Fakultät für Informatik und Mathematik
Hochschule München

Reactive Web Backends mit Spring WebFlux

Seminararbeit

Aktuelle Technologien zur Entwicklung verteilter Java-Anwendungen

Vorgelegt von:	Michael Fuchs
	michael.fuchs@hm.edu
	Wolfratshauser Str. 15
	82335 Berg
Matrikelnummer:	19878416
Fachsemester:	6
Studiengang:	B.Sc. Informatik

Abgabe: 17. Mai 2019

Dozent: Michael Theis

Inhaltsverzeichnis

Inhaltsverzeichnis iii

Abbildungsverzeichnis iv

Listings v

1 Einleitung 1

2 Konzepte des Reaktiven Programmierens 2
- 2.1 Grundlagen 2
- 2.2 Reaktive Systeme 3
- 2.3 Project Reactor 5
- 2.4 Hype oder Propagation of Change 7

3 Das Spring WebFlux-Modul 8
- 3.1 Überblick 8
- 3.2 Klassische annotierte Controller 9
- 3.3 Funktionale Endpunkte 11
- 3.4 WebClient 14
- 3.5 Datenquellen 15
- 3.6 Testen 16
- 3.7 Security 17

4 Evaluation von Spring WebFlux 19

5 Fazit 21

Literatur 22

A Anhang a
- A.1 Vergleich Servlet-Stack mit Reactive-Stack . . . . a

Eidesstattliche Erklärung b

Abbildungsverzeichnis

1 Zeitleiste eines `Mono` mit *Processor* [4] 5
2 Zeitleiste eines `Flux` mit *Processor* [4] 5
3 WebFlux ist sowohl im klassischen als auch im asynchronen Servlet-Container lauffähig, wie z.B. Tomcat, Jetty oder Undertow [8]. 8
4 Beantwortung von mehreren Requests im blockierenden Servlet-Container [1] . . . . a
5 Beantwortung von mehreren Requests im nicht-blockierenden Reactive-Stack [1] . . . . a

Listings

1 Imperative Implementierung des Interfaces `Comparator` . . . 3
2 Funktionale Implementierung des Interfaces `Comparator` . . . 3
3 Interfaces für die Spezifikation der *Reactive Streams* . . . 4
4 Das `Mono` wird erst bei einem Aufruf der `subscribe`-Methode ausgeführt. 6
5 Zunächst werden mehrere Strings kontaktiert und anschließend wird ein `Mono` mit einem `Flux` verknüpft. . . . 6
6 Jede Millisekunde generiert das `Flux` einen Long-Wert und entnimmt davon die ersten Fünf, um diese anschließend auszugeben. . . . 6
7 Spring WebFlux als Maven-Abhängigkeit . . . 9
8 Spring MVC-Controller mit reaktiven Datentypen . . . 10
9 `cURL`-Aufruf des obigen Controllers im *Listing 8* . . . 10
10 Parameter als reaktive Daten . . . 10
11 Explizite Deklaration von *Server-sent Events*, bei welcher jede Sekunde eine Nachricht gesendet wird. . . . 11
12 Beispiel einer Handler-Function . . . 12
13 Verschiedene Arten Routen zu deklarieren und anschließend miteinander zu verknüpfen. . . . 12
14 Router-Funktion als Bean . . . 13
15 Handler für das vorherige *Lisiting 14* . . . 13
16 Konfiguration eines WebClients mit Basis-URL und Erweiterung um *Basic Authentication* durch die Methode `basicAuthentication(...)` der Klasse `ExchangeFilterFunctions`. . . . 14
17 Asynchrone Verarbeitung der Ergebnisse von mehreren Urls ohne Callbacks 14
18 Kombination mehrerer reaktiver Repositorys mit dem WebClient . . . 15
19 Deklaration einer reaktiven Zugriffsmöglichkeit mit Hilfe von Spring Data 15
20 Reaktive Datenverarbeitung innerhalb eines Post-Handlers . . . 16
21 `TestWebClient` als einfache Zugriffsmöglichkeit innerhalb des Tests . . . 17
22 Konkrete Implementierung eines `UserDetailService` . . . 18
23 Relative Konfiguration der URL-Sicherheit . . . 18

1 Einleitung

Die allgegenwärtige Verfügbarkeit des Internets und die weite Verbreitung von Smartphones bzw. Tablets propagieren große Belastungsspitzen von Enterprise-Webanwendungen. Der Ausbau des Highspeed-Internets und die Anzahl der steigenden öffentlichen Access-Points in Cafés, Bahnhöfen oder Hotels beschleunigen den Trend des *Ubiquitous computing* [10]. Die Gesellschaft ist zu jeder Zeit online und die simultane Integration von Webanwendungen in das Alltagsgeschehen ist gegenwärtig. Einen bedeutenden Anteil an dieser Entwicklung trägt der Streamingdienst Netflix, welcher mit einer Spitzenlast von ca. 40 % des weltweiten Internet Traffics belastet wird [5].
Die immer noch beständige Beliebtheit der Programmiersprache Java [9] und die weite Verbreitung des Spring-Frameworks, welches die Vereinfachung des JavaEE-Standards kompensiert, bringt nicht nur Veränderungen des Nutzerverhaltens mit sich, sondern auch die Notwendigkeit, sich mit alternativen Technologien auseinanderzusetzen. Die MVC-Architektur implementiert die Servlet-API des JavaEE-Standards und blockiert alle anderen Verbindungen, solange die TCP-Verbindung eines Requests aktiv ist. Mit Spring 5 wurde eine Alternative, der Reactive Stack, implementiert, welcher eine asynchrone Verarbeitung von Requests erlaubt.
In dieser Seminararbeit werden zunächst die Grundlagen des Reaktiven Programmierens erklärt, um anschließend auf reaktive Systeme eingehen zu können. Hierbei werden die Konzepte der asynchronen Datenverarbeitung anhand von Beispielen erläutert, welche mit dem Framework *Project Reactor* implementiert wurden. Nun können die Vorteile des reaktiven Programmierens erklärt werden. Im nächsten Kapitel wird das Spring WebFlux-Modul detailliert erläutert, sodass klassisch annotierte Controller oder funktionale Endpunkte programmiert werden können. Ferner wird auf den WebClient eingegangen, welcher primär für die Kommunikation zwischen Servern verwendet wird. Anschließend werden die wichtigsten Punkte weiterer Bestandteile von Webanwendungen, wie Datenquellen, Testen und Security, im reaktiven Umfeld erklärt. Im Anschluss wird Spring WebFlux mit der herkömmlichen Servlet-API des JavaEE-Standards verglichen, um abschließend die Vor- und Nachteile von Spring WebFlux evaluieren zu können.

