andrena OBJECTS

Experts in agile software engineering

Was können 4 Threads, was 600 nicht können?

Von synchronen zu reaktiven Webservices mit Webflux und Reactor

Java Forum Stuttgart 2022

Jonas Nagy-Kuhlen
jonas.nagy-kuhlen@andrena.de

Jonas Nagy-Kuhlen

- Agiler Softwareentwickler bei andrena seit 2019
- Entwickelt professionell Software seit mehr als 8 Jahren
  - Desktop-Anwendungen in C#/.NET
  - Webservices basierend auf Java/Spring
- Fokus auf performancekritische, nebenläufige Systeme
- Begeistert sich für agile Methoden und Clean Code

github.com/jnagykuhlen

andrena in Zahlen und Fakten

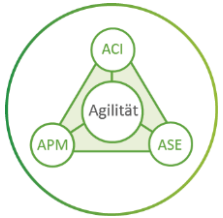
Gründung
1995 in Karlsruhe

Leistungen

Umsatz
2021 ~ 34 Mio. €
2020 ~ 28 Mio. €

6 Standorte
Karlsruhe (Hauptsitz), Frankfurt,
München, Stuttgart, Mannheim, Köln

Mitarbeiter*innen
rund 400



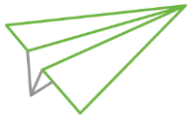
Kernkompetenzen
Digitale Produktinnovationen

