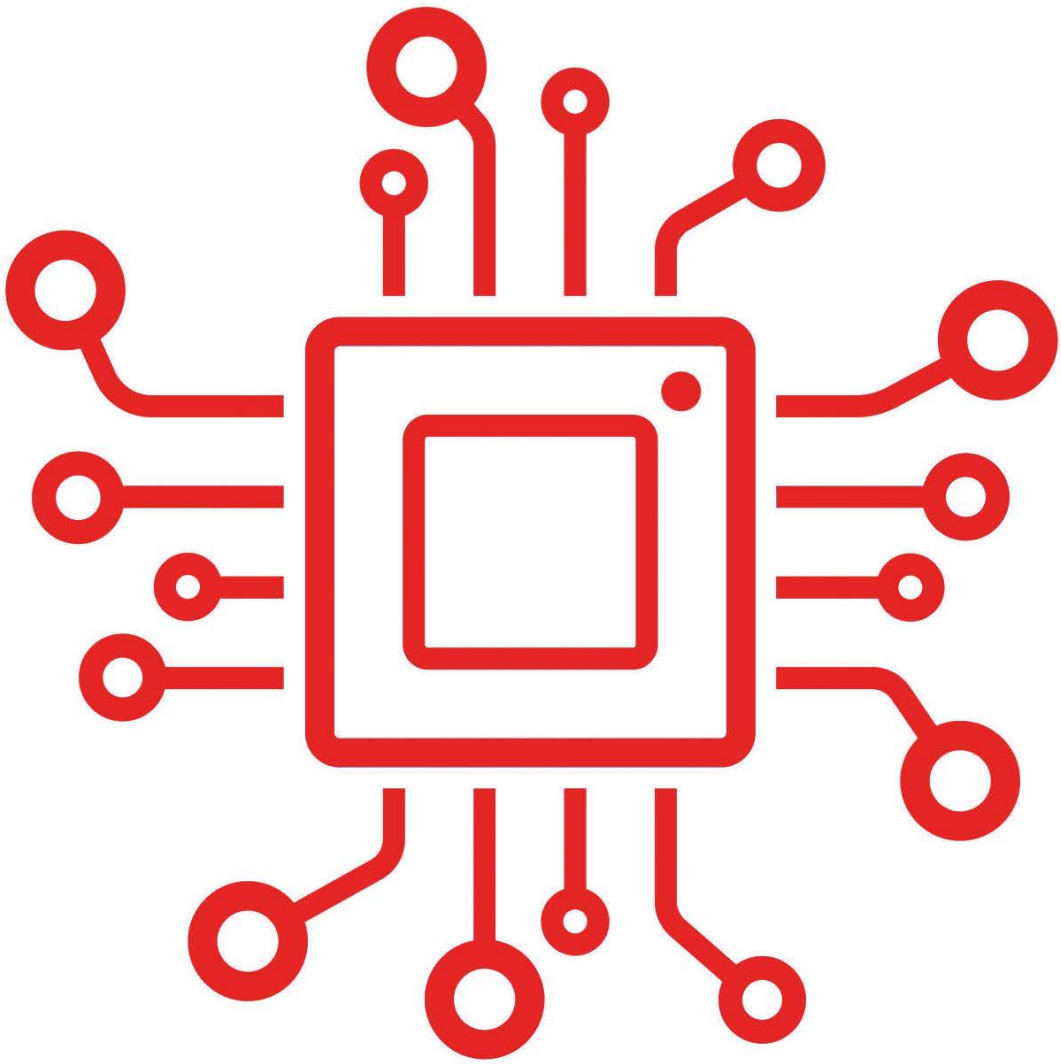

Cloud-native Microservice-Entwicklung mit Helidon

Sven Bernhardt, Opitz Consulting Deutschland

Schneller, besser und innovativer – das sind Kernattribute, die Unternehmen heute befähigen, langfristig rentabel und konkurrenzfähig zu bleiben. Cloud-Plattformen bilden hierfür eine wichtige Basis, denn die Cloud ist ein wichtiges Vehikel, um innovative Ideen schnell und effizient voranzubringen. Dafür braucht es allerdings zwingend Applikationen, die in der Lage sind, die Cloud richtig zu nutzen und ein Umdenken in der Art und Weise, wie Applikationen implementiert und betrieben werden: Das Zauberwort heißt Cloud-native.

