com/GEDOPLAN/micro-comparison`
  Demo project

- `www.gedoplan-it-training.de`
  Trainings in Berlin, Bielefeld, inhouse
  - neu: JEE Microservice Foundation
  - neu: Java DevOps: Development und Delivery mit Docker und Kubernetes

- `www.gedoplan-it-consulting.de`
  Reviews, Coaching, ...
  Blog

- `dirk.weil@gedoplan.de`
- `@dirkweil`