Agenda

Motivation
Ein Erfahrungsbericht

Reaktive Programmierung
Asynchrone Codeausführung in Java

Live Demo
Hello World Enterprise Edition™



Learnings

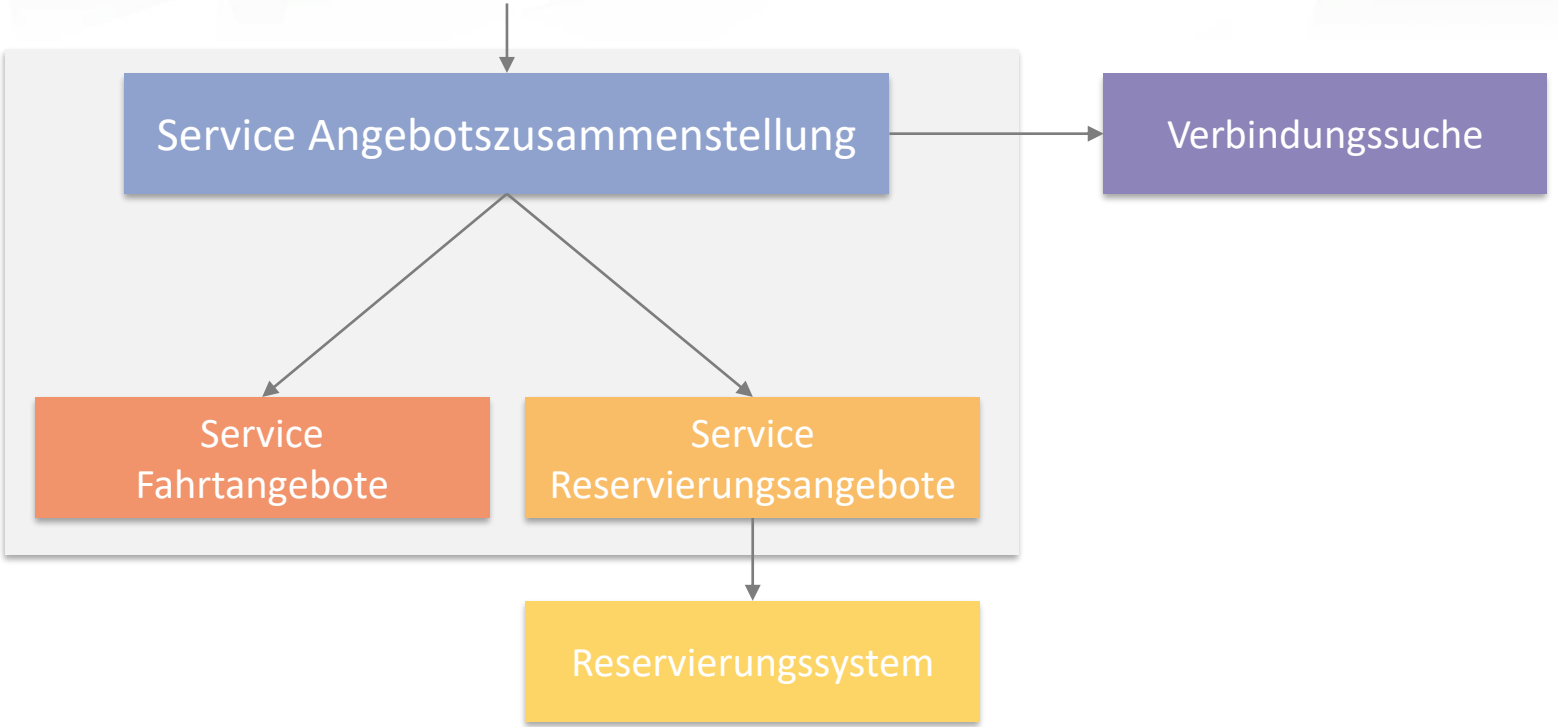
andrena
OBJECTS

Motivation: Ein Erfahrungsbericht

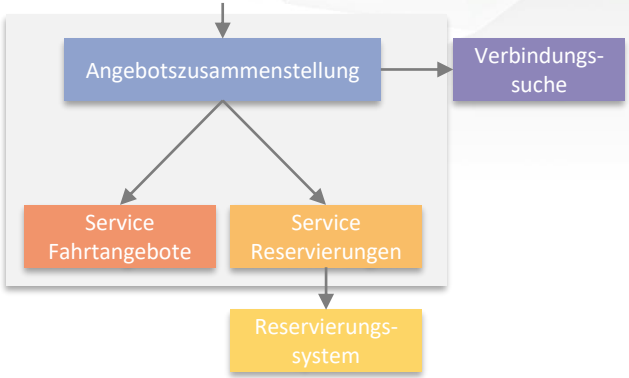
Warum haben wir uns entschieden, den reaktiven Weg zu gehen?

Was ist der „reaktive Weg“ überhaupt?

