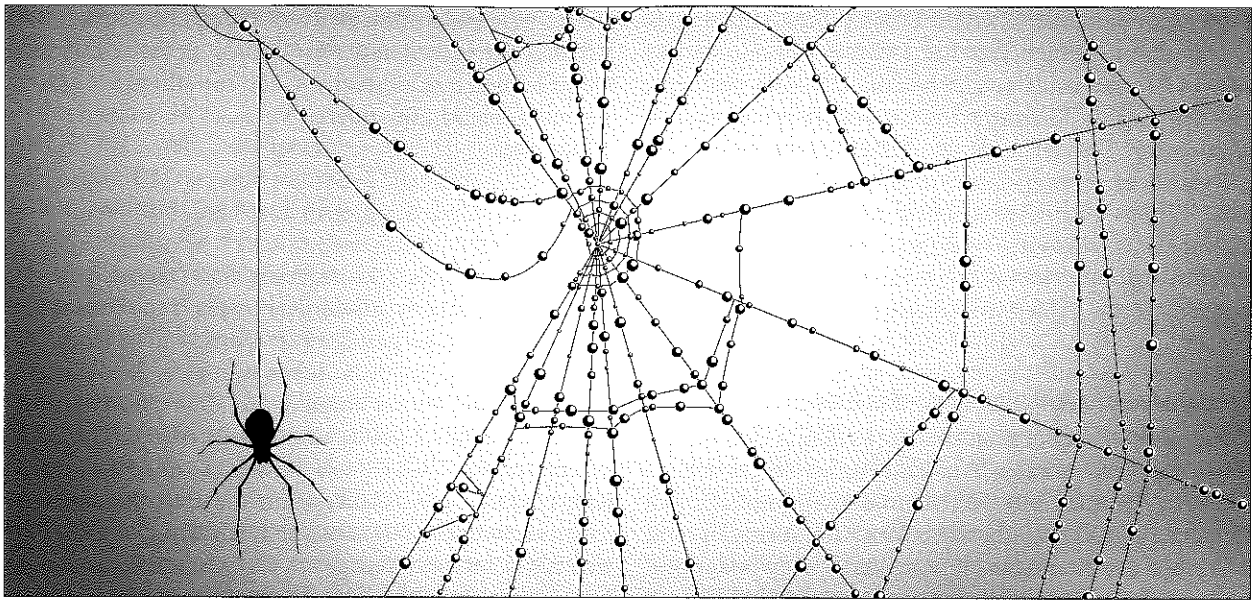

Vert.x für Microservices

Eine reaktive Lösung

Microservices stellen uns Entwickler vor organisatorische Herausforderungen und zwingen uns zur Beschäftigung mit dem Thema verteilte Systeme. Genau für diese verteilten Systeme wurde die neue Generation reaktiver Anwendungsframeworks konzipiert. Vert.x hat sich hier in den letzten Jahren einen guten Namen gemacht und sich als eine beliebte Alternative zu den klassischen Frameworks etabliert.

von Jochen Mader

Microservices haben die Art, in der wir Anwendungen bauen, stärker verändert als jeder andere Trend zuvor. Die Änderungen ziehen eine breite Schneise durch die gesamte IT. Anforderungsmanagement, Teamstruktur, Operations, Infrastruktur und eingesetzte Frameworks haben sich in den letzten Jahren auf diesen Architekturstil zubewegt. Die Erwartungen an die eingesetzten Frameworks sind dabei hoch:

- elastische Skalierbarkeit
- gute Testbarkeit
- im Framework eingebaute Resilienz
- ausgereifte Monitoringunterstützung
- kompaktes Deployment

Einer der interessantesten Kandidaten bei der Auswahl eines passenden Microservices-Frameworks ist Vert.x. An dieser Stelle möchte ich auch gleich meinen Vert.x-Elevator-Pitch loswerden: Vert.x ist ein reaktives, modulares, nachrichtenorientiertes, polyglottes Framework, mit dem sich performante Microservices-Architekturen aber auch klassische Anwendungen umsetzen lassen. Im Jahre 2011 von Tim Fox aus der Taufe gehoben, hat Vert.x mittlerweile drei Major-Versionen, eine gut funktionierende Community und mit Red Hat einen starken Corporate-Sponsor vorzuweisen. Im folgenden Artikel werde ich zeigen, was Vert.x so interessant macht und weshalb es sich ausgezeichnet für Microservices eignet.

