auf, dass eine LED defekt ist. Was tun? Es wurde ein kleines Testprogramm erdacht, das auf Knopfdruck alle LEDs aufleuchten lässt. Die LEDs waren ok, also musste es ein Programmfehler sein. Daraufhin haben Jakob und Daniel debuggt und den Fehler gefunden.

Henry und Henri – elektronischer Würfel: Man drückt abwechselnd Knopf B, um einen neuen Wert zu erhalten. Wer eine „6" würfelt, sieht und hört eine Belohnung.

Moritz und Kit – Pacman 2018: Man benutzt alle vier Eingabemöglichkeiten, damit sich der Pacman-Punkt über das Spielfeld bewegt: Knopf A nach links, Knopf B nach rechts, A+B nach oben, Schütteln nach unten, aber Vorsicht am Rand!

Johann und Eric – Verschlüsselung light: In bester Forschermanier wurden erst mal die Zahlenräume getestet und eine möglichst große Multiplikation gerechnet. Dann eine fast endlose Zeichenkette ausgeben – wird der Calliope das aushalten? Und es musste eine Verschlüsselung her, um den Code zu schützen: Erst eine bestimmte Kombination von Knopf A und B gibt den gewünschten Text aus. Als Variante entstand noch ein „Keylogger", der die Tastendrücke grafisch ausgab.

Weitere Informationen unter *„https://www.kids4it.de/calliope-workshop".*

Tino Sperlich
tino.sperlich@gmx.de

Tino Sperlich arbeitet als Software-Entwickler und Requirements Engineer in Hamburg. Er unterstützt regelmäßig Kids4IT, eine gemeinnützige User Group, und veranstaltet monatliche Workshops in der Hansestadt, um Schülern die Faszination von Technik allgemein und Programmierung im Speziellen zu vermitteln.

Microservices mit dem Helidon Framework

Marcel Amende, Oracle Deutschland B.V. & Co. KG

Von moderner Java-Entwicklung wird heute viel Agilität verlangt. Man will schnell und einfach neue Projekte starten, diese von der ersten Minute an automatisiert testen und früh in Produktion bringen. Es braucht die richtigen Frameworks, damit man sich auf die funktionalen Anforderungen konzentrieren kann, ohne Abstriche bei Codequalität und -leistung zu machen. Das Spring-Ökosystem [1] mit dem im Jahr 2014 veröffentlichten Spring-Boot-Projekt [2] ist sicherlich ein Pionier der modularen Entwicklung in Java. Es eignet sich gut für die Entwicklung von Microservices, wobei es in Bezug auf Funktionsumfang und Größe am oberen Ende des Werkzeugspektrums liegt. Mit dem Open-Source-Projekt Helidon [3] gibt es nun eine leichtgewichtige Alternative, die insbesondere Enterprise-Java-Entwicklern einen leichten Einstieg ermöglicht.

Klein ist fein

Neue technische Möglichkeiten bringen Veränderung. Die Veränderungsgeschwindigkeit nimmt zudem immer weiter zu. Auch die „Kunst" des Programmierens ist dadurch im steten Fluss. Java ist in dieser Hinsicht sicherlich keine Experimentierplattform, versucht allerdings konsequent, Trends aufzugreifen und diese zu professionalisieren, sobald sie sich bewähren. Für die Entwicklung von mehrschichtigen, monolithischen Webapplikationen ist mit Java EE eine beeindruckende Plattform entstanden. Heute ist jedoch die Entwicklung von (Cloud-) Applikationen in Form von kleinen, modularen und ereignisgetriebenen Diensten, sogenannten Microservices, gefragt. Die funktional reichhaltigen, aber schwergewichtigen Applikationsserver-Plattformen verlieren vor diesem Hintergrund trotz aller Optimierungen an Attraktivität. Zumal sich auch die Art der Ausbringung verändert. Statt eine reine Java-Applikation in ein Applikationsserver-Cluster auszubringen und alle weiteren Ressourcen zu referenzieren, packt man in Microservice-Architekturen in sich geschlossene Funktionalität mit allen benötigten Abhängigkeiten, Bibliotheken und Ressourcen in unabhängig lauffähige (Docker *[4]* -) Container. Diese werden wiederum auf einer verwalteten Containerplattform, zum Beispiel mit Kubernetes *[5]*, betrieben.