2 Konzepte des Reaktiven Programmierens

2.1 Grundlagen

Programme der imperativen Sprachen wie C# oder Java bestehen aus Folgen von Anweisungen, welche immer in der gleichen Reihenfolge mit korrespondierenden Zustandsänderungen ausgeführt werden. Die moderne Multiprozessor-Architektur bei Servern in Rechenzentren als auch bei mobilen Endgeräten ermöglichen eine parallele Verarbeitung von großen Datenmengen. Um die volle Leistungsspanne der Hardware nutzen zu können, sind funktionale Programmiersprachen - wie Haskell - das Mittel der Wahl. Die Ausführung findet ohne Seiteneffekte statt und garantiert eine sichere Implementierung von Multi-Thread-Anwendungen ohne zusätzlichen Aufwand oder besondere Raffinesse. Die wichtigsten Elemente des funktionalen Sprachkerns werden mit Hilfe von Lambda-Ausdrücken definiert, sodass diese eine wichtige Grundlage für die reaktive Programmierung bilden. Ferner wird der Datenfluss von Programmen auch funktional behandelt. Dies kann man sich am besten durch das Berechnen des Terms $b * c + d$ erklären:
Bei der imperativen Ausführung des Ausdrucks $a = b*c+d$ steht das Ergebnis sofort fest, egal ob sich die Werte b, c oder d noch ändern. Die reaktive Programmierung reagiert hingegen dynamisch auf Änderungen der Werte, d.h. falls sich der Wert c ändert, so wird das Gesamtergebnis neu berechnet. Der Datenfluss des Programms reagiert auf Events und propagiert Änderungen an jeder Referenz des Codes, sodass die Werte dynamisch zur Laufzeit verändert werden können [3, S. 1-4]. Die konkreten Veränderungen der Register, des Stacks oder des Heaps sind demnach unabhängig vom Program-Counter. Die funktionale reaktive Programmierung ist ein neues Paradigma und lehnt sich an das Observer-Pattern und auf den Callback-Mechanismus von Javascript an. Dabei wird grundsätzlich zwischen zwei Komponenten voneinander unterschieden: Der *Observer* dient als Quelle von Daten, wie z.B. einen Input eines Users, eine Antwort der REST-Schnittstelle oder die Änderung einer einzelnen Variable. Das *Observable* hingegen reagiert entsprechend auf die entgegengenommenen Daten und suggeriert eine dynamische Reaktion.
Warum sollte man die reaktive der funktionalen oder sogar imperativen Programmierung

vorziehen, wenn das abstrahierende *Observer-Pattern* schon existiert? Zunächst ist funktionaler Code wesentlich kürzer und einfacher zu verstehen, wie im folgenden Beispiel dargestellt wird:

Listing 1: Imperative Implementierung des Interfaces `Comparator`

```
final Comparator<Integer> comp = new Comparator<>() {
    @Override
    public int compare(Integer o1, Integer o2) { return o2 - o1; }
};
```

Listing 2: Funktionale Implementierung des Interfaces `Comparator`

```
final Comparator<Integer> comp = (o1, o2) ->  o2 - o1;
```

Außerdem ist die JDK-Implementierung des *Observer-Pattern* schon längst überholt, weil diese den heutigen Anforderungen nicht mehr gewachsen ist und berechtigterweise auf *Deprecated* gesetzt worden ist. Abgesehen davon wird der Lesefluss durch die verteilte Implementierung des *Observer-Pattern* extrem erschwert[1].

2.2 Reaktive Systeme

Die detaillierte Erklärung von Spring WebFlux im nächsten Kapitel 1 erfordert eine grundlegende Erläuterung über Reaktive Systeme und wie *Project Reactor*[2] dieses System implementiert [2, S. 173].
Stabilität, Widerstandsfähigkeit und Flexibilität sind nur einige Anforderungen von modernen Applikationen, egal ob eine klassische Desktop- oder eine parallelisierte Enterprise-Anwendung entwickelt wird. Um allen Anforderungen gerecht zu werden, wurde am 16.09.2014 das *Reactive Manifesto*[3] spezifiziert als eine Definition von reaktiven Systemen.

- Gleichbleibend schnelle und zuverlässige Antwortzeit.
- Fehler eines Systems dürfen sich nicht auf andere Systeme auswirken.
- Eine automatische Wiederherstellung von Systemen gewährt die Hochverfügbarkeit.

[1]Eine ausführliche Erläuterung wird im nächsten Kapitel 1 beschrieben.
[2]Das *Project Reactor* ist unter `https://projectreactor.io/` zu finden.
[3]Das *Reactive Manifesto* kann unter `htttps://www.reactivemanifesto.org` eingesehen werden.

- Dynamisches Reagieren auf jede Änderung des Workloads
  –> Komponenten können gemeinsam genutzt oder reproduziert werden.
- Kostengünstige Anpassung der Rechnernetze ist erforderlich.
- Lose Kopplung aller Komponenten erfordert asynchronen Austausch.
- Jeder Knoten muss das *Back-Pressure-Pattern* unterstützen, um auf Änderungen der Last oder der Flusskontrolle reagieren zu können.
- Nicht blockierende Kommunikation verringert den Systemaufwand, weil jederzeit die Verbraucherressourcen aktiv sind.

Die Spezifikation *Reactive Streams*[4] definiert vier einfache Interfaces, die die Grundlage aller weiteren Implementierungen bilden. Des Weiteren ist die Implementierung von *Reactive Stream* Teil der *Flow-API* des JDK9 [2, S. 173-175].

Listing 3: Interfaces für die Spezifikation der *Reactive Streams*

```
1  public final class Flow {
2      /**
3      * A producer of items received by Subscribers.
4      * @param <T> the published item type
5      */
6      @FunctionalInterface
7      public static interface Publisher<T> { ... }
8      /**
9      * A receiver of messages.
10     * @param <T> the subscribed item type
11     */
12     public static interface Subscriber<T> { ... }
13     /**
14     * Message control linking a Publisher and Subscriber.
15     */
16     public static interface Subscription { ... }
17     /**
18     * A component that acts as both a Subscriber and Publisher.
19     * @param <T> the subscribed item type
20     * @param <R> the published item type
21     */
22     public static interface Processor<T,R> extends Subscriber<T>,
           ↪ Publisher<R> { ... }
23 }
```

[4]Diese Spezifizierung ist unter `https://www.reactive-streams.org` definiert.

2.3 Project Reactor

Sowohl *RxJava* als auch *Project Reactor* sind Implementierungen der gerade erwähnten *Flow-API* und letztere bildet die Basis-Implementierung ohne zusätzliche Abhängigkeiten für Spring Webflux. Die Library *Reactor*[5] erweitert die *Reactive Streams* um zwei wesentliche Bestandteile: `Flux[N]` und `Mono[0|1]`. Diese sind asynchrone und kombinierbare Sequenzen von Elementen, welche entweder aus keinem, einem oder beliebig vielen Elementen bestehen können. Im Wesentlichen transformiert der *Processor* die Daten für den *Subscriber*, welche der *Publisher* veröffentlicht. Diesen Zusammenhang stellen die beiden folgenden Marble-Diagramme in Abhängigkeit der Zeit dar [2, S. 175].:

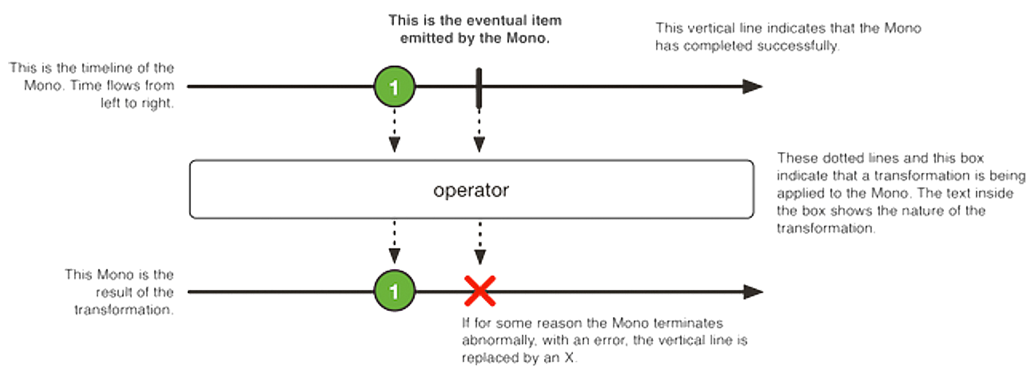

Abbildung 1: Zeitleiste eines `Mono` mit *Processor* [4]