Micro!!!!111

Der kleinste Vert.x-Microservice besteht – inklusive Imports und Java-Boilerplate – aus 224 Zeichen. Ohne den Java-Pflichtanteil sind es sogar nur 103 Zeichen (Listing 1).

Fehlt noch das Deployment. Das Fat JAR hat sich für JVM-basierte Microservices als Deployment-Ar-

tefakt durchgesetzt [1]. Alle Vert.x Build Blueprints liefern das Erzeugen von Fat JARs direkt mit. In der Maven-Variante [2] reicht hierfür ein *mvn package*, für den Gradle Blueprint [3] verwendet man *gradlew shadowJar*. Danach liegt im Build-Ordner eine Datei mit der Bezeichnung *<AppName>-fat.JAR*. Diese kann man mit *java -jar <PathToFatJar>/<AppName>-fat.jar* ausführen, direkt deployen (z. B. mit Nomad [4] von HashiCorp) oder klassisch in einen Docker-Container packen. Fertig ist der vollständige „Hello World"-Microservice.

Verticles sind Kernbausteine jeder Anwendung

Dieses erste Beispiel zeigt die Verwendung des Vert.x-API als Serverbaukasten. Auf diese Art lassen sich schnell kleine Server schreiben. Für Microservices empfiehlt sich hingegen die Verticle-basierte Variante. Verticles sind seit jeher der Kernbaustein von Vert.x-Anwendungen. Sie sind die wichtigste Organisations- und Skalierungseinheit und unterliegen einigen wichtigen Regeln. Jedes Verticle

- leitet von *AbstractVerticle* ab.
- wird nie von mehr als einem Thread gleichzeitig verwendet.
- wird immer von dem gleichen Thread bedient.
- besitzt keinen mit anderen Verticles geteiltet Zustand.
- darf keinen blockierenden Code enthalten.
- kommuniziert mit anderen Verticles über einen EventBus.

Das kleine Codebeispiel in Listing 2 hilft beim Verständnis.

Das *MainVerticle* koordiniert das Deployment der Anwendung. Zuerst initialisiert es das Deployment von zwei *RestVerticles* gefolgt vom Deployment eines *DatabaseVerticles*. Somit lauschen nun zwei *RestVerticles* an Port 8000. Vert.x sorgt dafür, dass sich eingehende Requests gleichmäßig auf diese verteilen. Greift ein Client auf den URL */user/1* zu, schickt das *RestVerticle* eine entsprechende Anfrage an die EventBus-Adresse „loadusername". Für diese hat sich das *DatabaseVerticle* als Empfänger registriert. Es nimmt die *userId* entgegen, lädt den passenden Usernamen und schickt ihn an den Anfrager zurück. Das *RestVerticle* nimmt die Antwort entgegen, schreibt sie in die noch offene *Response* und schließt den Request damit ab.

Wie bereits erwähnt, können Verticles nur über den EventBus miteinander Daten austauschen. Sie sind somit extrem lose über eine Adresse und das Format der übermittelten Nachrichten miteinander gekoppelt. Der EventBus unterstützt drei Varianten zur Übermittlung von Nachrichten:

- *vertx.eventbus().send(<address>,<payload>)*: Peer to Peer, existiert der Empfänger nicht, wird die Nachricht ohne Rückmeldung verworfen.
- *vertx.eventbus().send(<address>,<payload>,<callback>)*: Request Reply, es wird eine Rückantwort vom Empfänger erwartet. Existiert kein Empfänger, wird der Callback darüber benachrichtigt. Zusätzlich kann ein Time-out für die Antwort angegeben werden. Das ist sehr zu empfehlen!
- *vertx.eventbus().publish(<address>,<payload>)*: Sendet an alle unter der Zieladresse registrierten Empfänger. Existiert kein Empfänger, wird die Nachricht ohne Rückmeldung verworfen.

Grundsätzlich liefert Vert.x keine Garantien für die Zustellung von Nachrichten. Das lässt sich nur durch die Verwendung von Variante 2 (Send mit Reply) erreichen.