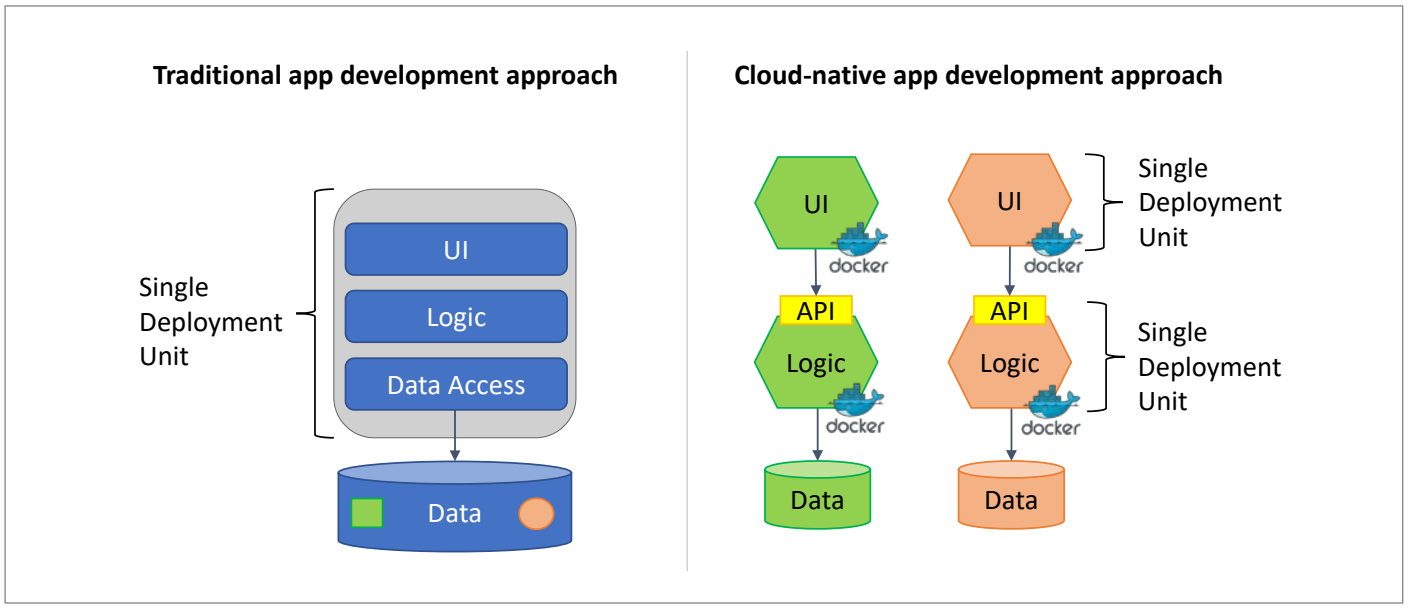

Abbildung 1: Traditionelle vs. Cloud-native Applikationen (Quelle: Sven Bernhardt)

Cloud-native Applikationen sind Applikationen, die von Anfang an mit speziellem Fokus auf die Cloud entwickelt werden. Im Gegensatz zu klassischen On-Premises-Applikationen, die mit „Lift and Shift" ohne jegliche architektonische Veränderungen in die Cloud gebracht werden, sind Cloud-native Applikationen in der Lage, die besonderen Möglichkeiten der Cloud-Technologie optimal auszuschöpfen, wie zum Beispiel die quasi unbegrenzte Skalierbarkeit. Dafür braucht es eine neue Form der Applikationsarchitektur und -implementierung.