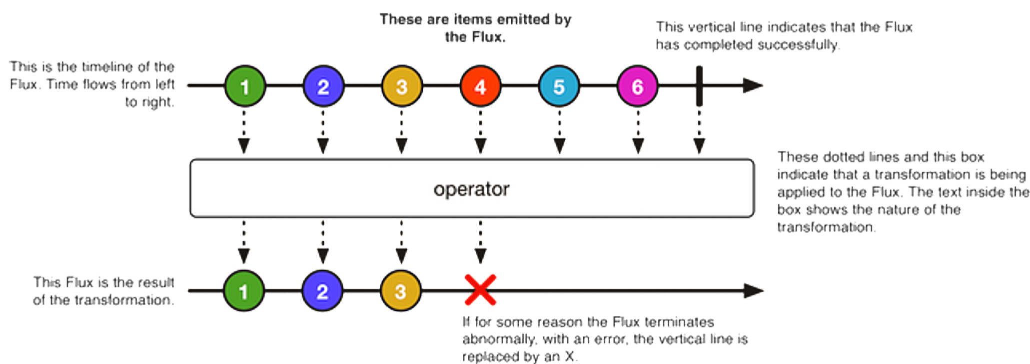

Abbildung 2: Zeitleiste eines `Flux` mit *Processor* [4]

[5] Die Details der Implementierung können unter `https://projectreactor.io/` eingesehen werden.

Außerdem dienen die *Reactors* als IPC-Komponente oder als klassische HTTP-Engine, um die Unterstützung von Websocket, TCP- und UDP-Verbindungen mit entsprechendem De- und Encoding der gesendeten Daten zu gewährleisten. Schließlich kann ein Durchsatz von mehreren Millionen Nachrichten pro Sekunde mit niedrigem Verbrauch der Ressourcen erzielt werden [2, S. 176].
Des Weiteren muss der Programmierer das richtige Concurrency-Modell wählen, um die definierten Arbeitseinheiten einem Thread zuzuordnen. Die Initialisierung der verschiedenen Scheduler findet mit Hilfe der dazugehörigen Factory-Klasse statt. Es stehen somit Scheduler zur Verfügung, welche entweder den gerade arbeitenden Thread oder einen einzelnen, wiederbenutzbaren Thread verwenden. Nichtsdestotrotz kann auch ein elastischer Pool von Threads verwendet oder ein Thread eines Worker-Pools verwendet werden [7, S. 271].
Die folgenden Beispiele (Listings 7, 5 und 6) demonstrieren einen kleinen Ausschnitt der Reactor-API und zeigen die Konstruktion des Datenflusses auf:

Listing 4: Das `Mono` wird erst bei einem Aufruf der `subscribe`-Methode ausgeführt.

```
1 Mono.just("Hello, World")
2     .subscribe(System.out::println);        // Ausgabe: Hello, World
```

Die Kombination von mehreren reaktiven Datentypen ist auch möglich:

Listing 5: Zunächst werden mehrere Strings kontaktiert und anschließend wird ein `Mono` mit einem `Flux` verknüpft.

```
1 Mono.just("Hello")
2     .flatMap(s -> Mono.just(s + " , World"))
3     .subscribe(System.out::println);        // Ausgabe: Hello, World
4 Mono.just("Hello")
5     .contactWith(Flux.just(",", "World"))
6     .subscribe(System.out::println);        // Ausgabe: Hello, World
```

Im Folgenden werden die Daten nicht auf Basis von skalaren Werten, sondern auf Basis von Datenstrukturen erzeugt.

Listing 6: Jede Millisekunde generiert das `Flux` einen Long-Wert und entnimmt davon die ersten Fünf, um diese anschließend auszugeben.

```
1 Flux.interval(Duration.ofMillis(1L))
2     .subscribeOn(Schedulers.parallel())
3     .take(5).map(Long::toString)
4     .subscribe(System.out::println);        // Ausgabe: 1 2 3 4 5
```

Die parallele Verarbeitung wird dabei explizit durch den Scheduler gefordert.

2.4 Hype oder Propagation of Change

Um den technischen Anforderungen und den domänenspezifischen Zielen einer modernen Webanwendung gerecht zu werden, ist eine durchdachte und stabile Software-Architektur nötig. Die richtige Architektur eines neuen Produktes erfordert eine unvoreingenommene Bewertung aller beteiligten Software-Komponenten. Schließlich können aktuelle Trends in der Softwareentwicklung Denkanstöße liefern, um neue Denkweisen oder Technologien in die klassiche Software-Architektur zu integrieren. Die folgende Aufgliederung fasst einige Punkte des Statusquo der Software-Architektur zusammen, welche unmittelbar mit dieser Seminararbeit in Verbindung stehen [6].

- Microservices erfordern sehr gute Qualität der Software.
- Zuverlässigkeit und Hochverfügbarkeit von massiv verteilten Software-Systemen.
- Event-getriebene Systeme bieten viele Vorteile bei lose gekoppelten Microservices.

Die Libraries der Reaktiven Programmierung sind meist in Form einer *Fluent-API* geschrieben, um eine kompakte Implementierung zu ermöglichen und um die verstreute Ansammlung von Business-Logik zu vermeiden. Die Event-getriebene Schreibweise der domänenspezifischen Logik lässt sich leichter verstehen und mit Hilfe von Continuous Code Analysis steigt die Qualität der Software.
Bei der herkömlichen Programmierung bleibt ein Thread bis zur vollständigen Beantwortung des Requests blockiert. Um eine beständige und zugleich schnelle Massenverarbeitung von Daten bei gleichzeitig vielen Zugriffen garantierten zu können, müssen mehrerer Instanzen einer Applikation installiert werden. Eine ökonomisch vertretbare Hochverfügbarkeit erfordert also eine gut skalierbare Infrastruktur und eine dynamisch Erweiterung von massiv parallel gesteuerten Rechnernetzen. Das reaktive Programmiermodell arbeitet dagegen asynchron und kann aus diesem Grund deutlich mehr Anfragen pro Zeiteinheit beantworten, was wiederum eine Reduktion der Rechenkosten bedeutet[6].

[6] Erläuterung der Gegenüberstellung im Kapitel 3 mit einer äquivalenten Verfügbarkeit.

