Java aktuell

iJUG Verbund
www.ijug.eu

Coding Continuous Delivery
Hilfreiche Werkzeuge für die Jenkins-Pipeline

Besser Programmieren
Die Entwicklungsumgebung optimal nutzen

Aus der Praxis
Microservices richtig einsetzen

Java und mehr

Ausgabe 03/2018
Java aktuell
Praxis. Wissen. Networking. Das Magazin für Entwickler
D: 4,90 EUR | CH: 9,80 CHF | A: 5,60 EUR | Benelux: 5,80 EUR
4 191978 304903 03

DOAG 2018
Logistik + IT

14. Juni 2018 in Köln

logistik.doag.org

iJUG Verbund

Werden Sie Mitglied im iJUG!

20% Rabatt auf JavaLand -Tickets

Ab 10,- EUR im Jahr erhalten Sie ein Jahres-Abonnement der Java aktuell

Mitglied im **J**ava **C**ommunity **P**rocess

www.ijug.eu

Save the Date

20. - 23. Nov 2018

in Nürnberg

2018.doag.org

Eventpartner: AOUG AUSTRIAN ORACLE USER GROUP · SOUG swiss oracle user group · iJUG Verbund · ORACLE

Reaktive Programmierung mit Java und Spring

Torsten Kohn, Comsysto Reply GmbH

Reaktive Programmierung gewinnt in der Entwicklung von Backend-Systemen immer mehr an Interesse. Der Artikel bietet einen Einblick und zeigt, wie man mit Java und Spring eine reaktive Anwendung schreibt.

Bei der Entwicklung von Software gibt es unterschiedliche Vorgehensweisen und Schwerpunkte. So kann man synchron und blockierend entwickeln *(siehe Abbildung 1)*. Bei diesem Beispiel führt der „main"-Thread eine lang andauernde I/O-Operation aus, was zur Folge hat, dass die Anwendung während der Ausführung der I/O-Operation blockiert ist und auf deren Ergebnis wartet. Der Code dazu dürfte für einen Entwickler einfach umzusetzen sein. Eine Verbesserung ist die Auslagerung der Operationen in neue Threads *(siehe Abbildung 2)*.

In diesem Fall werden neue Threads für die Operationen erzeugt und die Ergebnisse am Ende zusammengeführt. Das ist effektiver, wenn mehrere Operationen notwendig sind, um das Ergebnis zu erzeugen. Schlussendlich wird der „main"-Thread blockiert, um die Ergebnisse zu erhalten. Das Erzeugen dieser Threads ist kostenintensiv, das Warten sowie Zusammenführen kompliziert.