C10K

Bis heute setzen viele Webserver auf das so genannte One-Thread-Per-Request-Paradigma. Bei diesem lauscht ein Thread am Server-Socket. Er akzeptiert eingehende Verbindungen, delegiert ihre Verarbeitung jedoch an andere Threads. Dies ist notwendig, weil in klassischen *java.io.Sockets* eine Operation beliebig lang blockieren kann. Wurde z. B. eine Verbindung aufgebaut und der Client stirbt direkt danach, muss der Thread bis zum Auslaufen des TCP/IP-Time-outs warten. Auch das Füllen des zugehörigen Puffers kann beliebig lang dauern und ist stark von der verfügbaren Bandbreite abhängig. Bei blockierenden Aufrufen müssen wir also dafür sorgen, dass diese in separaten Threads passieren, um responsive zu bleiben. In den letzten Jahren ist die Zahl der Requests, die ein Server beantworten muss, drastisch gestiegen. Mit einer wachsenden Zahl an Threads fällt eine oft vergessene Operation immer mehr ins Gewicht: der Thread Context Switch. Viele Threads teilen sich eine CPU. Jedes Mal, wenn ein Thread ausgeführt werden soll, verdrängt er einen anderen von der CPU. Diese Operation ist vergleichsweise billig, aber eben nicht umsonst. Mit der Zeit steigen diese Kosten bis zu einem Punkt, an dem die CPU mehr Zeit mit dem Context Switch als mit der eigentlichen Arbeit verbringt. Dieses Problem ist unter der Bezeichnung C10K (10 000 Concurrent Connections) bekannt. Im NIO-Fall kooperiert die JVM mit dem Betriebssystem. Mithilfe von Selektoren wird ermittelt, aus welchen geöffneten Verbindungen ohne blockieren gelesen oder geschrieben werden kann. Somit kann ein einzelner Thread mehrere 10 000 Anfragen gleichzeitig bearbeiten.

Listing 1: Ein kleiner Vert.x-Microservice

```
import io.vertx.core.Vertx;

public class HttpServer {
  public static void main(String[] args) {
    Vertx.vertx().createHttpServer().requestHandler(req -> req.response().
                                    end("Hello World")).listen(8000);
  }
}
```

Das Format der zu übermittelnden Nachrichten unterliegt außerdem einigen Beschränkungen. Folgende Typen sind erlaubt:

- *JsonObject* oder *JsonArray*
- Primitive Datentypen: int, float, String etc.
- Vert.x-Buffer-Objekte
- Nachrichten, für die ein eigener Codec registriert wurde (Kryo, Thrift, Avro etc.)
- Klassen, die Serializable implementieren, können nicht verschickt werden

Da alle Kommunikation zwischen den Bestandteilen einer Vert.x-Anwendung über den EventBus erfolgt, ist er auch der perfekte Ort, um Load Balancing umzusetzen. Existieren für die gleiche EventBus-Adresse mehrere Empfänger, wird automatisch ein Non-Weighted-Round-Robin-Ansatz verwendet, um die Nachrichten zu verteilen. Wird z. B. das *DatabaseVerticle* zum Flaschenhals, können wir dies einfach mehrfach deployen. Nachrichten werden nun weitestgehend gleichmäßig auf die beiden Instanzen verteilt.

Bleibt noch zu klären, wie ein HTTP-Server und eine Datenbankschnittstelle mit so wenigen Threads auskommen. Vert.x setzt vollständig auf Non Blocking IO. Statt wie im klassischen IO blockierend aus einer Datei oder einem Socket zu lesen, können wir seit JDK 1.4 die Klassen aus *java.nio* verwenden. Dies ermöglicht es, mehrere 10 000 Netzwerkverbindungen in einem Thread abzuarbeiten statt eine entsprechende Anzahl spawnen zu müssen (warum das nicht so gut ist, siehe Kasten „C10K"). Die Verwendung der Java-NIO-Klassen ist nicht ganz einfach, weshalb Vert.x unter der Haube auf Netty [5] als NIO-Layer zurückgreift. Mehr Details zu NIO gibt es in diversen Einsteigertutorials [6].

Elastizität: Verteilte Systeme als Default

Ob aus Gründen der Skalierung oder um die Resilienz zu erhöhen: Irgendwann erreichen wir den Punkt, an dem eine Maschine nicht mehr ausreicht. An den Aufbau eines Clusters gehen viele Entwickler mit solcher Vorsicht heran, als würde man sich einem schlafenden Bären nähern. Verteilte Systeme gelten nicht grundlos als eine der großen Herausforderung in der Softwareentwicklung. Wer nicht weiß, wovon ich rede, der sollte sich umgehend mit den Fallacies of Distributed Computing [7] vertraut machen. Im Gegensatz zu vielen anderen Frameworks betrachtet Vert.x die Verteilung als den Default-Fall, denn das gesamte Framework wurde als Basis für verteilte Systeme konzipiert. Das wichtigste Basiselement für den Aufbau einer verteilten Anwendung ist der EventBus.