3 Das Spring WebFlux-Modul

3.1 Überblick

Spring ist in den letzten Jahren zu einem der wichtigsten Frameworks im Umfeld von Webanwendungen geworden, weil dieses Framework die Handhabung der JavaEE Software-Komponenten vereinfacht und eine lose Kopplung dieser ermöglicht. Ferner betrachtet bieten die Module Web-MVC und Web-Flow übersichtliche MVC-Strukturen sowie etablierte Techniken, wie JPT und JSP. Seit Spring 5 wird die reaktive Programmierung auf Basis der *Flow-API* des JDK9 unterstützt. Aus diesem Grund und mit der Verfügbarkeit der Servlet-3.1-Umgebungen koexistieren Spring Web MVC und Spring WebFlux. Um die reaktive Programmierung effektiv nutzen zu können, haben auch andere Spring-Projekte, wie z.B. Spring Data, sowie Webserver asynchrone Schnittstellen bereitgestellt.
Wie jede andere Komponente in der Spring-Familie ist auch WebFlux ein flexibles Modul, bei welchem zwei unterschiedliche Programmiermodelle - funktional oder basierend auf Annotationen - genutzt werden können. In der folgenden Abbildung ist der reaktive Stack auf der rechten Seite und der klassische Stack auf der linken Seite abgebildet [8].

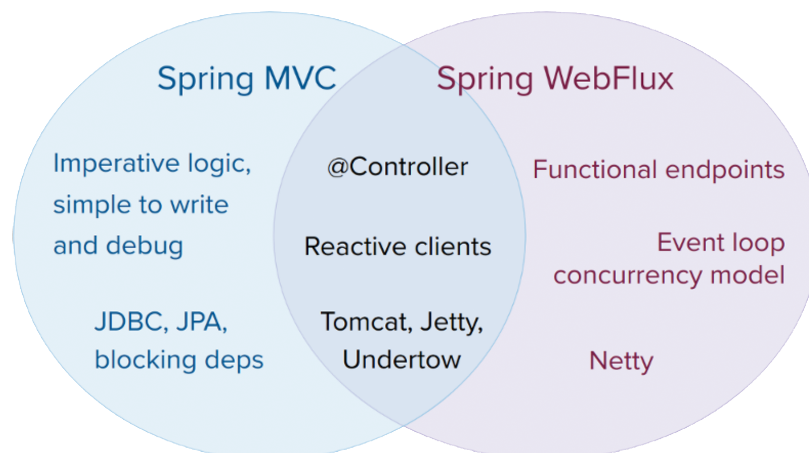

Abbildung 3: WebFlux ist sowohl im klassischen als auch im asynchronen Servlet-Container lauffähig, wie z.B. Tomcat, Jetty oder Undertow [8].

Grundsätzlich kann jeder Server für Spring WebFlux verwendet werden, welcher entweder den Servlet-3.1-Container (Tomcat oder Jetty) implementiert oder eine asynchrone Laufzeitumgebung (Netty oder Undertow) bereitstellt. Die Korrelation zwischen Server und Anwendungsfall wird im folgendem aufgelistet:

- Netty: Meist genutzter asynchroner Server mit gemeinsamer Nutzung von Ressourcen durch Client und Server –> Standard von `spring-boot-starter-webflux`
- Undertow: Keine Verwendung der Servlet-API beim Zugriff durch reaktiven Stack
- Tomcat & Jetty: Kein gleichzeitiges Nutzen beider Stacks ohne Abstriche
  - MVC nutzt Servlet-API direkt, ist jedoch auf Servlet-Blocking angewiesen
  - Webflux verwendet Servlet-API ausschließlich hinter einem Low-Level-Adapter

Der einzige Unterschied bei annotierten Controllern zwischen Spring MVC und Spring WebFlux besteht im Parallelitätsmodell und den Standardannahmen für das Blockieren von Threads. Im Allgemeinen verwenden Webanwendungen mit einem Servlet-Container einen großen Thread-Pool, weil jeder Request solange den aktuellen Thread blockiert, bis die Anfrage beantwortet ist. Bei nicht blockierenden Servern hingegen wird davon ausgegangen, dass die Requests keinen Thread bei der Beantwortung blockieren. Aus diesem Grund wird ein Thread-Pool mit fester und kleinerer Größe verwendet [8].
Schließlich kann Spring WebFlux beispielsweise mit dem folgenden Spring-Boot Starter als Maven-Abhängigkeit in jedes Projekt eingebunden werden.

Listing 7: Spring WebFlux als Maven-Abhängigkeit

```
1 <dependency>
2     <groupId>org.springframework.boot</groupId>
3     <artifactId>spring-boot-starter-webflux</artifactId>
4     <version>2.1.4.RELEASE</version>
5 </dependency>
```

3.2 Klassische annotierte Controller

Die Annotation aus dem *spring-web*-Modul, wie z.B. `@Controller` oder `@GetMapping`, als auch die dazugehörigen Rückgabetypen der Controller unterstützen reaktive Datentypen. Aus diesem Grund kann die Verwendung der Typen innerhalb eines konkreten Kontextes, Spring WebFlux oder Spring MVC, nicht leicht unterschieden werden. Ein bemerkenswerter Unterschied besteht darin, dass WebFlux auch reaktive `@RequestBody`-Argumente unterstützt [8].

Der Aufruf des *Listing 7* aus *Abschnitt 2.3* liefert nun das Ergebnis `"Hello, World"` als HTTP-Response zurück.

Listing 8: Spring MVC-Controller mit reaktiven Datentypen

```
@RestController
public class RestController {
    @GetMapping("/")
    public Mono<String> get(
        @RequestParam(defaultValue = "World") String name) {
        return Mono.just("Hello")
            .flatMap(s -> Mono.just(
                String.format("%s, %s\n", s, name)
            ));
    }
}
```

Lediglich der Typ des Rückgabewertes lässt darauf zurückschließen, dass eine Ähnlichkeit zu den klassischen MVC-Controllern besteht. Die Verarbeitung des Requests wird jedoch vollständig blockadefrei ausgeführt. Der Aufruf des *Listing 8* aus *Kapitel 2.3* liefert nun das Ergebnis `"Hello, World"` als HTTP-Response.

Listing 9: `cURL`-Aufruf des obigen Controllers im *Listing 8*

```
$ curl 127.0.0.1:8080
Hello, World
```

Auf der Seite der Clients ändert sich zunächst einmal nichts, weil sowohl Server als auch der aufrufende Client das selbe Protokoll sprechen. Die Spezifikation des HTTP-Standards erwartet außerdem eine Übereinstimmung des Content-Typs. Die üblich annotierten Request-Parameter oder Pfadvariablen unterstützen leider keine reaktiven Datentypen. Falls diese asynchron behandelt werden sollten, so müssen diese ebenfalls reaktiv implementiert werden *Listing 10*. In *Zeile 4* des Code-Fragments wird aus dem `Mono` ein `Flux`, um ein einzelnes Element auf mehrere abzubilden. Zunächst werden unendliche viele Elemente erzeugt, um anschließend durch die `zipWith`-Methode mit der Folge $1, 2, ..., 22, 23$ vereinigt zu werden. Ferner entstehen Tupel, welche wiederum in Strings umgewandelt werden. Der *Publisher* ist die terminale Operation der Pipeline und somit werden keine neuen Elemente mehr generiert.

Listing 10: Parameter als reaktive Daten

```
@PostMapping("/")
public Flux<String> post(
    @RequestBody Mono<String> name) {
```

```
4         return name.flatMapMany(n -> Flux.generate(s -> s.next(n)))
5             .zipWith(Flux.range(1, 23))
6             .map(t -> String.format("Hello, %s\n", t.getT1()));
7 }
```

Des Weiteren ist die Generierung eines kontinuierlichen Stroms von Daten möglich. Dieser Stream wird erst dann beendet, sobald die Verbindung unterbrochen wird. So kann beispielsweise ein Download einer großen Datei durch den Browser behandelt werden, ohne einen Thread zu blockieren.