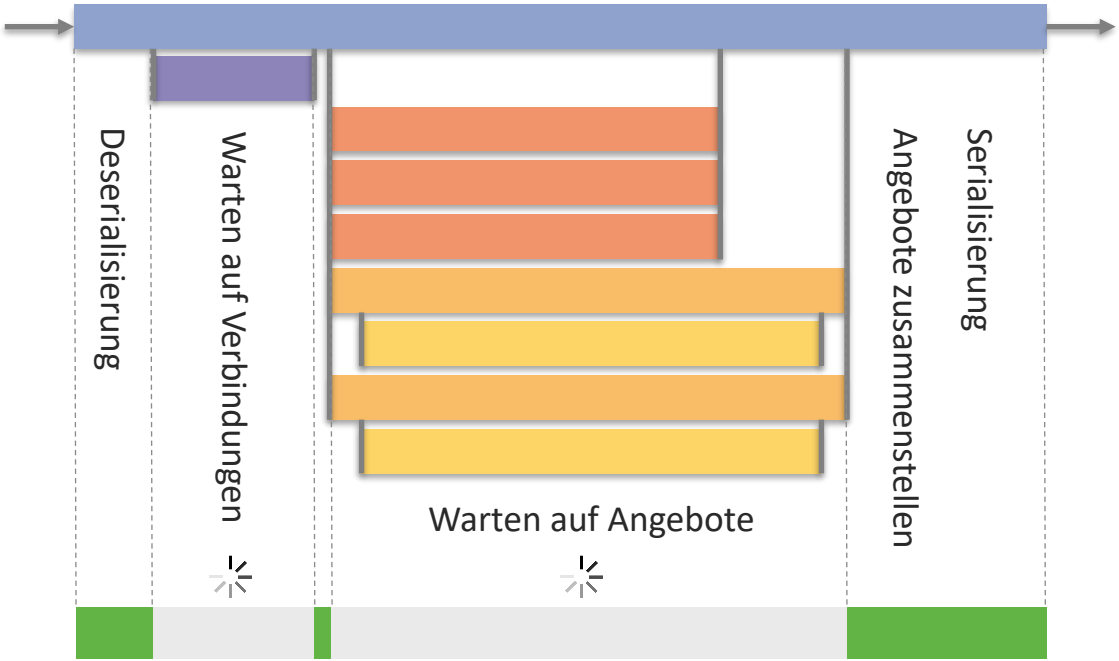
Service-Architektur Angebotsberechnung



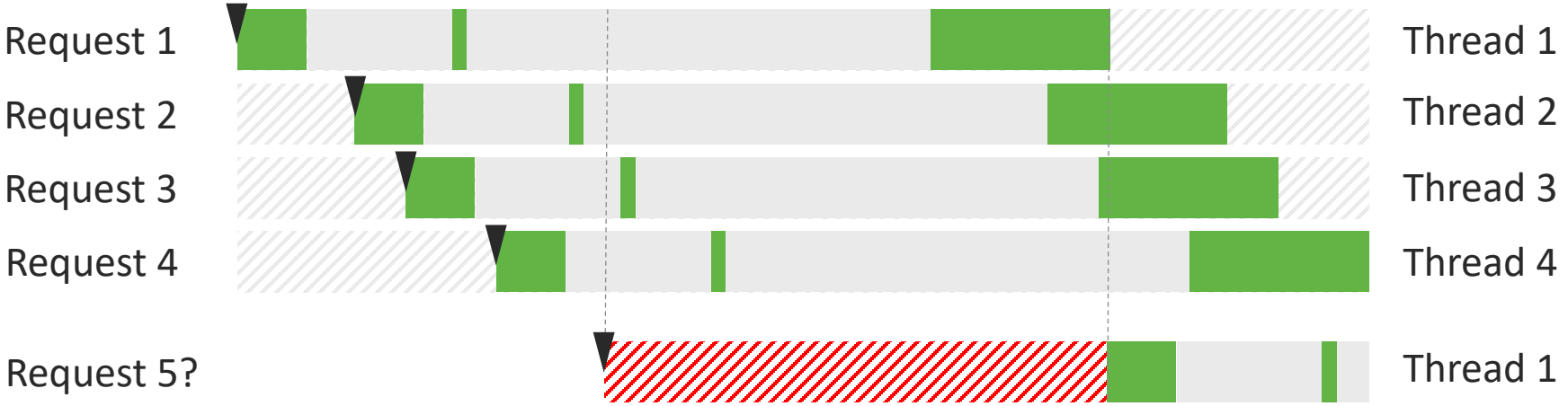
Service-Architektur Angebotsberechnung



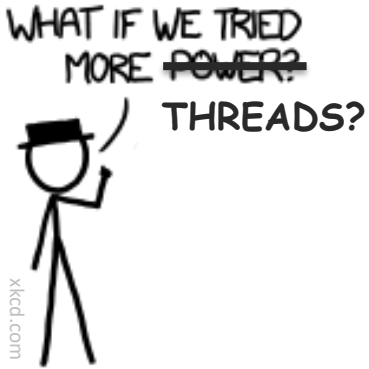
Call Trace



Thread-per-Request-Modell



Thread-per-Request-Modell




```
@Bean
ExecutorService executorService() {
    return Executors.newFixedThreadPool( nThreads: 600);
}
```

Thread-per-Request-Modell

- Großer Threadpool
- Hohe CPU-Last durch Polling + Context Switches
- Trotzdem nur begrenzter Durchsatz