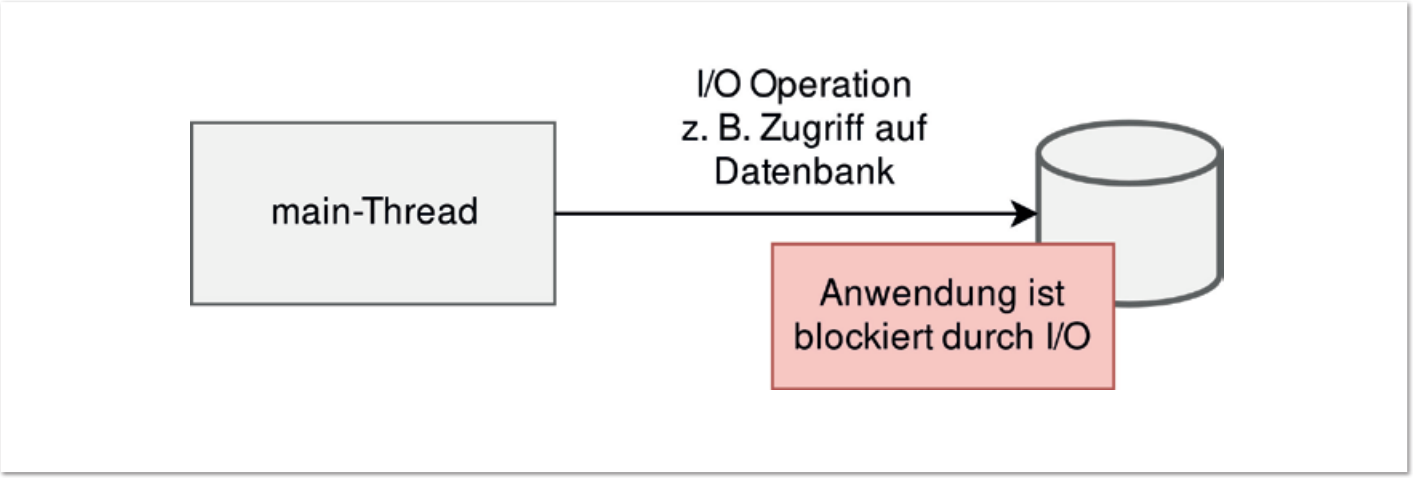

Abbildung 1: „main"-Thread wird durch I/O blockiert

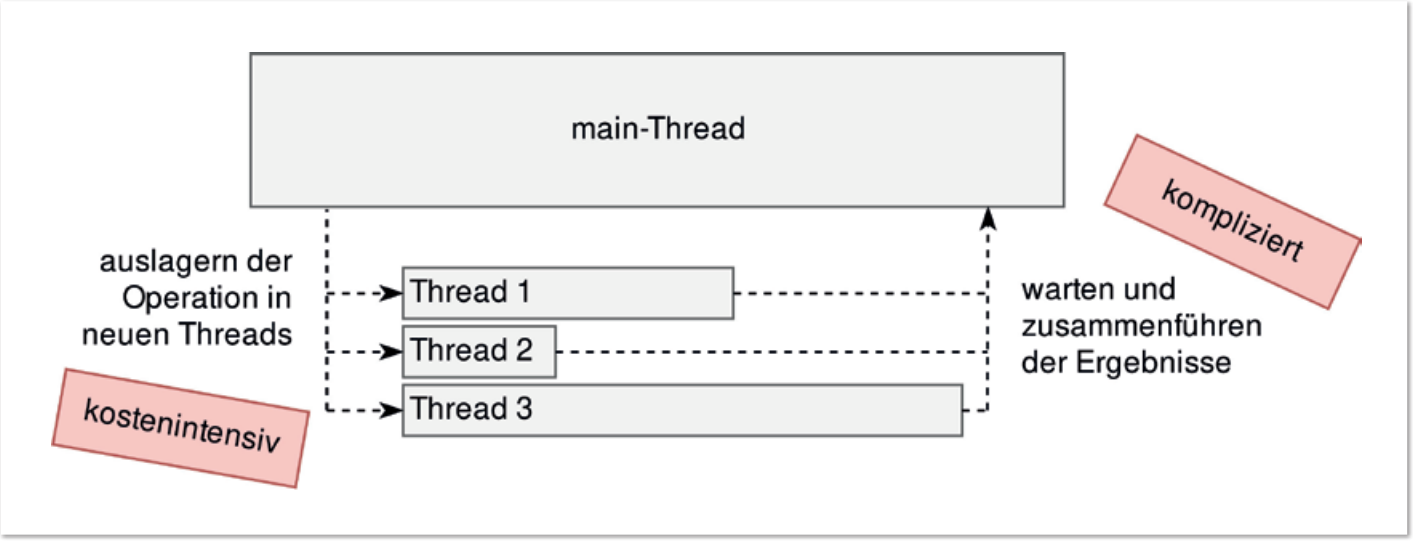

Abbildung 2: Operationen in neue Threads auslagern: asynchron, aber blockierend

Beide gezeigten Ansätze blockieren die Anwendung. Was man hingegen erreichen möchte, ist asynchrone, nicht blockierende Verarbeitung. Daher folgt der dritte Ansatz mit einer Ereignisschleife (Event-Loop, *siehe Abbildung 3*).

Dabei arbeitet ein Thread mehrere Anfragen ab und kann die Verarbeitung dieser Anfragen unterbrechen. Dies führt zu einer verbesserten Ressourcen-Nutzung, da kein Thread im Hintergrund auf Ergebnisse wartet. Es werden somit nur so viele Threads verwendet, wie auch benötigt werden. Implementieren kann man diesen Ansatz in Java unterschiedlich, etwa mit Callbacks, was schnell zu einer Callback-Hölle führt, sodass man den Überblick über die aufgerufenen Funktionen verliert.