Anhand der Liste möglicher Nachrichtenformate ist vielleicht aufgefallen, dass der Bus nur einfach zu serialisierende Objekte als Payload zulässt. Der Hintergrund hierfür ist, dass der EventBus nicht nur innerhalb einer JVM, sondern auch zur Verbindung mehrerer Instanzen zum Einsatz kommt. Für den Verwender ist es vollkommen irrelevant, ob eine Adresse in der lokalen Instanz oder auf einem anderen Rechner liegt. Um den EventBus über mehrere Maschinen auszudehnen, braucht es natürlich ein klein wenig Infrastruktur, die Vert.x aber bereits vollständig mitbringt.

Um den Cluster aufbauen zu können, sind zwei Dinge notwendig: Eine *ClusterManager*-Implementierung muss im Klassenpfad verfügbar sein (z. B. das *vertx-hazelcast*-Modul, das die Hazelcast-basierte Default-Variante einbindet) und das Fat JAR muss mit dem Parameter *-cluster* gestartet werden.

Nehmen wir das Beispiel von vorhin und verpacken sowohl das *RestVerticle* als auch das *DatabaseVerticle* in eigene Fat JARs. In unserem hypothetischen Cluster existieren drei Maschinen. Auf zweien werden sowohl das *RestVerticle-fat.jar* als auch das *DatabaseVerticle-fat.jar* mit der Option *-cluster* gestartet. Auf der dritten wird nur das *DatabaseVerticle-fat.jar* gestartet. Alle Beteiligten finden sich nun Dank des *ClusterManagers* gegenseitig. Eingehende HTTP-Requests werden über

Listing 2: Verticle

```
public class MainVerticle extends AbstractVerticle {
  @Override
  public void start() throws Exception {
    vertx.deployVerticle(RestVerticle.class.getName(), new DeploymentOptions().
                                                                    setInstances(2));
    vertx.deployVerticle(DatabaseVerticle.class.getName());
  });
}

public class RestVerticle extends AbstractVerticle {
  @Override
  public void start() throws Exception {
    Router router = Router.router(vertx);
    router.get("/user/:userId").handler(req -> {
      String userId = req.request().params().get("userId");
      vertx.eventBus().<String>send("loadusername", userId, response -> req.response().
                                                        end(response.result().body()));
    });
    vertx.createHttpServer().requestHandler(router::accept).listen(8000);
  }
}

public class DatabaseVerticle extends AbstractVerticle {

  @Override
  public void start() throws Exception {
    vertx.eventBus().<String>consumer("loadusername").handler(msg -> {
      String username = ...
      ...
      msg.reply(username);
    });
  }
}
```

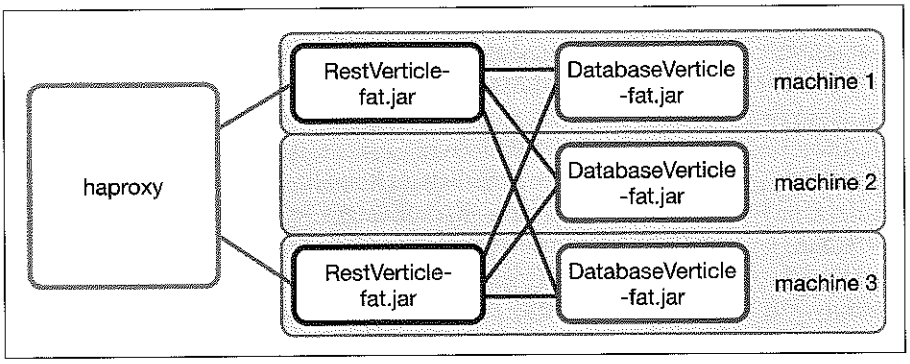

Abb. 1: Eingehende HTTP-Requests werden über HAProxy an die REST-Server verteilt; um die Verteilung der Zugriffe auf die drei „DatabaseVerticles" kümmert sich Vert.x über ein Round-Robin-System

HAProxy an unsere beiden REST-Server verteilt. Um die Verteilung der Zugriffe auf die drei *DatabaseVerticles* kümmert sich Vert.x mit dem bereits erwähnten Round Robin.