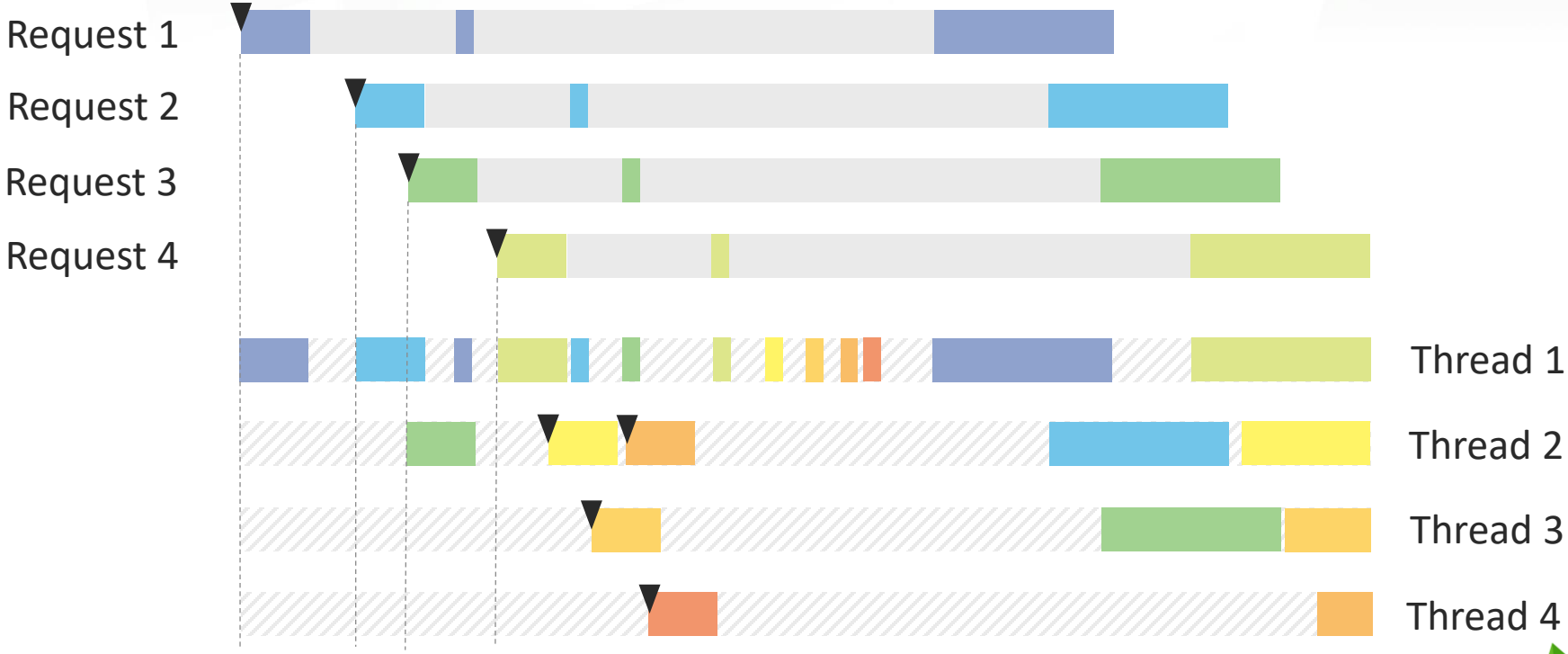
```
@Bean
ExecutorService executorService() {
    return Executors.newFixedThreadPool( nThreads: 600);
}
```

Je 1 Thread wird benötigt für jeden
- eingehenden Request *
- (parallelen) ausgehenden Request


Threads warten/blockieren die meiste Zeit

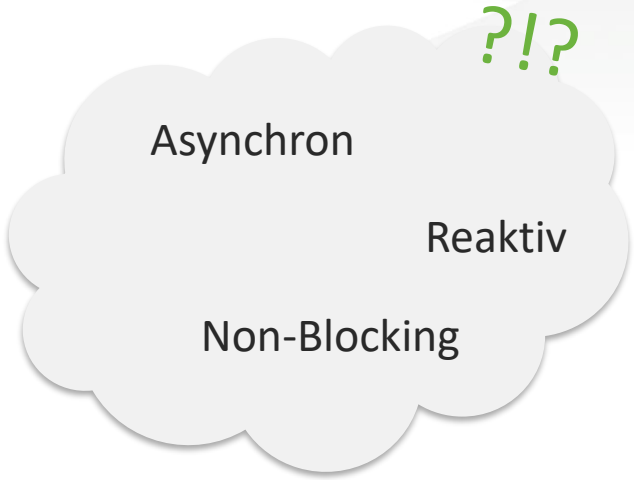

Gibt es ein geeigneteres Modell?

Event Loop-Modell



Event Loop-Modell

- Kleiner Threadpool
  - Anzahl CPUs
- Tatsächlicher Workload bestimmt CPU-Last
- Durchsatz unabhängig von Threadpool-Größe




```
final int DEFAULT_POOL_SIZE =
  Optional.ofNullable(System.getProperty("reactor.schedulers.defaultPoolSize"))
                  .map(Integer::parseInt)
                  .orElseGet(() -> Runtime.getRuntime().availableProcessors());
```

github.com/reactor

andrena
OBJECTS

Reaktive Programmierung: Asynchrone Codeausführung in Java

Webflux + Reactor gegen den Rest der Welt?