Darüber hinaus kann man in Java „Future" (mit Java 8 „CompletableFuture") verwenden, damit die Anwendung asynchron agiert, ohne Callbacks zu verwenden. Ein großer Nachteil ist, dass dies Ergebnis-basiert funktioniert. Das bedeutet, dass beim Erzeugen eines „Compl etableFuture<List<String>>" die gesamte Liste aufgebaut wird (kein Streaming möglich) und man dadurch den Vorteil der asynchronen Verarbeitung verliert. Hier kommt nun reaktive Programmierung ins Spiel, um die Anwendung asynchron und nicht blockierend zu programmieren; und das alles mit Unterstützung des Streaming-API.

Reaktive Programmierung

Reaktive Programmierung (Reactive Programming) ist ein Programmier-Paradigma. Es fokussiert sich auf Datenströme und das Propagieren von Änderungen. Ziel ist es, asynchrone, eventbasierte und nicht blockierende Operationen/Komponenten zu entwickeln. Dieses Vorgehen verhindert Ressourcen-Verschwendung durch auf I/O-Operationen wartende Threads.

Ein reaktives Beispiel-Programm ist Excel: Ändert man den Wert in einer Zelle, dann ändert sich auch der Wert in der Zelle mit der Berechnung. Microsoft hat mit der Reactive-Extentions-Bibliothek (Rx) im .NET-Ökosystem den Ansatz der reaktiven Programmierung gestartet. Die Bibliothek „RxJava" hat diesen Ansatz auf die JVM gebracht, wodurch reaktive Programmierung Einzug in Java hielt. Zwischenzeitlich hat sich eine Initiative gegründet, die eine Spezifikation bereitstellt, um einen Standard für das Verarbeiten von asynchronen Datenströmen mit nicht blockierendem Code anzubieten.

Spezifikation und Manifest

Entwickler von Netflix, Pivotal, Twitter und weiteren Unternehmen arbeiten zusammen an einer Spezifikation für reaktive Programmierung, den sogenannten „Reactive Streams" *[1]*. Das Projekt stellt neben der Spezifikation ein Java-API, ein Technology-Compatibility-

Kit (TCK, zur Validierung der eigenen Implementierung gegen die Spezifikation) sowie Beispiel-Implementierungen bereit. Das Projekt ist unter der Creative-Commons-Zero-Lizenz *[2]* veröffentlicht. Dabei ist im Einzelnen zu prüfen, ob diese Lizenz im Projekt eingesetzt werden darf.

Interessant ist zudem das „reaktive Manifest" *[3]* von Jonas Bonér, Dave Farley, Roland Kuhn und Martin Thompson, das kurz und prägnant die Eigenschaften von reaktiven Systemen darstellt. Die Autoren definieren dazu vier reaktive Qualitäten, die in der Architektur der Softwarekomponente beachtet werden sollen, um reaktive Systeme/Komponenten kombinierbar zu machen:

- Das System ist „responsive" (antwortbereit) und antwortet zeitgerecht. Hier sollen konsistente Antwortzeiten eingehalten werden, damit man Fehler durch Ausbleiben von Antworten erkennen und behandeln kann.
- Bei Ausfällen bleibt das System „resilient" (widerstandsfähig) und ist responsive. Diese Widerstandsfähigkeit wird durch das Replizieren von Funktionalität, Abschirmung der Komponenten, Eindämmung von Fehlern und das Delegieren von Verantwortung erreicht.
- Durch die Widerstandsfähigkeit (siehe oben) bleibt die Komponente „elastic" (elastisch) und kann durch das Replizieren auf eine sich ändernde Last reagieren, weitere Ressourcen beanspruchen oder diese freigeben.
- Zur Sicherstellung der Entkopplung sowie Isolation werden zwischen den Komponenten asynchron Nachrichten übermittelt. Dies wird mit der vierten Qualität „message-driven" (nachrichtenorientiert) beschrieben. Durch die Reactive-Streams-Spezifikation und das reaktive Manifest können reaktive Systeme einheitlich designt werden, ohne das Rad immer wieder neu erfinden zu müssen. Dies fördert Kompatibilität und ermöglicht, Komponenten in Projekten wiederzuverwenden.