Die Aufgabe des *ClusterManagers* ist es, die Informationen über registrierte EventBus-Adressen zwischen allen registrierten Knoten zu synchronisieren. Er sorgt so auch dafür, dass neue Knoten bekannt gemacht und verschwundene Knoten entfernt werden. Auf diesem Weg kann eine Vert.x-Anwendung elastisch auf Lastveränderungen und Ausfälle reagieren. Neben den üblichen Aufgaben der Clusterverwaltung bietet der *ClusterManager* noch einige grundlegende verteilte Datenstrukturen: Distributed MultiMap, Distributed Lock und Distributed Counter. Mit diesen lassen sich auch komplexe Anforderungen ohne viel Aufwand umsetzen.

Die Hazelcast-Variante bietet sich für die meisten Szenarien an, da sie die notwendige Infrastruktur über einen embedded Hazelcast abbildet. Sollte aber bereits andere Infrastruktur verfügbar sein oder andere Gründe für eine alternative Implementierung vorliegen, gibt es auch die Möglichkeit, eigene Implementierungen zu verwenden. Außerdem gibt es auch bereits drei fertige Alternativen in Form von Modulen: JGroups, Apache Ignite und ZooKeeper.

Metriken: wissen, wie es dem Cluster geht

Metriken sind die wichtigste Quelle für Informationen über den Zustand eines Clusters. Sie ermöglichen es, sowohl das Sizing eines Clusters an die gegebene Last anzupassen als auch die Identifikation von Problemen. Für Vert.x lässt sich das Ermitteln von Metriken durch das Setzen einer der System-Properties *vertx.metrics.options.enabled=true* oder *vertx.metrics.options.jmxEnabled=true* und das Einbinden der Dependency *io.vertx:vertx-dropwizard-metrics:<vertx_version>* aktivieren. Ab diesem Zeitpunkt sammelt Vert.x die Zahl der verwendeten Threads, den EventBus-Durchsatz pro Adresse, die EventBus-Statistiken sowie die HTTP-Statistiken zu Durchsatz und die Zahl der gelieferten HTTP-Codes.

Bei diesem Ansatz wird davon ausgegangen, dass ein externer Prozess die Daten via JMX oder mithilfe von Jolokia über HTTP pullt. Alternativ hierzu gibt es auch noch eine Integration mit Hawkular [8]. Im Modul *vertx-hawkular-metrics* befindet sich zum einen alles, um die Metriken zu ermitteln, zum anderen auch ein Mechanismus, um die Werte zu einem Hawkular-Server zu übertragen.

Security einfach integrieren

Im Bereich Authentifizierung und Autorisierung wird es wohl nie eine One-Size-fits-all-Lösung geben, weshalb moderne Frameworks eine saubere Integration verschiedener Varianten bieten müssen. Im Fall von Vert.x gibt es aktuell fünf Varianten in Form von Auth-Provider-Implementierungen: JDBC Auth, JWT Auth, MongoDB Auth, OAuth 2 und Shiro Auth. Alle diese Varianten ermöglichen sowohl Authentifizierung als auch rollenbasierte Autorisierung. Zusätzlich zu dieser schon recht umfangreichen Liste lassen sich eigene Verfahren durch die Implementierung des Vert.x-Common-Auth-API umsetzen.

Integration, Datenbanken und HTTP 2

Auch wenn wir gerne auf einer grünen Wiese starten, sind es doch die Migrationsprojekte, in denen Microservices gerade ihre größten Erfolge feiern. Alte Monolithen werden nie durch einen so genannten Big Bang abgelöst, sondern werden schrittweise zerlegt und soweit wie möglich durch neue Microservices ersetzt. Diese müssen natürlich mit den Resten des Monolithen oder anderen bestehenden Middlewarestrukturen kommunizieren können. Hier punktet Vert.x mit einem gut funktionierenden JCA-Adapter und einer vollständigen Camel-Anbindung.

Auch im Bereich Datenbankadapter tut sich einiges. Neben den offiziellen Adaptern für JDBC, MongoDB, Redis, MySQL und PostgreSQL lassen sich auch HBase, Cassandra und Neo4j über diverse Communitymodule anbinden. Im Zweifelsfall ist man genau eine E-Mail auf der Mailingliste von der Anbindung der Datenbank der Wahl entfernt.

Ein paar grundsätzliche Tipps

- Vermeide die Callback Hell durch die Verwendung von *CompositeFuture*.
- Verwende RxJava, alle Vert.x-Module sind auch als Rx-ified-Versionen verfügbar.
- Alle Calls in Vert.x sind asynchron und geben dir die Möglichkeit, einen Callback zu registrieren, um über den Erfolg einer Operation informiert zu werden. Nutze sie!
- Wenn du eine Nachricht schickst und an der Antwort interessiert bist, verwende immer einen Time-out.