Problem

```
public String getSynchronous() {
   Response response = httpClient.sendRequest(URL);
   return processResponse(response);
}
```

sendRequest und *processResponse* werden im gleichen Thread ausgeführt

Lösung?

```
public void getAsynchronous(Consumer<String> callback) {
   httpClient.sendRequest(URL, response ->
      callback.accept(processResponse(response)));
}
```

commons.wikimedia.org/wiki/File:Mardi_Gras_Headed_for_Hell,_Bourbon_Street_1989_01.jpg

Sprachunterstützung *

C# 5 [2012]

```
async Task<String> GetAsynchronous()
{
   Response response = await httpClient.SendRequest(URL);
   return ProcessResponse(response);
}
```

JS ES6 [2017]

```
async function getAsynchronous() {
   const response = await httpClient.sendRequest(URL);
   return processResponse(response);
}
```

Java 9 + Reactor [2017]

```
Mono<String> getAsynchronous() {
   return httpClient.sendRequest(URL)
      .map(this::processResponse);
}
```

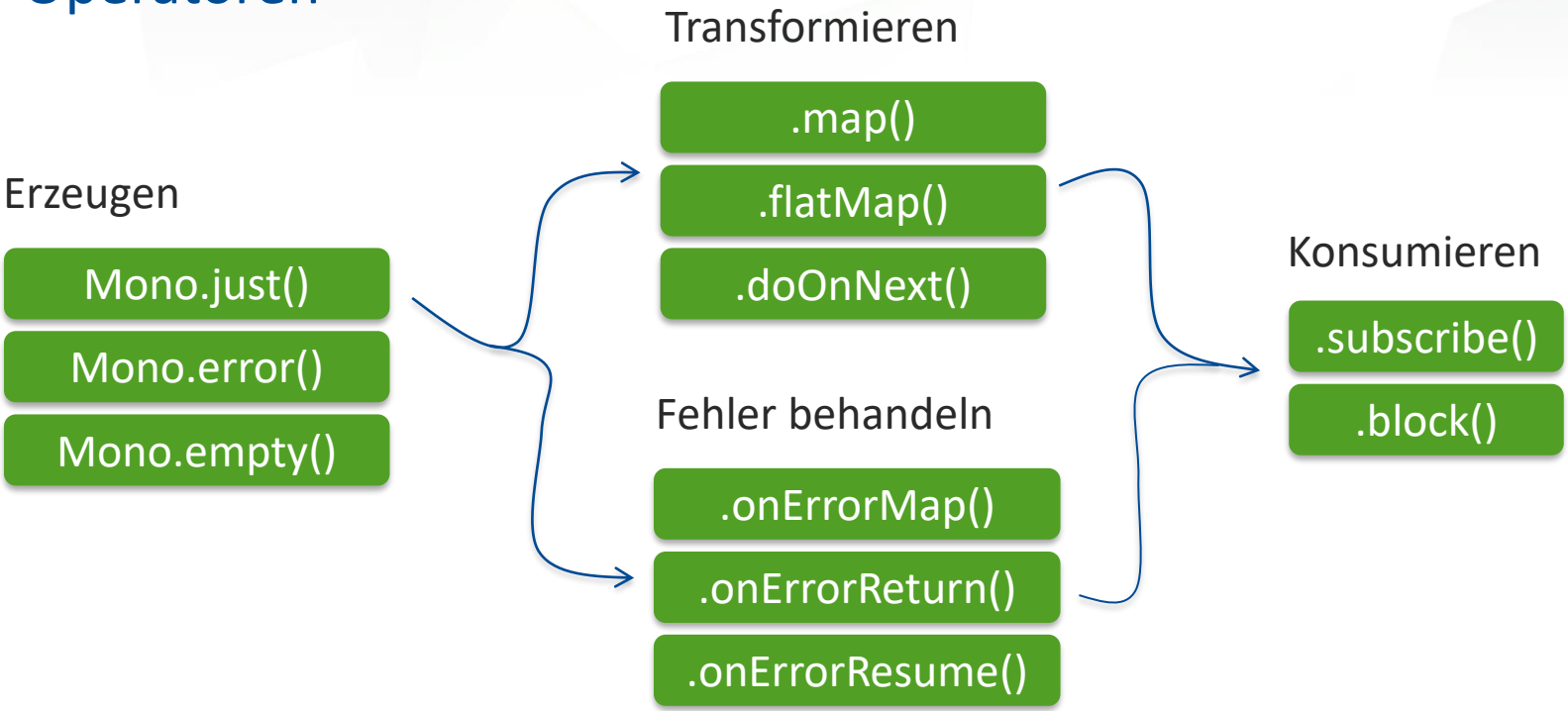
Kotlin 1.3 [2018]

```
suspend fun getAsynchronous(): String {
   val response = httpClient.sendRequest(URL)
   return processResponse(response)
}
```