Cloud-native Applikationen

Cloud-native Applikationen sind speziell für die Cloud konzipiert. Aber was bedeutet das eigentlich konkret?

Die Cloud Native Computing Foundation (CNCF) beschreibt Cloud-native Applikationen als

- lose gekoppelt,
- belastbar („resilient"),
- handhabbar („manageable")
- und beobachtbar („observable").

Zudem müssen Änderungen an solchen Applikationen schnell und zuverlässig durchgeführt werden können, was durch eine robuste und durchgängige Automatisierungsstrategie zu untermauern ist. *[1]*

Abbildung 1 zeigt, was die CNCF-Definition in Bezug auf eine Operationalisierung konkret bedeutet. Technologisch setzt Cloud-native massiv auf Containerisierung. Architektonisch stehen vor allem Konzepte wie Microservices, APIs und DevOps im Vordergrund.

Die Implementierung von Cloud-nativen Applikationen erfolgt unter Berücksichtigung der sogenannten Twelve-Factor-App-Prinzipien. Hierbei handelt es sich nicht etwa um neue, bahnbrechende Entwicklungsprinzipien, sondern eher um eine Sammlung von bereits etablierten Entwicklungspraktiken, wie beispielsweise die Verwaltung des Sourcecodes in einem Versionsmanagementsystem oder die Externalisierung von Konfigurationen in Umgebungsvariablen. *[2]*

Cloud-native Java

Wie bereits erwähnt, werden Cloud-native Applikationen containerisiert betrieben. Dies sollte bereits bei der Entwicklung beachtet werden. Bei einer Java-basierten Anwendung bedeutet das beispielsweise, dass diese „standalone" in der JVM gestartet und betrieben werden kann (zum Beispiel mittels Kommandozeilen-Befehls `java -jar app.jar`). Das verwendete Frameworks sollte also entsprechende Mechanismen bereitstellen.

Ein prominentes Beispiel hierfür ist Springboot *[3]*, das im Umfeld der Cloud-native Entwicklung bei Java-Entwicklern sehr beliebt ist. Springboot kann auf alles zurückgreifen, was das Spring-Ökosystem zu bieten hat und kann somit eine Vielzahl verschiedenster Anwendungsszenarien abdecken. Das bedeutet leider Segen und Fluch zugleich: Entwickler, die das Spring-Framework nur wenig kennen, verlieren ob des Umfangs und der Komplexität des Ökosystems schnell den Überblick. Die Lernkurve für Spring-Neulinge ist dementsprechend hoch. Das Schreiben einfacher CRUD-Services kann so zu einer großen Herausforderung werden.

Da aber die Java-Community stets aktiv und kreativ ist, hat sich im Bereich der Microservice-Entwicklung in den letzten Jahren einiges getan. Neben Springboot existieren daher mittlerweile eine Vielzahl leichtgewichtiger Frameworks, die die Entwicklung von Microservices einfach und effizient machen. *Abbildung 2* zeigt eine Übersicht der aktuell prominentesten Frameworks in diesem Bereich. Dabei handelt es sich bei der Übersicht um eine Momentaufnahme, die keinen Anspruch auf Vollständigkeit hat.

Neben den sogenannten Full Stack Frameworks zeigt die Abbildung auch MicroProfile-Frameworks sowie Microframeworks.

MicroProfile ist eine Community-getriebene Spezifikation zur Entwicklung von Java-Enterprise-basierten Microservices. Diese Spezifikation wird von der Eclipse Foundation *[4]* gehostet und umfasst eine Sammlung von Einzelspezifikationen. Dabei bedient sie sich aus existierenden Spezifikationen des klassischen Java EE beziehungsweise Jakarta EE-

Abbildung 2: Cloud-native Java Frameworks, eine Übersicht (Quelle: Sven Bernhardt)

Umfelds. Anders als bei Java Enterprise existiert bei MicroProfile keine Referenzimplementierung. Das macht den Spezifikationsprozess leichtgewichtig und ermöglicht kurze Release-Zyklen.