Interfaces

Die Spezifikation definiert vier Interfaces, siehe dazu *Listing 1*. Das Publisher-Interface ist die Quelle für die zu verarbeiten Elemente. Damit man diese Daten verarbeiten kann, muss man sich mit einem Subscriber registrieren. *Abbildung 4* zeigt den Zusammenhang

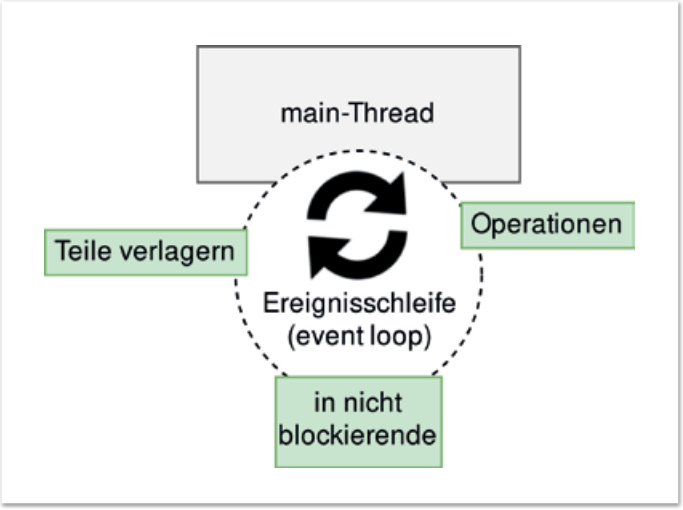

Abbildung 3: Verwendung einer Ereignisschleife zur Abarbeitung der Operationen (Operationen in nicht blockierende Teile verlagern)

zwischen den beiden Interfaces. Damit der Publisher Daten bereitstellt, muss sich mindestens ein Subscriber registrieren. Passiert dies nicht, werden keine Daten vom Publisher verarbeitet. Sobald ein Subscriber sich registriert hat, benachrichtigt der Publisher ihn über vorhandene Daten und sendet diese an den Subscriber.

Zwischen Publisher und Subscriber sitzt ein Subscription-Objekt *(siehe Listing 1)*. Es steuert den Datenfluss. Erhält der Subscriber zu viele Daten vom Publisher, kann er durch die Subscription die Menge reduzieren. Genauso kann der Subscriber die Menge der Daten erhöhen. Dies wird in diesem Zusammenhang „Backpressure" genannt und soll den Subscriber vor Überlastung schützen *(siehe Abbildung 4)*. Zusätzlich gibt es das Processor-Interface, bei dem sich die Implementierung zugleich als Publisher und Subscriber verhält.

Implementierungen für Java-Entwickler

RxJava wurde in der Einführung schon erwähnt als sogenannte „Reactive Extensions" für die JVM. Seit Version 2 verwendet RxJava das API der Reactive Streams und hält sich an die Spezifikation. Eine weitere Implementierung der Reactive-Streams-Interfaces bietet das Project „Reactor" *[4]* von Pivotal.