Listing 11: Explizite Deklaration von *Server-sent Events*, bei welcher jede Sekunde eine Nachricht gesendet wird.

```
1 @GetMapping(path = "/stream", produces = MediaType.
    ↪ TEXT_EVENT_STREAM_VALUE)
2 public Flux<String> stream() {
3     return Flux.interval(Duration.ofSeconds(1))
4         .map(s -> String.format("Hello, %d", s));
5 }
```

Entweder werden die Objekte einzeln als JSON-Objekte als *Server-sent Events*[1] serialisiert. Die als JSON dekodierten Elemente werden jeweils einzeln geflusht und dienen primär für die direkte Kommunikation zwischen Servern (s. Abschnitt 3.4). Alternativ kann als Medientyp `application/json` verwendet werden, wenn keine *Server-sent Events* genutzt werden sollen. Jedoch sollte beachtet werden, dass die Daten erst dann gesendet werden, wenn dies als Medientyp angegeben ist oder wenn der Client *Server-sent Events* explizit anfordert. Die SSE-Daten werden per Default als JSON dekodiert, dienen aber primär als Kommunikationsmittel zwischen Server und Browser bzw. sonstigen Clients [7, S. 275-277].

3.3 Funktionale Endpunkte

Grundlegende Konzepte Das Handeln der HTTP-Requests kann mit Hilfe des funktionalen Programmiermodells des WebFlux-Moduls auf verständliche und zugleich prägnant kurze Weise beschrieben werden. Die folgenden Bausteine bilden das Grundgerüst für dieses Modell:

- `HandlerFunction`: Diese Klasse nimmt `ServerRequests` entgegen, um diese jeweils mit einer eigenen `ServerResponse`-Instanz zu beantworten. Zum einen abstrahie-

[1] Die HTML-Spezifikation *Server-sent Events* (SSE) erlaubt eine einseitige Kommunikation, druch die ereignisgesteuerte Daten an den Client geschickt werden können - wovon reaktive Datentypen profitieren.

ren die Interfaces `ServerRequest` sowie `ServerResponse` WebFlux von der Servlet-API und zum anderen werden durch die Immutability Seiteneffekte vermieden. Die Klasse `HandlerFunction` ist ein `@FunctionalInterface` und kann entsprechend als Klasse, Lambda oder Methodenreferenz geschrieben werden. Im *Listing 12* wird ein einkommender Request erfolgreich mit der Response `Hello, World` ohne Verzögerung beantwortet, weil durchgehend reaktive Datentypen eingesetzt werden.

- `RouterFunction`: Hierbei werden die HTTP-Requests, wie bei den klassischen Annotationen `@RequestMapping`, auf entsprechende Handler in Abhängigkeit der Pfade, Medientypen und HTTP-Methoden abgebildet. Die eingehenden Verbindungen können entweder manuell, also statisch, oder automatisiert durch Prädikate (`RequestPredicates`) auf die korrespondierenden Controller geleitet werden. Alternativ können auch Inline-Functions zum Auf- und Abbau der Routen verwendet werden. Im *Listing 13* werden die unterschiedlichen Möglichkeiten sowie die Verknüpfung von Router-Funktionen aufgezeigt.

- `HandlerFilterFunction`: Anfragen können vor- oder nachgelagert durch Implementierungen der `HandlerFilterFunction` gefiltert werden, um die Verarbeitungskette bei Bedarf abbrechen zu können. Die Funktionsweise der Filterung ist vergleichbar mit dem *Servlet Filtern* [8] [7, S. 282-284].

Listing 12: Beispiel einer Handler-Function

```
1  HandlerFunction<ServerResponse> helloWorld =
2      request -> ServerResponse.ok().body(
3          Mono.just("Hello, World"), String.class);
```

Listing 13: Verschiedene Arten Routen zu deklarieren und anschließend miteinander zu verknüpfen.

```
1  RouterFunction<ServerResponse> helloSpringRoute = request -> {
2      if(request.method() == GET && request.path().equals("/"))
3          return Mono.just(r ->
4              ok().body(Mono.just("Hello, World"),String.class));
5      else
6          return Mono.empty();
7  };
8  RouterFunction<ServerResponse> helloWorldRoute =
9      RouterFunctions.route(GET("/helloworld"), helloWorld);
10 RouterFunction routes = helloSpringRoute.and(helloWorldRoute);
```

Funktionale Endpunkte mit Spring-Boot Spring-Boot erkennt alle `RouterFunctions` als Beans und registriert diese im entsprechenden Handler. Die Standardeinstellung des Spring-Boot Starters ist Netty und dies spiegelt sich im Typ des Handlers wieder.

Listing 14: Router-Funktion als Bean

```
@Bean
RouterFunction<?> router(SiteHandler site, ApiHandler api) {
    return route(GET("/"), site::index)
        .andNest(path("/api"),
            route(POST("/filmWatched"), api::filmWatched)
    .andRoute(GET("/watchedRightNow"), api::watchedRightNow));
```

Die kompakte Schreibweise erlaubt ein einfaches Verständnis des komplexen Datenflusses: Alle Anfragen «`/`» werden auf die Methode `SiteHandler#index` geroutet. Falls der Präfix der Anfrage «`/api`» ist, so werden alle Request auf die entsprechenden Sub-Routen weitergeleitet.

Listing 15: Handler für das vorherige *Lisiting 14*

```
class ApiHandler {
    // FilmWatchedEventRepository als Attribut & Konstruktor;

    Mono<ServerResponse> filmWatched(ServerRequest request) {
        return ok().body(request.bodyToMono(Film.class)
                .map(FilmWatchedEvent::new)
                .flatMap(repository::save), FilmWatched.class);
    }

    Mono<ServerResponse> watchedRightNow(ServerRequest request) {
        return ok().contentType(MediaType.TEXT_EVENT_STREAM)
                .body(repository.streamAllBy()
                        .filter(e ->
                                e.getWatchedOn().isAfter(now())
                        ), FilmWatched.class);
    }
}
```

Lediglich der Body der Response unterscheidet sich von den beiden Methoden, denn beide geben ein `Mono<ServerResponse>` zurück. Die Methode `filmWatched` erstellt einen neuen Film und persistiert diesen in der Datenbank. Schließlich wird das gerade eben gespeicherte Ergebnis dem Client als Bestätigung zurückgegeben. Im Gegensatz dazu gibt die Methode `watchedRightNow` einen kontinuierlichen Stream von Ereignissen in Form eines `Flux<FilmWatchedEvent>` zurück [7, S. 284-288] [8].

3.4 WebClient

Die Beantwortung von asynchronen HTTP-Requests können durch das Interface `WebClient` auf einfache Art und Weise beantwortet werden und ist zugleich eine Alternative zum klassischen RestTemplate. Die Nachrichten zwischen den Kommunikationspartnern werden dabei nicht als `InputInstream` bzw. `Outputstream` behandelt sondern reaktiv als `Flux<Buffer>`. Die Nachricht im *Buffer* wird in der Regel automatisch interpretiert, sodass JSON-, XML- und SSE-Dateien serialisiert und deserialisiert werden können [8].