Die wichtigste anstehende Änderung findet aber mit dem nächsten Update von Vert.x statt: die Integration von HTTP 2. Die lang ersehnte neue Version des HTTP-Standards wird aktuell in den Kern von Vert.x aufgenommen und sollte mit dem nächsten Release verfügbar sein.

So viel mehr

Es ist wie immer schwer, in so wenigen Zeichen alles unterzubringen, was ein Framework wie Vert.x mitbringt, und man muss einiges auslassen. Ein paar Dinge, die ich aber zumindest noch erwähnt haben möchte, gibt es dann doch noch: Vert.x ist polyglott. Derzeit gibt es sprachspezifische APIs für Java, JavaScript, Ceylon, Groovy und Ruby. Es gibt eine integrierte, erweiterbare Telnet-/SSH-Shell, und alle APIs der Kernmodule sind auch in einer RX-Version verfügbar. Frameworks wie Akka lassen sich über Reactive Streams anbinden. Und es gibt natürlich ungezählte Module aus der Community. Noch viel wichtiger: Die Dokumentation von Vert.x ist detailliert, vollständig und liefert viele Codebeispiele, die direkt ausprobiert werden können.

Schöne neue Welt

Microservices verändern die IT, und das tiefgreifender als jeder andere Architekturstil zuvor. Reaktive Frameworks wie Vert.x bilden hier die perfekte Basis für eine neue Generation verteilter Systeme. Gleichzeitig muss man allerdings bei aller möglichen Geschwindigkeit im Auge behalten, dass die Entwicklung der eigentlichen Microservices nur einen kleinen Teil der Herausforderungen ausmacht, vor denen wir stehen. Agilität, Automatisierung und DevOps sind wichtige Grundbausteine, auf denen erfolgreiche IT aufbaut. Ohne sie wird auch das beste Framework keine bleibende Veränderung im Unternehmen hinterlassen können.

Jochen Mader ist Lead IT Consultant bei der codecentric AG, Trainer, regelmäßiger Speaker auf Konferenzen, Autor diverser Fachartikel und generell an allem interessiert, was Softwareentwicklung spannend macht.

@codepitbull

Links & Literatur

[1] Mader, Jochen: „Java JARs have curves": https://blog.codecentric.de/en/2015/06/real-jars-have-curves/

[2] Einfaches Vert.x-Maven-Projekt: https://github.com/vert-x3/vertx-examples/blob/master/maven-simplest

[3] Einfaches Vert.x.-Gradle-Projekt: https://github.com/vert-x3/vertx-examples/tree/master/gradle-simplest

[4] Dadgar, Armon: „Nomad": https://www.hashicorp.com/blog/nomad.html

[5] Netty: http://netty.io/

[6] Jenkov, Jacob: „Java NIO Tutorial": http://tutorials.jenkov.com/java-nio/index.html

[7] Fallacies of Distributed Computing: https://de.wikipedia.org/wiki/Fallacies_of_Distributed_Computing

[8] Hawkular: http://www.hawkular.org/

Feature	Vert.x
Minimale Anwendung	Eine Java-Klasse und ein Build-Skript (z. B. Maven, Gradle)
REST-Unterstützung	Ja
REST-APIs	Notwendige Funktionen werden durch Vert.x-Web bereitgestellt.
Messaging-Unterstützung	Ja
Messaging-APIs	Vert.x-EventBus, AMQP, RabbitMQ plus diverse Communitymodule
Fat JAR Deployment	Ja
Build-Tools	Maven, Gradle
Konfiguration	JSON-Dateien im JAR oder auf dem Dateisystem und Umgebungsvariablen
Metriken	Ja
Protokolle	Dropwizard Metrics, Hawkular, JMX, HTTP
Sicherheit	Authentifizierung und Autorisierung basierend auf verschiedenen Backends: JDBC Auth, JWT Auth, MongoDB Auth, OAuth 2, Shiro Auth
Health-Check	Nein
Protokolle/Daten	/
Stärken	- sehr gute Performancecharakteristik - sehr gute Skalierungseigenschaften, sowohl horizontal als auch vertikal - starke Community - viele Erweiterungen verfügbar - sehr umfangreiche Dokumentation
Resilience	- im Framework durch den *ClusterManager* umgesetzt - Integration von Hystrix für synchrone Calls möglich

Tabelle 1: Überblick über die Vert.x-Features