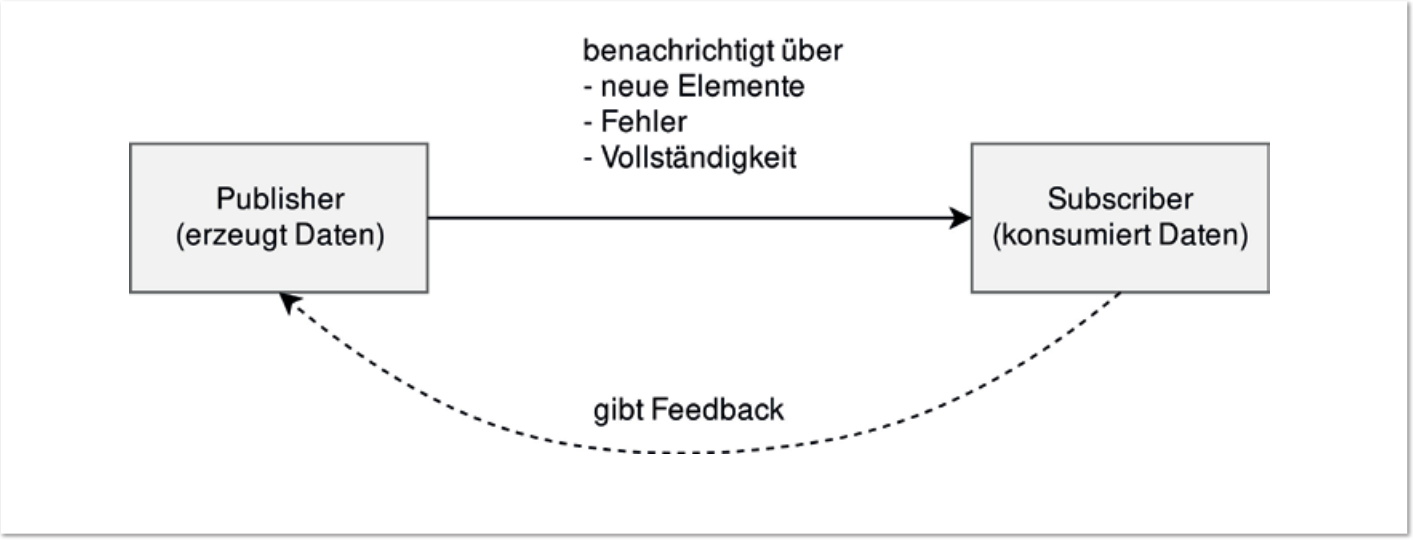

Abbildung 4: Zusammenhang zwischen Publisher und Subscriber

```
public interface Publisher<T> {
    public void subscribe(Subscriber<? super T> s);
}

public interface Subscriber<T> {
    public void onSubscribe(Subscription s);
    public void onNext(T t);
    public void onError(Throwable t);
    public void onComplete();
}

public interface Subscription {
    public void request(long n);
    public void cancel();
}

public interface Processor<T, R> extends
Subscriber<T>, Publisher<R> {
}
```

Listing 1: Interfaces aus der Reactive-Streams-Spezifikation

```
// Imports, Konstruktoren sowie GETTER und SETTER fehlen
zur Übersicht
@Document
public class Todo {
    @Id
    private String id;
    private String title;
    private boolean completed;
}
```

Listing 2: Modell Todo

```
public interface TodoRepository extends
ReactiveMongoRepository<Todo, String> {
}
```

Listing 3: Repository für Todo-Objekte

Beide Projekte bieten vorgefertigte Subscriber- und Publisher-Implementierungen, um die Entwicklung von reaktiven Applikationen zu erleichtern. Mit Java 9 sind die Interfaces in das Flow-API übernommen worden. Hier muss man als Entwickler die Interfaces selbst implementieren und sich um die Einhaltung der Spezifikation kümmern, da es keine vorgefertigten Implementierungen gibt.

Reaktiv mit Spring

Nachdem das Grundgerüst für reaktive Programmierung geschaffen ist, nun das Beispiel-Projekt mit Spring Boot. Spring bietet über die Webseite *„https://start.spring.io“* einen einfachen Projekt-Generator für Spring-Boot-Applikationen *(siehe Abbildung 5)*. Die Abhängigkeit „Reactive Web“ beinhaltet das Projekt „Reactor“, das als Implementierung für die reaktive Programmierung dient.

Das generierte Projekt ist der Einstieg für die nächsten Beispiele. Wichtig ist, dass man Version 2 oder höher bei Spring Boot verwendet sowie die Abhängigkeit zu Reactive Web und Reactive MongoDB hinzufügt, um in die Vorzüge der reaktiven Programmierung zu kommen.

Als Beispiel-Anwendung dient ein Todo-Webservice, der über REST angesprochen wird und die Daten in einer MongoDB speichert. Das Projekt zeigt auf, was Spring an Implementierungen für eine reaktive Anwendung bereitstellt und welche für das Beispiel genutzt werden. Das Projekt ist auf GitLab veröffentlicht *[5]* und im Artikel nur in Ausschnitten gezeigt. *Listing 2* zeigt das Modell „Todo“, ein einfaches POJO. Die Klasse verwendet keine speziellen Typen für die reaktive Programmierung und dient nur zur Datenhaltung. Die beiden Annotationen „@Document“ und „@Id“ dienen dem Datenbank-Zugriff.