Umfang und Ökosystem sind bei den MicroProfile-Frameworks bei Weitem nicht so ausgeprägt wie bei Springboot. Dieser Umstand kann für bestimmte Anwendungsfälle Einschränkungen bedeuten, macht sie aber intuitiver und erleichtert Neulingen den Einstieg.

Neben den MicroProfile-Frameworks zeigt *Abbildung 2* auch sogenannte Microframeworks. Microframeworks sind charakterisiert durch eine reaktive, nicht-blockierende („non-blocking") Architektur. Sie sind optimiert auf schnelle Startup- und Verarbeitungszeiten. Frameworks dieser Kategorie verzichten meist auf jegliche Form impliziter „Framework Magic" wie Dependency Injection. Damit übernehmen Entwickler dann allerdings selbst die Verantwortung für bestimmte Abläufe, wie beispielsweise die korrekte Initialisierung von Klassen- und Objektnetzen oder die Freigabe bestimmter Ressourcen. Abhängig vom Anwendungsszenario erhöhen sich damit einerseits die Menge des Boilerplate Code und der Testaufwand. Auf der anderen Seite sind diese Frameworks durch wenige externe Abhängigkeiten jedoch extrem leichtgewichtig und flexibel.

Helidon Framework-Architektur

Bei Helidon handelt es sich um eine Sammlung von Bibliotheken für die Implementierung Java-basierter Microservices, deren Entwicklung maßgeblich durch Oracle getrieben wird. Das Open Source Framework wurde 2018 in der Version 1.0 released. Interessant dabei ist, dass das Framework, wie in *Abbildung 2* dargestellt, zwei unterschiedliche Varianten unterstützt: Helidon MP und Helidon SE. Basis für beide Framework-Varianten ist Netty *[5]*, ein asynchrones, Event-getriebenes Framework.

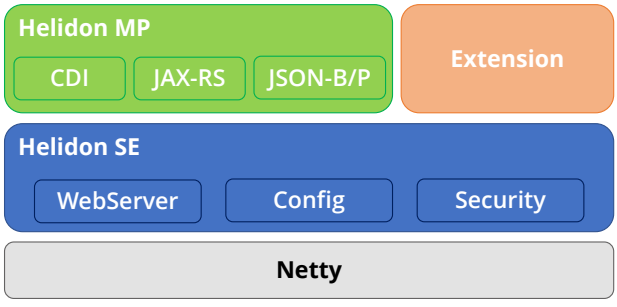

Abbildung 3: Architektur des Helidon Frameworks (Quelle: Sven Bernhardt)

Bei **Helidon SE** handelt es sich um ein leichtgewichtiges, reaktives Microframework, das ein funktionales Programmiermodell unterstützt. Wie in *Abbildung 3* dargestellt beinhaltet das Framework einen reaktiven Webserver sowie Features für ein flexibles Konfigurationsmanagement und Security.

Der reaktive Webserver zeichnet sich durch ein einfaches, funktionales Routing- Modell aus und bietet unter anderem Support für Open Tracing, Metrics sowie Healthchecks. Für die Implementierung von RESTful-Services werden zudem JAX-RS (REST-Service Support) und JSON-P (JSON Parsing) unterstützt. Da Cloud-native Applikationen dem Mantra: „Build once, run anywhere" aus den Twelve-Factor-App-Prinzipien *[1]* folgen, wird ein flexibler Konfigurationsmechanismus angeboten, der unterschiedliche Formate wie YAML oder Property-Files und auch dynamische Konfigurationsanpassungen zur Laufzeit unterstützt. Des Weiteren werden Funktionalitäten für die Umsetzung von Security-Anforderungen in den Bereichen Inbound- und Outbound-Authentifizierung, Autorisierung sowie Auditing bereitgestellt.