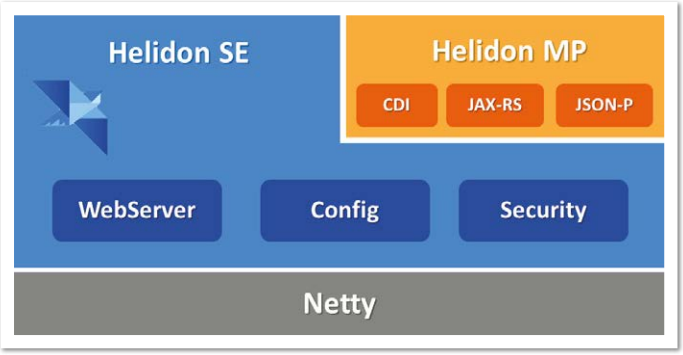

Abbildung 1: Helidon-Architektur

Dadurch ändern sich die Anforderungen an die Java-Plattform und an die Java Frameworks. Bei Ausbreitung in Hunderte Funktionen und Container ist die Größe und Startzeit der virtuellen Maschine entscheidender als das Deployment-Modell. Die Antwort aus Sicht von Java EE auf die neuen Anforderungen ist die Verschlankung zum MicroProfile, bei dem alle für die Entwicklung von Microservices irrelevanten Bibliotheken herausgestrichen werden. Letztlich ist aber auch eine Beschränkung auf Java SE und somit ein Verzicht auf sämtliche Pack- und Ausbringungsfunktionalität absolut möglich.

Helidon-Einführung

Das Open Source Framework Helidon ist eine Sammlung von Java-Bibliotheken, die das Schreiben von Microservices vereinfacht. Im Kern nutzt es das Netty Framework *[6]*, das speziell für die schnelle und einfache Erstellung von asynchronen, ereignisgetriebenen Netzwerkapplikationen geschaffen ist. Durch seine Nutzung der Java NIO (non-blocking IO) APIs werden bei Netzwerkoperationen keine Threads blockiert, was einen deutlich besseren Durchsatz und eine geringere Latenz bei gleichzeitig minimiertem Ressourcenbedarf ermöglicht. Helidon stellt auf dieser Basis einen reaktiven Webserver bereit, bei dem ein einzelner Thread mehrere (TCP-) Verbindungen gleichzeitig abhandeln kann. Grundlage dieser Implementierung ist zudem das Reactive Pattern *[7]*, das die Registrierung der Event Handler für die ereignisbasierte Verarbeitung der Serviceanfragen übernimmt. Dadurch wird letztlich eine gute vertikale Skalierung innerhalb einer virtuellen Maschine erreicht.

Helidon unterstützt zudem das Java MicroProfile. Weithin bekannte APIs wie JAX-RS für die Erstellung von RESTful Web Services, JSON-P für das Parsen von JSON-Nachrichten und Dependency Injection (CDI) sind nutzbar.

Mit Helidon erstellte Microservices werden als Docker-Container gepackt. Die Nutzung in Kombination mit einer, zum Beispiel durch Kubernetes, verwalteten Containerplattform bietet sich daher an. Helidon bietet darüber hinaus Integrationsmöglichkeiten beispielsweise mit dem Prometheus Framework für die Visualisierung operationaler Metriken oder mit dem Zipkin Framework für Nachverfolgbarkeit und Latenzanalysen in verteilten Microservice-Umgebungen.

Programmiermodell der Wahl

Bei Helidon kann man sich zwischen zwei Programmiermodellen entscheiden: Man spricht von Helion SE bei Verwendung des MicroFramework oder von Helidon MP bei Verwendung des MicroProfile. Das MicroFramework folgt einem rein funktionalen Programmierstil, bei dem man die von Helidon zur Verfügung gestellten APIs für WebServer, Konfigurationsverwaltung und Sicherheitsfunktionalität direkt nutzt. *Listing 1* zeigt ein Beispiel. Diese Variante bietet volle Transparenz und Kontrolle.