Methoden wie „findAll(), save(S entity)“ oder „findById(ID id)“ werden vom Interface geerbt; in diesem Beispiel stehen das Generic „S“ für „Todo“ sowie „ID“ für „String“. Der Namenszusatz „Reactive“ lässt erahnen, dass sich das Repository anders verhalten wird. Man kann als Alternative vom Interface „MongoRepository“ erben, das dann keinen reaktiven Support bietet. Im Beispielprojekt wird das Repository durch einen Service gekapselt. Da keine Logik im Service vorhanden ist, wird der Service hier nicht vorgestellt.

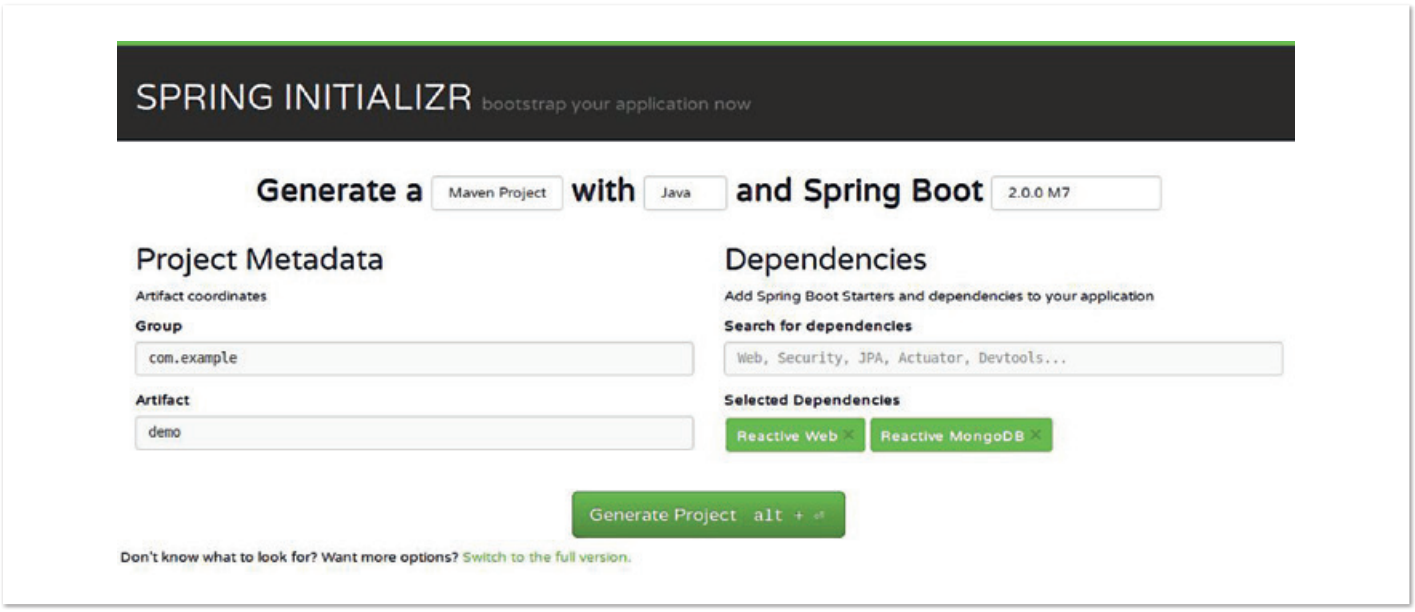

Abbildung 5: Spring-Boot-Projekt-Generator (siehe „https://start.spring.io“)

Listing 4 zeigt einen Ausschnitt des REST-Controllers mit den „GET"-Methoden. Die Annotationen „@RestController", „@RequestMapping" und „@GetMapping" dienen dazu, die Klasse zu markieren, damit Spring die Klasse als Controller erkennt und zusätzlich das Mapping für die verschiedenen REST-Methoden auflösen kann.