Bei **Helidon MP** handelt es sich um eine MicroProfile-Implementierung. Wie *Abbildung 3* zeigt, setzt Helidon MP auf die im vorherigen Abschnitt erläuterten Basisbausteine von Helidon SE auf, was eine effiziente Weiterentwicklung der beiden

Framework-Varianten ermöglicht. Der MicroProfile-Spezifikation folgend werden die Java EE-Spezifikationen für Context and Dependecy Injection (CDI), JAX-RS, JSON-B und JSON-P unterstützt.

Darüber hinaus ist Helidon MP über CDI-Extensions *[6]* flexibel erweiterbar. Derzeit existieren etwa CDI-Extensions für Java Persistence API (JPA) oder Java Transaction API (JTA). Beide Spezifikationen sind nicht Bestandteil der MicroProfile- Spezifikation, werden aber bei der Entwicklung von datenbankzentrischen Services benötigt; zusätzlich existieren weitere Extensions für den Zugriff auf Oracle Cloud Resources, wie z.B. Storage.

Aus Entwicklersicht ist der Ansatz von Helidon hochspannend: Entwickler können sich je nach Use Case für eine Variante entscheiden. Das Grundframework bleibt unverändert, einzig das Programmiermodell ändert sich. Das macht die Entwicklung effizienter und bietet mehr Flexibilität bei der Umsetzung von Business-Anforderungen.

Quickstart in die Helidon-Entwicklung

Der initiale Start in die Microservice-Entwicklung mit Helidon ist einfach: Für die Erzeugung eines initialen „Hello World"-Projekts als Startpunkt für die weitere Entwicklung können entsprechend vorhandene Maven-Archetypen für Helidon SE und Helidon MP verwendet werden.

```
mvn archetype:generate -DinteractiveMode=false \
    -DarchetypeGroupId=io.helidon.archetypes \
    -DarchetypeArtifactId=helidon-quickstart-mp \
    -DarchetypeVersion=1.4.1 \
    -DgroupId=com.opitzconsulting.helidon \
    -DartifactId=greeting-service-mp \
    -Dpackage=com.opitzconsulting.helidon
```

Listing 1: Projekt-Setup für einen Helidon Service (Helidon MP)

Der Maven-Aufruf in *Listing 1* erzeugt eine Verzeichnisstruktur für einen Helidon MP Service, die neben den üblichen Artefakten wie einer `pom.xml` und einer initialen Java-Projektstruktur sowie den erforderlichen Konfigurationsdateien (beispielsweise `logging.properties`) ein Dockerfile sowie ein Kubernetes-Deployment-Manifest enthält. *Abbildung 4* zeigt dies sehr detailliert. Analog zu dem in *Listing 1* dargestellten Aufruf zur Erzeugung eines Helidon MP-Projekts, wird ein Helidon SE-Projekt unter Verwendung des SE-spezifischen Maven-Archetyps durchgeführt.

Wie *Abbildung 4* zeigt, sind die resultierenden Projekte für Helidon SE und MP vom Aufbau her sehr ähnlich. Zur Ausführungszeit funktionieren beide Services sogar exakt gleich: Sie bieten eine HTTP-Ressource an, die einen „Hello World"-String zurückzuliefert. Die Implementierung hingegen unterscheidet sich aufgrund der Programmiermodelle.

Unterschiedliche Programmiermodelle

Helidon SE unterstützt ein funktionales, Helidon MP ein deklaratives Programmiermodell. Was dies konkret bedeutet, wird im Folgenden anhand von Beispielen erläutert. Der vollständige Code des in diesem Artikel verwendeten Beispiels findet sich in *[7]*.