Listing 16: Konfiguration eines WebClients mit Basis-URL und Erweiterung um *Basic Authentication* durch die Methode `basicAuthentication(...)` der Klasse `ExchangeFilterFunctions`.

```
1 @Bean
2 WebClient webClient() {
3     return WebClient
4         .create("http://127.0.0.1:8080/api")
5         .mutate()
6         .filter(basicAuthentication("spring", "boot"))
7         .build();
8 }
```

Die Filterung von Anfragen ist ein essentieller Bestandteil einer strukturierten und effizienten Verarbeitung. Die `ExchangeFilterFunction` ist ein funktionales Interfaces und kann zu komplexeren Spezifikationen verkettet werden, um die Anfragen des WebClients filtern zu können. Des Weiteren wird durch `mutate()` ein Builder auf Basis der bisher gegebenen Informationen gebaut, welcher nachfolgend bequem auf die eigenen Bedürfnisse abgestimmt werden kann.
Die Liste der Filme wird im *Listing 17* zunächst abgerufen, um anschließend die Antwort aus dem `Flux<DataBuffer>` extrahieren zu können und um weiterverarbeitet zu werden.

Listing 17: Asynchrone Verarbeitung der Ergebnisse von mehreren Urls ohne Callbacks

```
1 client
2     .get().uri("/films")
3     .retrieve().bodyToFlux(Film.class)
4     .filter(film -> film.getTitle().contains("goldfinger"))
5     .flatMap(film -> client
6         .get().uri("/films/{id}/stream", film.getId())
7         .retrieve().bodyToFlux(Film.class))
8     .subscribe(System.out::println);
```

Durch den Aufruf der `stream`-Methode im *Listing* **??** werden die einzelnen Elemente auf eine Menge von Elementen abgebildet. Die *Streaming*-Methode gibt natürlich keinen

richtigen Film wieder, sondern demonstriert die reaktive Verarbeitung anhand eines Beispiels. Ein `Mono<Film>` wird aus der Datenbank asynchron geladen und auf eine unendlich Menge an Objekten abgebildet. Der Code der ersten `doOnNext`-Methode wird erst dann ausgeführt, wenn der Datenfluss in Gang gesetzt wird. Innerhalb des `flatMap`-Aufrufs wird der Film einem Service (*Listing 18*) übergeben. Dieser stellt alle Filme zur Verfügung, die gerade aktiv angesehen werden. Schließlich wird die Verarbeitungskette auch erst begonnen, wenn eine explizite *Subscription* vorhanden ist [8] [7, S. 279-281].

Listing 18: Kombination mehrerer reaktiver Repositorys mit dem WebClient

```
1  @GetMapping(path = "/api/integer/{id}/stream", produces = MediaType.
       ↪ TEXT_EVENT_STREAM_VALUE)
2  public Flux<Integer> stream(@PathVariable Long id) {
3      return integerRepository.findById(id)
4          .flatMapMany(number -> Flux.<Integer>generate(sink -> sink.
               ↪ next(number)))
5          .zipWith(Flux.interval(Duration.ofSeconds(1)))
6          .doOnNext(tuple -> client
7              .post().uri("/api/watched")
8              .accept(MediaType.TEXT_EVENT_STREAM)
9              .body(Mono.just(tuple.getT1()), Integer.class)
10             .retrieve().bodyToFlux(IntegerWatchedEvent.class)
11             .subscribe(System.out::println))
12         .map(Tuple2::getT1);
13 }
```

3.5 Datenquellen

Mit SpringData steht ein sehr einfacher Zugriff auf die unterschiedlichsten Datenbanken zur Verfügung, welche mit JDBC inhärent synchron angebunden werden und dadurch pro Verbindung einen Thread belegen. Die Datenbanken (z.B. MongoDB, Redis, Apache Cassandra oder Couchbase) verbinden sich mit einem asynchronen Treiber, um die CRUD-Operationen reaktiv von Ende-zu-Ende durchführen zu können. An der Verwendung von Spring Data ändert sich nichts und jegliche Funktionalität kann wie gewohnt verwendet werden. Außerdem können bei den Zugriffen auf die konkrete Datenbank reaktive Datentypen zurückgegeben werden.

Listing 19: Deklaration einer reaktiven Zugriffsmöglichkeit mit Hilfe von Spring Data

```
1  public interface IntegerRepository
2      extends ReactiveCassandraRepository<Integer, Long> {
```

```
3       @AllowFiltering
4       Flux<Integer> findGreaterThan(int number);
5   }
```

Der folgende Post-Handler nimmt eine `List<Integer>` als `@RequestBody` entgegen, um anschließend weiterverarbeitet zu werden. Zunächst werden alle alten Elemente aus der Datenbank gelöscht und diese Methode gibt ein `Mono<Void>` zurück. Nun wartet `thenMany` solange bis das `Mono` signalisiert, dass der Datenbankzugriff abgeschlossen ist und füllt alle neuen Elemente in die Pipeline. Jeder Wert, der nicht `null` ist, wird gespeichert und das Ergebnis ausgegeben. Der Aufruf der Methode `blockLast()` ist notwendig, weil die Verarbeitungskette nur dann in Gang gesetzt wird, wenn mindestens ein *Subscriber* zur Verfügung steht [8].

Listing 20: Reaktive Datenverarbeitung innerhalb eines Post-Handlers

```
1   @PostMapping("/numbers")
2   void saveNumbers(@RequestBody List<Integer> numbers) {
3       integerRepository.deleteAll()
4           .thenMany(Flux.just(numbers))
5           .filter(Objects::nonNull)
6           .flatMap(integerRepository::saveAll)
7           .doOnNext(System.out::println)
8           .blockLast();
9   }
```

3.6 Testen

Das *Project Reactor* bildet die Basis des Technologie-Stacks von Spring WebFlux und stellt eine Bibliothek `io.projectreactor:reactor-test` zur Verfügung, mit der verschiedene reaktive Szenarien getestet werden können. Die reaktiven Datentypen, zukünftige Ereignisse, welche zu einem bestimmten Zeitpunkt auftreten, können durch den `StepVerifier` getestet werden. Darüber hinaus bietet Spring zusätzlich zu den üblichen Test-Annotationen einen `@AutoConfigureWebTestClient`. Der `WebTestClient` ist eine spezielle Instanz des `WebClients`, welcher auf das Testen von asynchronen WebFlux Endpunkten spezialisiert ist.

Der folgende Integrationstest *Listing 21* testet den reaktiven Endpunkt *Listing 15* inklusive richtiger Datenbankverbindung. Ein normaler Unit-Test wird im Umfeld von Spring MVC mit der Annotation `@WebMvc` eingeleitet und `@WebFluxTest` ist das korrespondierende Pendant. Die konkrete Logik eines Tests kann entweder via klassischen Annotationen oder mit einem funktionalen Programmiermodell implementiert werden [8].