Die Annotationen sind Entwicklern bekannt, die mit Spring arbeiten. Neu hingegen sind die Rückgabewerte „Flux<Todo>" und „Mono<Todo>". Hierbei handelt es sich um Publisher-Implementierungen aus dem Project Reactor. Vom REST-Controller bis zum Repository werden die Typen „Flux" und „Mono" durchgängig verwendet. Flux ist ein Publisher, der 0 bis n Elemente zurückliefert, Mono hingegen ist ein Publisher, der nur 0 bis 1 Element liefert. Beide Klassen implementieren das Publisher-Interface von Reactive Streams. Durch Nutzung des „ReactiveMongoRepository<T, ID>"-Interface sowie die durchgängige Nutzung von Flux und Mono erhält der Entwickler eine reaktive Anwendung ohne großen Mehraufwand.

Testen des Controllers

Das Spring-Framework sowie Project Reactor bieten Test-Bibliotheken an, um reaktive Programmierung zu testen; *Listing 5* zeigt die zwei Abhängigkeiten, *Listing 6* einen Test des REST-Controllers mit dem „WebTestClient". Die Klasse ist speziell zum Testen von reaktiven Programmen, da es ein nicht blockierender reaktiver Client ist. Der Test überprüft „MediaType", HTTP-Statuscode und „RequestBody".

Fallstricke bei reaktiver Programmierung

Eine Anwendung besitzt meist eine Datenbank-Schnittstelle und hier bestehen Defizite zu den vorhandenen reaktiven Datenbank-Treibern. Zurzeit gibt es offiziell nur Implementierungen für MongoDB, Redis und Casandra. Möchte man dagegen JDBC oder JPA einsetzen, hat man blockierenden Zugriff auf die Datenbank. Ob es für JDBC und JPA eine reaktive Implementierung geben wird, wird sich zeigen.

Fazit

Der Artikel gibt eine Einführung in die reaktive Programmierung und zeigt, dass es mit wenig Aufwand möglich ist, die Vorteile zu nutzen. Die reaktive Programmierung ist allerdings nicht das Allheilmittel; es ist im Projekt zu überprüfen, ob die Rahmenbedingungen erfüllt sind – dies fängt bei der Auswahl des Datenbank-Systems an und endet beim Entwickler-Wissen.

Vorteile bietet die reaktive Programmierung durch die Zusammensetzbarkeit der Komponenten (siehe dazu das reaktive Manifest) und den effektiven Ressourcen-Einsatz. Die nächsten Monate werden zeigen, wie die Java-Community reaktive Programmierung aufnimmt und vor allem wie das Flow-API aus Java 9 eingesetzt wird.

Quellen

[1] *http://www.reactive-streams.org*
[2] *https://creativecommons.org/publicdomain/zero/1.0/deed.de*
[3] *https://www.reactivemanifesto.org/de*
[4] *https://projectreactor.io*
[5] *https://gitlab.com/torstenkohn/todo-list-backend*


```
@RestController
@RequestMapping("/api/todos")
public class TodoRestController {
    @Autowired
    private final TodoService service;
    @GetMapping
    public Flux<Todo> all() {
        return this.service.all();
    }

    @GetMapping("/{todoId}")
    public Mono<Todo> byId(@PathVariable String to-
doId) {
        return this.service.byId(todoId);
    }
}
```

Listing 4: REST-Controller

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>io.projectreactor</groupId>
    <artifactId>reactor-test</artifactId>
    <scope>test</scope>
</dependency>
```

Listing 5: Abhängigkeiten für Testing

```
@Test
public void testTodosForId() {
    WebTestClient client = WebTestClient
            .bindToController(new
TodoRestController(service))
            .build();

    this.client.get().uri("/api/todos/t_1")
            .accept(MediaType.APPLICATION_JSON_UTF8)
            .exchange()
            .expectStatus().isOk()
            .expectBody()
            .jsonPath("$.title", "Wohnung putzen").
exists()
            .jsonPath("$.completed", "true").exists();
}
```

Listing 6: Test-Methode für den REST-Controller

Torsten Kohn
t.kohn@reply.de

Torsten Kohn ist Software Engineer bei der Comsysto Reply GmbH in München. Sein Schwerpunkt liegt in der Entwicklung von Software-Lösungen mithilfe von Spring und Java. Er ist stets auf der Suche nach neuen Konzepten und Vorgehen in der Entwicklung von Software mit Java und häufig auf Meetups rund um Software-Entwicklung im Raum München anzutreffen.