Für Entwickler, die auf ihre Erfahrung mit Enterprise Java (JEE) zurückgreifen möchten, bietet sich das MicroProfile mit seinem eher deklarativen Programmiermodell an. Hier werden Annotationen genutzt, um aus einfachen Java-Klassen Services und aus den Methoden der Klasse Service-Endpunkte zu machen. *Listing 2* zeigt die Deklaration eines einfachen REST-Service.

Schnelleinstieg in Helidon

Der Einstieg in Helidon gelingt schnell und mit wenigen Voraussetzungen. Man braucht am Entwicklerarbeitsplatz als Minimalvoraussetzung nur Java SE 8 oder Open JDK 8 als Laufzeitumgebung, Maven (ab Version 3.5) für den Buildprozess und Docker (ab Version 18.02) als Container für die Ausbringung des Microservice. Für den Fall, dass ein durch Kubernetes verwaltetes Containercluster Ziel der Ausbringung ist, benötigt man zusätzlich Kubectl (ab Version 1.7.4) für den administrativen Zugriff vom Client aus.

Bei der händischen Erstellung eines RESTful-Service mit Helidon beginnt man mit der Erstellung eines Maven-Projekts in einer Java-IDE der Wahl oder durch die Ausführung eines Standard-Maven-Archetyps auf Kommandozeilenebene *(siehe Listing 3)*.

Das dabei entstehende pom.xml muss um zwei Einträge ergänzt werden, um den Helidon-Web-Server und YAML als Helidon-Konfigurationsformat heranzuziehen *(siehe Listing 4)*.

Bei Verwendung einer IDE sollten die Main-Klasse und die Pfade abhängiger Bibliotheken automatisch eingefügt werden. Ansonsten

```
WebServer.create(
  Routing.builder()
     .get("/greet", (req, res)
        -> res.send("Hello MicroFramework!"))
     .build())
  .start();
```

Listing 1: Programmierbeispiel MicroFramework

```
public class GreetService {
  @GET
  @Path("/greet")
  public String getMsg() {
     return "Hello MicroProfile!";
  }
}
```

Listing 2: Programmierbeispiel MicroProfile

```
$ mvn archetype:generate
     -DarchetypeGroupId=org.apache.maven.archetypes
     -DarchetypeArtifactId=maven-archetype-quickstart
     -DarchetypeVersion=1.3
```

Listing 3

ist das detaillierte Vorgehen in der Sektion „Guides" der Helidon-Dokumentation beschrieben *[8]*.

Noch einfacher findet man den Einstieg, wenn man spezielle Helidon-„Quickstart"-Archetypen für Maven verwendet. Diese gibt es für beide Programmiermodelle. Für Helidon SE kann das in *Listing 5* gezeigte Kommando genutzt werden, wobei die Parameter „groupId", „artifactId" und „package" frei wählbar sind.

Dadurch wird ein kompilierbares und sofort lauffähiges Maven-Projekt generiert, das einen auf dem Helidon Framework basierenden RESTful-Service enthält. Zentraler Teil des Projekts ist eine ausführbare Klasse „Main.java", die den Web-Server startet und das Routing für den Service-Endpunkt erstellt *(siehe Listing 6)*.

Eine weitere Klasse „GreetService.java" stellt die auf dem Interface „io.helidon.webserver.Service" basierende Serviceimplementierung dar. Sie registriert sich über ein Update der Routingregeln und generiert für die verschiedenen Servicemethoden JSON-Ausgaben *(siehe Listing 7)*.

Den Build-Prozess für das Projekt stößt man über Maven mit $ mvn package an. Das Resultat ist eine Beispielapplikation in einem JAR-Archiv; alle zur Laufzeit benötigten abhängigen Bibliotheken werden in einem Unterverzeichnis „target/libs" gespeichert. Die Applikation kann durch Ausführen der JAR-Datei lokal gestartet und getestet werden: `$ java -jar target/quickstart-se.jar`

Die Applikation stellt einen RESTful-Service bereit, der exemplarisch einige GET- und POST-Methoden unterstützt, um Grußbotschaften in Form von JSON-Nachrichten auszugeben. Folgender Aufruf ist z.B. möglich: `$ curl -X GET http://localhost:8080/greet/Java%20Aktuell`
`{"message":"Hello Java Aktuell!"}`

Ein Microservice ist mit Helidon also in wenigen Sekunden erstellt: Beispielprojekt mit Maven aus einem Archetyp erstellen, packen und starten. Nun kann man mit der funktionalen Erweiterung und Anpassung fortfahren. Bleibt die Frage, wie man den Microservice auf einfache Art und Weise ausbringen und betreiben kann.