Listing 21: `TestWebClient` als einfache Zugriffsmöglichkeit innerhalb des Tests

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.MOCK)
@AutoConfigureWebTestClient
public class FilmRestControllerTest {
    @Autowired
    private WebTestClient client;

    @Test
    public void filmApiShouldWork() {
        FluxChangeResult<Film> result = client
                .mutate()
                .filter(basicAuthentication("spring", "boot"))
                .build()
                .get()
                .accept(TEXT_EVENT_STREAM)
                .exchange()
                .expectStatus().isOk()
                .expectHeader().contentType(TEXT_EVENT_STREAM)
                .returnResult(Film.class);
        StepVerifier.create(result.getResponseBody())
                .consumeNextWith(film ->
                    assertThat(film.getTitle(), is("James Bond")))
                .expectNextCount(9)
                .expectComplete()
                .verify();
    }
}
```

Mit dem Testclient können gleiche oder ähnliche Anforderungen wie mit dem Mock-MVC-Testframework spezifiziert werden. Nach dem `exchange`-Schritt werden die Erwartungen schrittweise überprüft. In diesem Beispiel erwartet die Filterung in *Zeile 6* eine erfolgreiche Authentifizierung des Clients. Nun können die Erwartungen mit dem HTTP-Status-Code und dem Medientyp verglichen werden. Zuletzt wird der `StepVerifier` aufgerufen, um sicherzustellen, dass das erste Element den gegebenen Kriterien entspricht und weitere neun Elemente auftreten, bevor der `Flux` terminiert. Der eigentliche Aufruf von `verify` löst die Verarbeitungskette aus.

3.7 Security

Spring Security kann sowohl die klassische Programmierung mit Annotationen als auch die rein funktionale Programmierung absichern. Die Security-Mechanismen werden mit

der Annotation `@EnableWebFluxSecurity` eingeschaltet. Bei Klassen der Authentifizierung (z.B. Benutzerverwaltung) ist eine Anpassung der Default-Implementierung nötig, um diese via Propertys konfigurieren zu können. Damit ist die Konfiguration unabhängig vom Programmcode und einfach externalisierbar.

Listing 22: Konkrete Implementierung eines `UserDetailService`

```
1  public class ReactiveUserDetailsServiceImpl
2      implements ReactiveUserDetailsService {
3      @Override
4      public Mono<UserDetails> findByUsername(String username) {
5          return Mono.justOrEmpty(
6                  "romy".equals(username)
7                  ? User.withUsername("romy").build()
8                  : null
9          );
10     }
11 }
```

Standardmäßig sind alle Pfade der Webanwendung mit Spring Security geschützt und im folgenden *Listing 23* werden einzelne Zugriffspunkte modifiziert, wie es auch unter den Security-Einstellungen von Spring-MVC üblich ist [8].

Listing 23: Relative Konfiguration der URL-Sicherheit