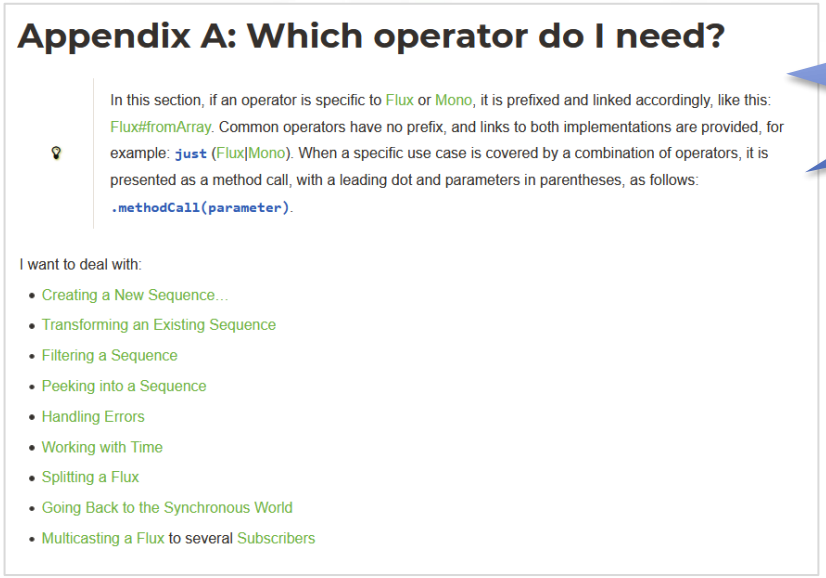
Project Reactor

- Implementierung der *Reactive Streams API* in Java 9
- Stellt zentrale asynchrone Stream-Datentypen bereit
  - **Mono** (0..1)
  - **Flux** (0..n)

Operatoren



Operatoren



Appendix A: Which operator do I need?

In this section, if an operator is specific to Flux or Mono, it is prefixed and linked accordingly, like this: Flux#fromArray. Common operators have no prefix, and links to both implementations are provided, for example: `just` (Flux|Mono). When a specific use case is covered by a combination of operators, it is presented as a method call, with a leading dot and parameters in parentheses, as follows: `.methodCall(parameter)`.

I want to deal with:

- Creating a New Sequence…
- Transforming an Existing Sequence
- Filtering a Sequence
- Peeking into a Sequence
- Handling Errors
- Working with Time
- Splitting a Flux
- Going Back to the Synchronous World
- Multicasting a Flux to several Subscribers

projectreactor.io/docs/core/release/reference/#which-operator

Spring Webflux

- Reaktiver Stack für Spring (Boot) Applications
- Basiert auf dem Event Loop-Modell

	API	Web Servers	HTTP Client
WebMVC	Servlet API	Tomcat (default) Jetty Undertow	RestTemplate
Webflux	Reactive API	Netty (default) Tomcat Jetty Undertow	WebClient

Spring Webflux

HTTP Server

```
@GetMapping(value = "/hello/world")
public Mono<String> getHelloWorld() {
   return Mono.just("Hello world!");
}
```