Helidon in der Cloud

Docker-Container sind heute typischerweise das Mittel der Wahl für das Packen und Ausbringen von Microservices, vor allem in privaten oder öffentlichen Cloud-Umgebungen. Dazu trägt bei, dass die Docker-Container mit Werkzeugen wie Kubernetes (K8s) auch in großer Anzahl und in großen Clustern effektiv verwaltet werden können.

An dieser Stelle werden wir die Oracle Container Native Services *[9]* der Oracle Cloud Infrastructure (OCI) heranziehen, um Helidon in einer Containerumgebung zu betrachten. Diese beinhalten eine verwaltete Kubernetes-Umgebung und eine Docker Registry. Mit einem kostenlosen Testzugang *[10]* lässt sich das Beispiel praktisch nachvollziehen. Lokale Installationen von Docker und Kubernetes sowie Docker Hub *[11]* und Cloud-Umgebungen vieler anderer Anbieter sollten ebenfalls nutzbar sein. Die in den folgenden Beispielen verwendeten Docker- und Kubernetes-Kommandos sind allgemeingültig. Nur die APIs für Zugang und Konfiguration sind anbieterspezifisch.

Der erste Schritt in Richtung der Ausbringung des Helidon-Beispielservice ist das Packen als Docker Image. Das mit dem Schnellstartbeispiel ausgelieferte Dockerfile sorgt dafür, dass Applikation und Blibliotheken in ein online frei beziehbares Basisimage mit bereits enthaltenem OpenJDK kopiert werden. `$ docker build -t quickstart-se target`

```
<dependencies>
     <dependency>
        <groupId>io.helidon.bundles</groupId>
        <artifactId>helidon-bundles-webserver</artifactId>
     </dependency>
     <dependency>
        <groupId>io.helidon.config</groupId>
        <artifactId>helidon-config-yaml</artifactId>
     </dependency>
</dependencies>
```

Listing 4

```
$ mvn archetype:generate
     -DinteractiveMode=false \
     -DarchetypeGroupId=io.helidon.archetypes \
     -DarchetypeArtifactId=helidon-quickstart-se \
     -DarchetypeVersion=0.11.0 \
     -DgroupId=io.helidon.examples.javaaktuell \
     -DartifactId=quickstart-se \
     -Dpackage=io.helidon.examples.javaaktuell.quick-
start.se
```

Listing 5

```
...
WebServer server =
     WebServer.create(serverConfig, createRouting());
server.start().thenAccept(ws -> {...});
...
```

Listing 6

```
...
@Override
public final void update(final Routing.Rules rules) {
  rules
     .get("/", this::getDefaultMessage)
     .get("/{name}", this::getMessage)
     .put("/greeting/{greeting}", this::updateGreeting);
}
...
```

Listing 7

```
$ docker images
REPOSITORY      TAG       IMAGE ID      CREATED      SIZE
quickstart-se   latest    1234567abcde  2 hours ago  88.6MB
```

Listing 8

```
$ kubectl create secret docker-registry <SecretName> \n
--docker-server=fra.ocir.io \n
--docker-username='/<TenantName>/<UserName>' \n
--docker-password=<AuthToken> \n
--docker-email='<AnyDummyEmail>'
```

Listing 9

Daraus resultiert ein ca. 88,6 MB großes und mit dem Tag „latest" versehenes Image namens „quickstart-se" im lokalen Docker Repository *(siehe Listing 8)*.