```
1  @Bean
2  SecurityWebFilterChain security(ServerHttpSecurity http) {
3      return http
4          .authorizeExchange()
5          .pathMatchers("/api/watchedRightNow")
6              .authenticated()
7           .pathMatchers("/api/filmWatched")
8              .hasRole("STORE")
9           .anyExchange().permitAll()
10          .and().httpBasic().and()
11          .build();
12 }
```

4 Evaluation von Spring WebFlux

Spring MVC, welches den Servlet-Container des JavaEE-Standards implementiert, und Spring WebFlux, das eine Implementierung des Reactive Stack ist, können in einem Projekt ohne Probleme koexistieren oder sogar zusammenarbeiten, wie es eingangs im *Abschnitt 3.1* erläutert wurde. Nichtsdestotrotz sind diese Technologien auf entsprechende Anwendungsfälle optimiert und folgende Zusammenstellung soll die Stärken und Schwächen zusammenfassen:

Anwendbarkeit von Spring WebFlux bezogen auf Anwendungsfälle

- Bei bestehenden funktionierenden MVC-Anwendungen ist keine Änderung notwendig, weil die imperative Programmierung die einfachste Möglichkeit ist, Code zu verstehen, zu schreiben und zu debuggen. Außerdem werden die meisten Libaries und Frameworks unterstützt [8].
- Spring WebFlux bietet eine große Auswahl an Servern (Netty, Tomcat, Jetty und Undertow), Unterstützung für den 3.1-Servlet-Container und jene Vorteile anderer reaktiver Programmiermodelle. Des Weiteren stehen klassische annotierte und funktionale Endpunkte zur Verfügung [8].
- Kompakte, funktionale Schreibweise durch Lambdas oder mit Kotlin[1], um funktionale Endpunkte zu realisieren. Dieser Ansatz ist gerade bei kleinen Mikrodiensten mit weniger komplexen Anforderungen zu empfehlen, sodass von großer Transparenz und Kontrolle profitiert werden kann [8].
- Durch die Microservice-Architektur kann eine Mischung von klassischen und reaktiven Endpunkten ohne zusätzliches Wissen verwendet werden. Die Unterstützung für dasselbe annotationsbasierte Programmiermodell in beiden Frameworks erleichtert die Wiederverwendung von Wissen und die Auswahl des richtigen Tools für den richtigen Job [8].

[1] Spring 5 stellt mit Kotlin eine leichtgewichtige DSL zur Verfügung, sodass Beans ohne Annotation, Refelctions oder Proxys im Application-Kontext registriert werden können. Des Weiteren können sehr einfach kompakte Handler- und Router-Funktionen erstellt werden.

- Bei blockierenden Datenbanken und den dazugehörigen Persistenz-APIs ist Spring MVC die beste Wahl für gängige Architekturen. Jedoch ist es technisch möglich, blockierende Aufrufe für einen separaten Thread auszuführen, dass aber den reaktiven Stack nicht optimal nutzt. Somit kann auch Spring WebFlux mit einigen Abstrichen für nicht reaktive Datenbanken verwendet werden [8].

- Die Verwendung des WebClients von Spring WebFlux kann analog zu dem Web-Client von Spring MVC verwendet werden. Folglich können reaktive Datentypen direkt in den klassischen MVC-Controllern zurückgegeben werden. Je größer die Latenz pro Anruf oder die Abhängigkeit zwischen Anrufen ist, desto drastischer sind die Vorteile. Schließlich können Spring MVC-Steuerungen auch andere reaktive Komponenten aufrufen [8].

- Bei einem großen Team sollte die große Lernkurve beim Umstieg von imperativer zu reaktiver bzw. reaktiv-funktionaler Programmierung berücksichtigt werden. Der Umstieg eines Monolithen auf die nicht blockierende Beantwortung von Requests sollte iterativ in kleinen Schritten begonnen werden, um die direkten Auswirkungen gleich evaluieren zu können [8].

Performance von Spring WebFlux im Vergleich zu Spring MVC[2]
Die Performance einer Webapplikation hat viele Eigenschaften und Bedeutungen. Die reaktive und nicht blockierende Verarbeitung von Requests sorgt im Allgemeinen nicht dafür, dass die Anwendung schneller läuft. In einigen Fällen kann dies durchaus möglich sein, vor allem wenn der WebClient parallel Daten einer Rest-Schnittstelle abruft. Grundsätzlich ist bei der blockierungsfreien Beantwortung mehr Arbeit erforderlich, was die Verarbeitungszeit geringfügig verlängert.
Der Hauptvorteil der nicht blockierenden Beantwortung von Requests ist die Möglichkeit, mit einer kleinen, festen Anzahl von Threads und weniger Speicher zu skalieren. Aus diesem Grund sind Anwendungen unter Volllast widerstandsfähiger, weil sie vorhersehbar skaliert werden können. Der reaktive Stack zeigt vor allem seine Stärken bei einer guten Latenz und dabei können die Unterschiede drastisch sein [8].

[2] Der Unterschied zwischen der Beantwortung von Requests bei Servlet-Stack und Reactive-Stack wird in der Illustration (*Anhang A.1*) dargestellt.

5 Fazit

Im Rahmen dieser Seminararbeit wurden zunächst die Konzepte der reaktiven Programmierung erklärt, um anschließend die Verarbeitung von HTTP-Requests innerhalb des Spring WebFlux-Moduls zu verstehen. Mögliche nicht-blockierende REST-Endpunkte können auf zwei verschiedene Arten implementiert werden: Die klassisch annotierten Controller erlauben eine genauso einfache Definition von Handlern, wie es bei Spring MVC üblich ist. Im Gegensatz dazu können die HTTP-Requests bei den funktionalen Endpunkten auf verständliche und zugleich kurze Weise verarbeitet werden. Schließlich wurde in diesem Kapitel Spring WebFlux mit der herkömmlichen Servlet API aus dem Java EE-Standard verglichen, um die Unterschiede der beiden Module hervorzuheben. Abschließend wurde Spring WebFlux nach seinen Stärken und Schwächen hinsichtlich der Anwendbarkeit sowie der Performance bewertet.
Grundsätzlich können Reactive Web Backends mit Spring WebFlux eine Alternative zu den üblichen Web Backends unter Berücksichtigung der folgenden Punkte sein: Die *Flow-API* der reaktiven Programmierung bietet eine kompakte und sehr leserliche Schreibweise, weil jegliche Business-Logik an einen Ort im Code bearbeitet wird und jede Translation eines Objektes sofort ersichtlich ist. Des Weiteren müssen bei modernen Web Backends oft mehrere Millionen Requests pro Sekunde verarbeitet werden und durch Spring WebFlux kann man diese harten Anforderungen schon mit wenigen Ressourcen erfüllen. Außerdem ist die reaktive Implementierung von Webapplikationen eine gute Alternative zur imperativen Programmierung, um neue Architekturmuster zu erlernen und um den eigenen Programmierstil zu verbessern.

Literatur

[1] Lokesh Gupta. *Spring WebFlux Tutorial.* URL: `https://howtodoinjava.com/spring-webflux/spring-webflux-tutorial/` (besucht am 21. 04. 2019).

[2] Felipe Gutierrez. *Pro Spring Boot 2: An Authoritative Guide to Building Microservices, Web and Enterprise Applications, and Best Practices.* apress, 2019.

[3] Andrea Magile. *Reactive Java Programming.* apress, 2016.

[4] Stephane Maldini und Simon Baslé. *Reactor 3 Reference Guide.* URL: `https://projectreactor.io/docs/core/release/reference/` (besucht am 21. 04. 2019).

[5] Chris Morris. *Netflix Consumes 15% of the World's Internet Bandwidth.* URL: `http://fortune.com/2018/10/02/netflix-consumes-15-percent-of-global-internet-bandwidth/` (besucht am 21. 04. 2019).

[6] Hartmut Schlosser. *Software-Architektur-Trends 2018: Auf diese Themen lohnt sich der Blick.* URL: `https://jaxenter.de/software-architektur-trends-2018-73941` (besucht am 21. 04. 2019).

[7] Michael Simons. *Spring Boot 2.* Bd. 1. dpunkt, 2018.

[8] Pivotal Software. *Web on Reactive Stack.* URL: `https://docs.spring.io/spring/docs/current/spring-framework-reference/web-reactive.html` (besucht am 22. 04. 2019).

[9] stackoverflow. *Developer Survey Results 2019.* URL: `https://insights.stackoverflow.com/survey/2019/#most-popular-technologies` (besucht am 21. 04. 2019).

[10] Mark Weiser. „The Computer for the 21st Century". In: *Scientific American* 265 (Sep. 1991), S. 66–75.

A Anhang

A.1 Vergleich Servlet-Stack mit Reactive-Stack

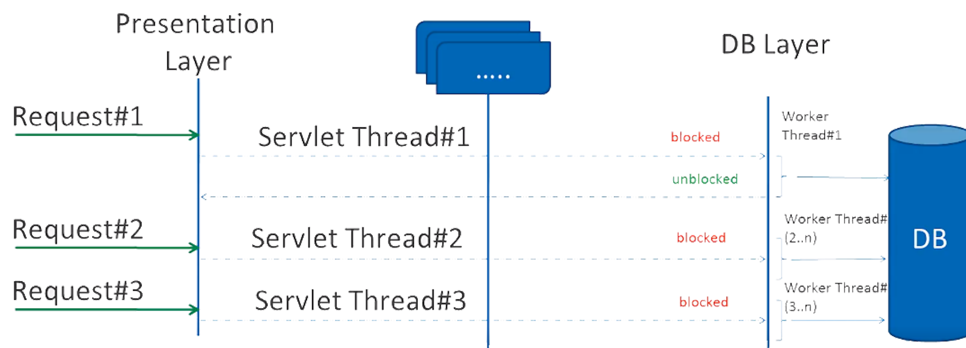

Abbildung 4: Beantwortung von mehreren Requests im blockierenden Servlet-Container [1]

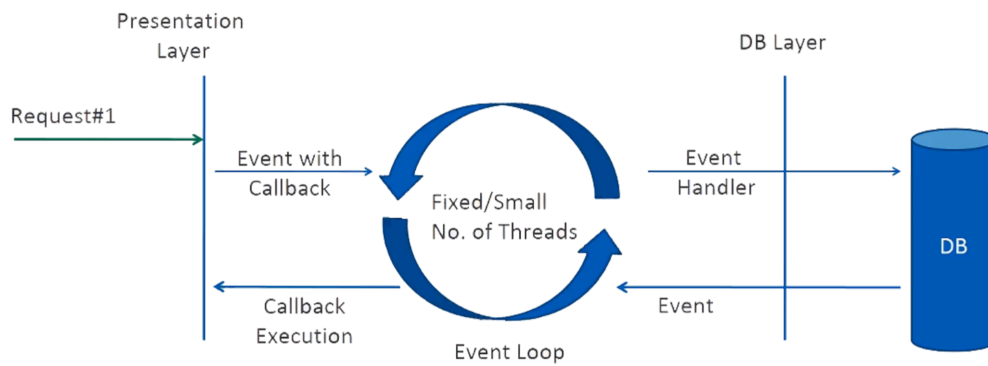

Abbildung 5: Beantwortung von mehreren Requests im nicht-blockierenden Reactive-Stack [1]

Eidesstattliche Erklärung über das selbstständige Verfassen der vorliegenden Seminararbeit

Hiermit versichere ich, dass ich die vorliegende Seminararbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle Textstellen, die dem Wortlaut oder dem Sinne nach anderen Texten entnommen sind, sowie alle Zeichnungen, bildliche Darstellungen, Skizzen und Tabellen wurden unter Angabe der Quellen (einschließlich des World Wide Web und anderer elektronischer Text- und Datensammlungen) nach den üblichen Regeln des wissenschaftlichen Zitierens nachgewiesen. Ich versichere ferner, dass die vorliegende Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war. Mir ist bewusst, dass wahrheitswidrige Angaben als Täuschungsversuch behandelt werden.

Höhenrain, den 17. Mai 2019 Michael Fuchs