Helidon-Applikationen haben immer eine zentrale Klasse, die sich für das Applikations-Bootstrapping verantwortlich zeichnet. In der Projektstruktur, die in *Abbildung 4* gezeigt wird, ist dies die Klasse `Main.java`. Hier erfolgen unter anderem die Runtime-Konfiguration und der Start

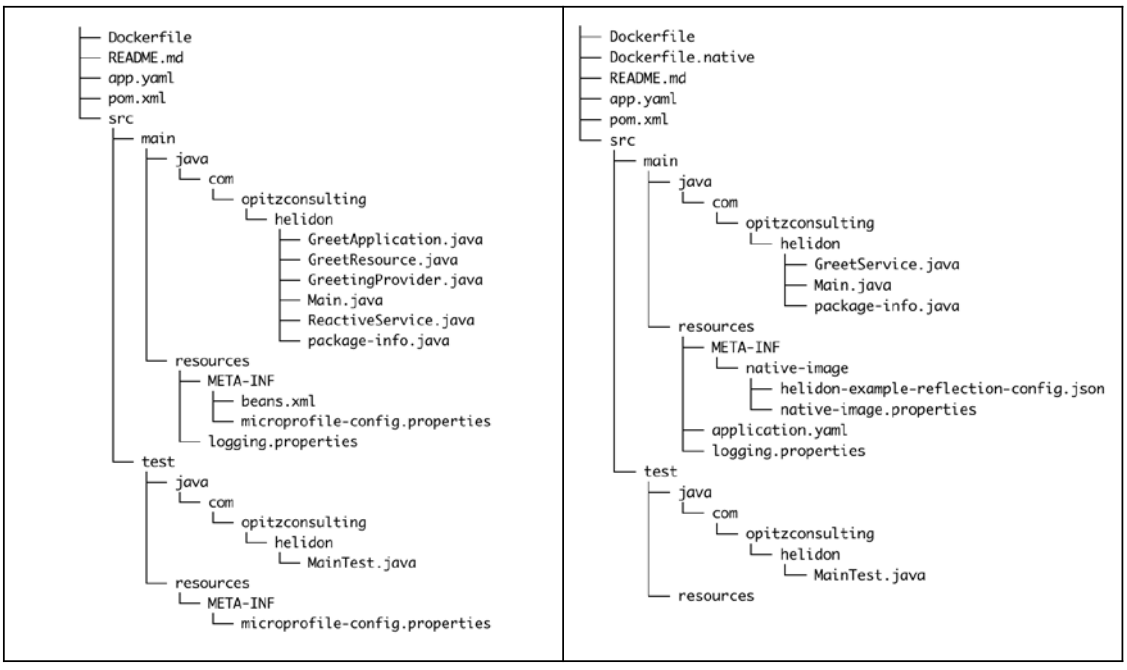

Abbildung 4: Projektstruktur Helidon MP (links) und Helidon SE (rechts) (Quelle: Sven Bernhardt)

```
static WebServer startServer() throws IOException {
        Config config = Config.create();
        ServerConfiguration serverConfig =
                ServerConfiguration.create(config.get("server"));
        WebServer server = WebServer.create(serverConfig, createRouting(config));
        ...
}

private static Routing createRouting(Config config) {

        MetricsSupport metrics = MetricsSupport.create();
        GreetService greetService = new GreetService(config);
        HealthSupport health = HealthSupport.builder()
                .addLiveness(HealthChecks.healthChecks())
                .build();

        return Routing.builder()
                .register(JsonSupport.create())
                .register(health)
                .register(metrics)
                .register("/greet", greetService)
                .build();
}
```

Listing 2: Auszug aus dem Applikations-Bootstrapping einer Helidon SE-Applikation

der Applikation. Für Helidon MP sieht diese Klasse übersichtlich aus, da große Teile des Bootstrapping implizit durch das Framework erledigt werden. Hierzu zählen unter anderem das Laden der Server-Konfiguration aus der Datei `microprofile-config.properties` sowie die Registrierung von Healthcheck- und Metrics-Endpunkten, über welche das Servicemonitoring erfolgt. Da in Helidon SE-Applikationen wie erwähnt weitestgehend auf Framework-Magic verzichtet wird, müssen solche Dinge explizit im Code deklariert werden.