Dieses lässt sich bereits lokal ausführen und mit denselben Endpunkten und Aufrufen testen wie beim vorherigen Start der Applikation als JAR-Datei. `$ docker run --rm -p 8080:8080 quickstart-se:latest`

Die Reise ist hier allerdings noch nicht zu Ende, schließlich soll die Applikation in einer vollverwalteten Umgebung im Cluster bereitgestellt werden. Erster Schritt dafür ist die Anmeldung bei der entfernten Docker Registry von der Docker-Kommandozeile aus. In diesem Fall handelt es sich um eine private Oracle Cloud Infrastructure Registry (OCIR) in der Rechenzentrumsregion Frankfurt („fra.ocir.io"): `$ docker login fra.ocir.io -u <TenantName>/<UserName> -p <AuthToken>`

Im nächsten Schritt wird das lokale Image mit einem Tag für den folgenden Push in die Registry in der Cloud versehen: `$ docker tag quickstart-se:latest fra.ocir.io/<TenantName>/<RepoName>/quickstart-se:latest`

Nun kann der „Push", das heißt der Upload des Docker Image, in die Cloud Registry erfolgen: `$ docker push fra.ocir.io/<TenantName>/<RepoName>/quickstart-se:latest`

Da es sich um eine private Registry handelt, muss dem Verwaltungswerkzeug Kubernetes nun eine Authentifizierung durch ein sogenanntes „Pull Secret" ermöglicht werden, um Docker Images per „Pull" für die direkte Ausbringung heranzuziehen. Dieses „Pull Secret" kann über die Kubernetes-Kommandozeile generiert werden *(siehe Listing 9)*.

Das Secret wird als Datei im Home-Verzeichnis des Betriebssystembenutzers abgelegt.

Mit dem Maven-Projekt wurde bereits ein Deskriptor „app.yaml" für die Ausbringung mit Kubernetes generiert. Dieser wird nun angepasst. In der Deployment-Sektion unter `spec: containers:` wird der Link auf das Docker Image in der privaten Cloud Registry gesetzt: image: `fra.ocir.io/<TenantName>/<RepoName>/quickstart-se:latest`

Zusätzlich wird für die Authentifizierung noch einen Verweis auf das generierte „Pull Secret" ergänzt:

```
imagePullSecret:
- name: <SecretName>
```

In der Datei ließe sich auch die Anzahl der Replikate erhöhen oder man könnte Memory-Limits für den Service setzen. Grundsätzlich ist jedoch bereits ohne weitere Anpassungen alles für die Ausbringung über die Kubernetes-Kommandozeile („kubectl") bereitet:

```
$ kubectl create -f target/app.yaml
```

Ist der Kubernetes-Proxy gestartet, kann man sich das Ergebnis der Ausbringung im Kubernetes Dashboard ansehen. Dort sieht man ein Deployment „quickstart-se", das als Service auf einem Kubernetes-Knoten („Pod") läuft *(siehe Abbildung 2)*.