Listing 2 zeigt, wie eine Helidon SE-Applikation konfiguriert wird. Die wichtigste Komponente in diesem Zusammenhang ist die Routing-Komponente, die im *Listing 2* in der Methode `createRouting` initialisiert wird. In dieser Methode werden der „Hello World"-Endpunkt sowie die Monitoring-Endpunkte der Applikation initialisiert und entsprechend registriert. Auch die Definition von Request Routings muss explizit ausimplementiert werden. Zu diesem Zweck werden in der Routing-Komponente entsprechende Routing-Regeln und zugehörige Request-Handler definiert, wie *Listing 3* zeigt.

Die Registrierung von Routing-Regeln und Request-Handlern erfolgt in der update-Methode, die jeder Helidon SE REST-Serviceendpunkt implementieren muss.

In Helidon MP erfolgt die Definition von Request Routings über die aus dem

```
public class GreetService implements Service {
    ...
    @Override
    public void update(Routing.Rules rules) {
        rules
            .get("/", this::getDefaultMessageHandler);
    }

    private void getDefaultMessageHandler(ServerRequest request,
                                          ServerResponse response) {
        sendResponse(response, "World");
    }
    ...
}
```

Listing 3: Routing-Regeln in Helidon SE

Java EE-Umfeld bekannten JAX-RS-Mechanismen. Hierbei werden die entsprechenden Annotationen (zum Beispiel `@GET`, `@ApplicationPath`, `@Path`) verwendet, um HTTP-Methoden-Mappings, Applikationspfade und anderes zu definieren.

Und es gibt noch weitere Unterschiede: Sollen beispielsweise zusätzliche Framework Features wie Tracing zur Erhöhung der Observability aktiviert werden, muss in einer Helidon SE eine entsprechende Maven-Dependency hinzugefügt werden. Die Verwendung einer Tracing-Komponente, wie sie in *Listing 4* gezeigt wird, muss dafür explizit im Code implementiert werden.

In einer Helidon MP-Applikation genügt es, die entsprechende Maven-Dependency für Tracing in der `pom.xml` zu ergänzen und die Applikations-Konfiguration entsprechend anzupassen.

Wie die Beispiele verdeutlichen, stellt Helidon MP viele Funktionen implizit bereit. Das muss nicht immer von Vorteil sein: Die implizite Bereitstellung kann unter anderem Fehleranalysen erschweren, wenn die Grundannahmen des Frameworks nicht erfüllt werden. Das Umbenennen der Datei `microprofile-config.properties` ist dafür ein gutes Beispiel.

Helidon und GraalVM

Die Java Virtual Machine (JVM) gilt als Standard-Laufzeitumgebung für Helidon-Applikationen. Diese eignet jedoch nicht gleichermaßen für alle Einsatzszenari-

```
static WebServer startServer() throws IOException {
        ...
    final Tracer appTracer = TracerBuilder.create("helidon-se").build();

        ServerConfiguration serverConfig = ServerConfiguration.builder(config.get("server"))
                .tracer(appTracer).build();
        ...
}
```

Listing 4: Open Tracing in Helidon SE-Applikationen

en im Cloud-native-Umfeld. Aus diesem Grund bietet Helidon auch Support für GraalVM an. *[8]*

Bei GraalVM handelt es sich um eine von Oracle entwickelte hochperformante, flexible Virtual Machine, die als Laufzeitumgebung für verschiedene Non-JVM-Sprachen (C/C++, JavaScript etc.) sowie für JVM-Sprachen (Java, Scala etc.) genutzt werden kann. GraalVM bietet einen Native Image Support, über den mithilfe des GraalVM Ahead-of-Time Compilers ein spezielles, für die Zielplattform compiliertes Image erzeugt werden kann. Dieses kann schließlich nativ auf der Zielplattform ausgeführt werden. Dieses native Image enthält alle Abhängigkeiten und macht die Installation einer Laufzeitumgebung überflüssig. Im Ergebnis führt dies zu kleineren Container-Images. Auch die Startup-Zeiten einer Applikation können so extrem gesenkt werden, da keine Laufzeitumgebung gestartet werden muss.