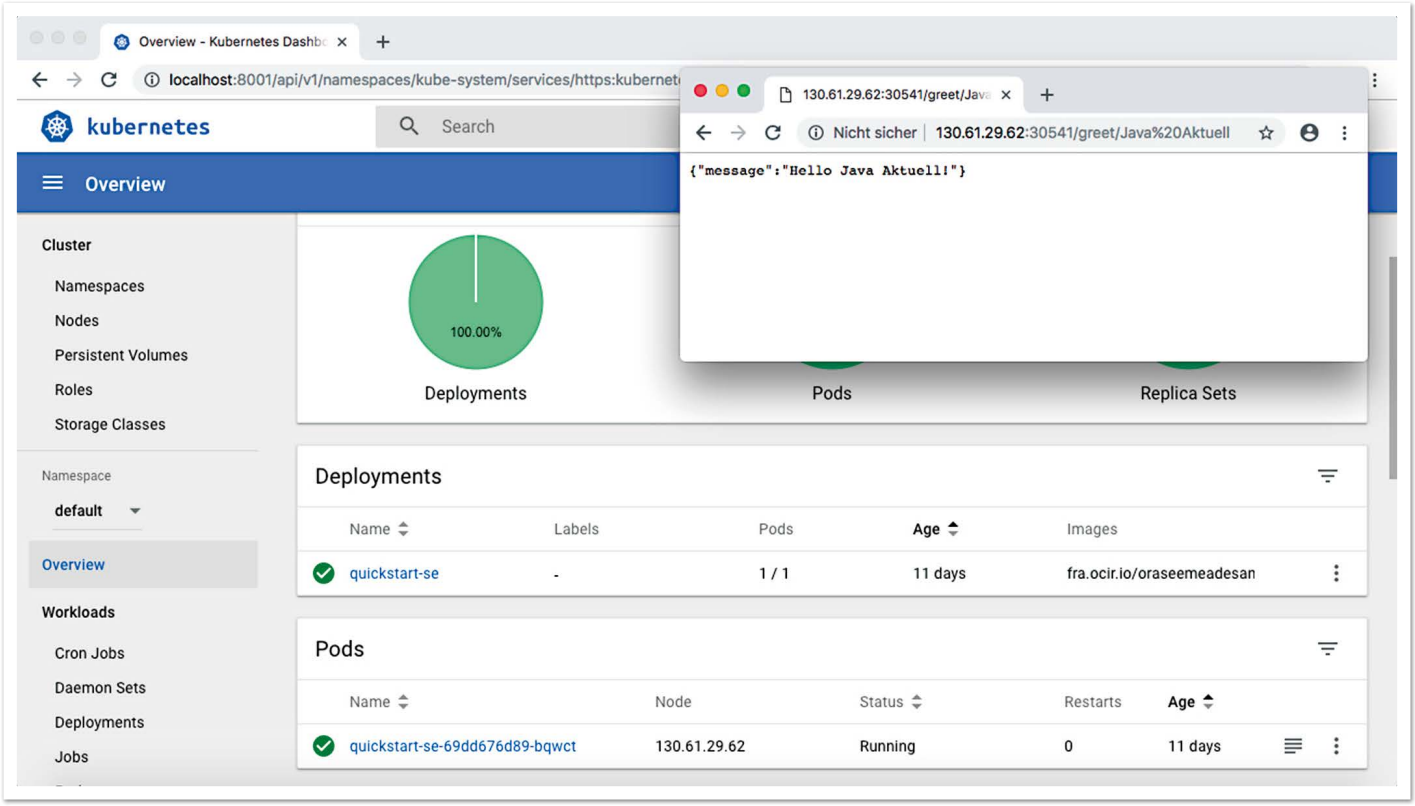

Abbildung 2: Kubernetes Dashboard - Overview

Fazit und Ausblick

Beim Entwickeln in und für die Cloud gehen die Veränderungen Hand in Hand. Vermehrte Anforderungen an Flexibilität und Agilität in der IT bereiten den Boden für neue Entwicklungsmethodiken wie die Mico-service-Entwicklung. Die Entwicklergemeinde schafft sich im dynamischen Open-Source-Umfeld die Werkzeuge selbst, um die anstehenden Entwickungsaufgaben zu vereinfachen. Mit Helidon reiht sich hier ein neues, schlankes und leicht zu erlernendes Framework ein, das auch von der Interoperabilität lebt. Denn in der Coud entstehen auf Basis von Docker, Kubernetes, Prometheus & Co. die passenden Ablauf-, Verwaltungs- und Überwachungsumgebungen für Microservices. Diese Art der Quasi-Standardisierung auf Basis von Open Source eröffnet dem Entwickler ungeahnte Möglichkeiten der Portabilität zwischen den Cloud-Anbietern. Was kann man von Helidon in Zukunft erwarten? Helidon stünde ein Eventsystem gut zu Gesicht, um aus einzelnen Microservices flexibel modulare Applikationen zu erstellen. Unterstützung für kommende Web-Standars wie HTTP/2 und MicroProfile-Versionen ist naheliegend. Zudem besteht großes Potenzial in einer engeren Integration mit dem JDK, aber auch zum Beispiel mit der Graal VM.

Referenzen

[1] *https://spring.io*
[2] *https://spring.io/projects/spring-boot*
[3] *https://helidon.io*
[4] *https://docker.com*
[5] *https://kubernetes.io*
[6] *https://netty.io/*
[7] *https://java-design-patterns.com/patterns/reactor/*
[8] *https://helidon.io/docs/latest/#/guides/01_SE_REST_web-service*
[9] *https://cloud.oracle.com/containers*
[10] *http://cloud.oracle.com/tryit*

Marcel Amende
Marcel.Amende@oracle.com

Geboren als Ingenieur, aufgewachsen bei Oracle, zu Hause in der Cloud. Als Solution Engineer repräsentiert Marcel die „Cloud Native"-Entwicklergemeinde von Oracle, die aus den Diensten der Oracle Cloud kreative Kundenlösungen gestaltet. Neben der Serviceentwicklung und Integration gilt sein besonderes Interesse auch dem Internet der Dinge.