GraalVM unterstützt nicht alle JVM Features vollständig. So ergeben sich bei der Verwendung spezieller JVM-Features, wie zum Beispiel Java Reflection, Einschränkungen bei der Benutzung der Virtual Machine (für eine vollständige Auflistung siehe *[9]*). Aus diesem Grund wird GraalVM aktuell nur für Helidon SE- Applikationen supportet. Bei der Erzeugung eines Helidon SE-Projekts wird, wie in *Abbildung 4* zu sehen ist, auch ein `Dockerfile.native` angelegt, was für den Build eines GraalVM Native-Image-basierten Containers verwendet werden kann.

Fazit

Wie der Artikel zeigt, ist Helidon ein spannendes Framework und eine ernsthafte Alternative zu Springboot, wenn es um die Entwicklung Cloud-nativer Applikationen geht. Vor allem die Unterstützung der beiden unterschiedlichen Framework-Varianten macht Helidon interessant, eröffnet dies doch mehr Spielraum bei der Lösungsfindung für spezifische Anforderungen.

Allerdings existieren derzeit viele Open Source Frameworks, die einen speziellen Fokus auf die Entwicklung Java-basierter Microservices haben. Stärkste Konkurrenten neben Springboot sind Quarkus und Micronaut. Es darf also mit Spannung beobachtet werden, wie sich Helidon gegen die Konkurrenz behauptet. Das Framework verfolgt eine großartige Vision, die Releases erfolgen sehr regelmäßig und die Community ist aktiv und wachsend. Beste Voraussetzungen also, um die ersten Schritte mit Helidon zu wagen!

Quellen / Zusatzmaterialien

[1] Quentin Hardy (2019): The problem with „Cloud Native“, *https://bit.ly/36APh5S*, zuletzt abgerufen am 14.01.2020

[2] Adam Wiggins (2017): The Twelve-Factor App, *https://12factor.net/*, zuletzt abgerufen am 14.01.2020

[3] Pivotal Software (2020): Spring Boot 2.2.2, *https://spring.io/projects/spring-boot*, zuletzt abgerufen am 14.01.2020

[4] Eclipse Foundation (2020): Eclipse MicroProfile, *https://microprofile.io/*, zuletzt abgerufen am 14.01.2020

[5] Netty Project (2019): Netty Project, *https://netty.io/*, zuletzt abgerufen am 14.01.2020

[6] Antoine Sabot-Durand, et al.: JSR 365: Contexts and Dependency Injection for Java 2.0, *https://docs.jboss.org/cdi/spec/2.0/cdi-spec.html#spi*, zuletzt abgerufen am 14.01.2020

[7] Sven Bernhardt Github, *https://github.com/svenbernhardt/helidon-greeting-service*, zuletzt abgerufen am 14.01.2020

[8] Oracle (2018, 2019): GraalVM, *https://www.graalvm.org/*, zuletzt abgerufen am 14.01.2020

[9] Oracle GitHub (2019): Native Image Java Limitations, *https://github.com/oracle/graal/blob/master/substratevm/LIMITATIONS.md*, zuletzt abgerufen am 14.01.2020

Über den Autor

Sven Bernhardt arbeitet als Senior Solution Architect für die Opitz Consulting-Deutschland GmbH. Er konzipiert und implementiert zukunftsorientierte, robuste Anwendungen. Sven Bernhardt arbeitet in verschiedenen Projekten, die in den Bereichen Cloud, Microservices und API-Management angesiedelt sind. In seiner Rolle ist er an der Entwicklung und Konzeption von Best Practices in Bezug auf moderne Lösungsarchitekturen (*http://omesa.io*) beteiligt. Weiterhin ist Sven Bernhardt Oracle ACE Director und als Referent auf verschiedenen IT-Konferenzen aktiv.

Sven Bernhardt
sven.bernhardt@opitz-consulting.com