HTTP Client

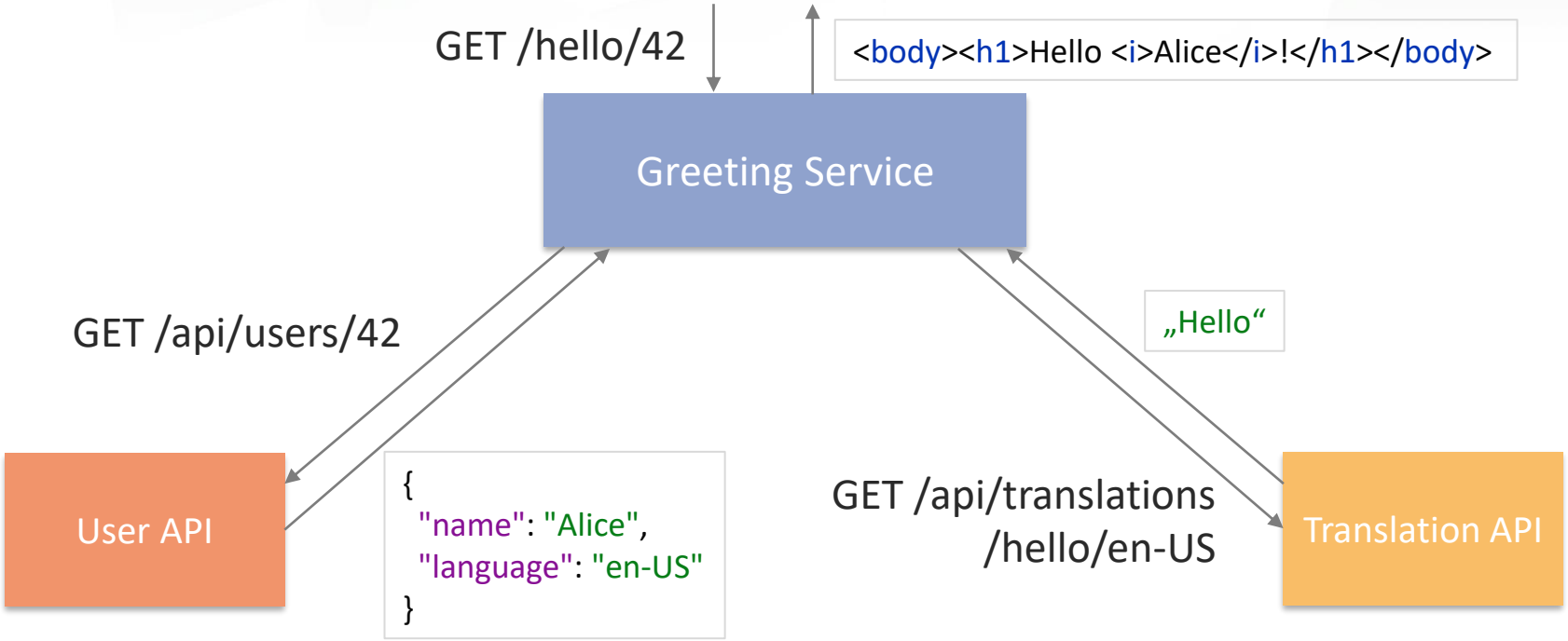
```
Mono<String> responseString = webClient.get()
      .uri(URL)
      .retrieve()
      .bodyToMono(String.class);
```


andrena
OBJECTS

Live Demo:
Hello World Enterprise Edition™

Hello World goes Microservice

Service-Architektur

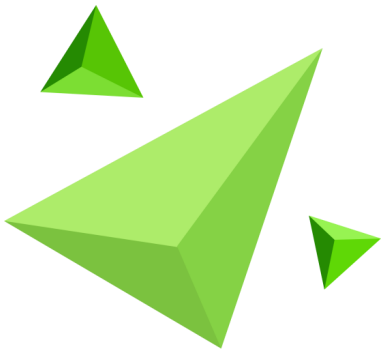


github.com/jnagykuhlen/WebfluxDemo

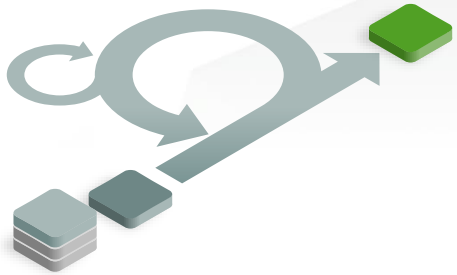
andrena
OBJECTS

Learnings

Die Zukunft ist reaktiv. Und jetzt?

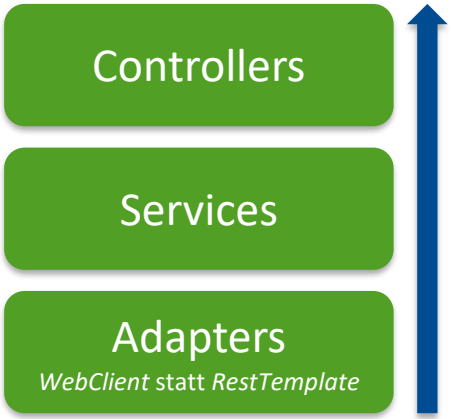


Best Practices



Ziel: Lieferfähigkeit zu jedem Zeitpunkt sicherstellen

- Klasse für Klasse umstellen
  - *block()* nutzen, um reaktive Komponenten in die synchrone Anwendung zu integrieren



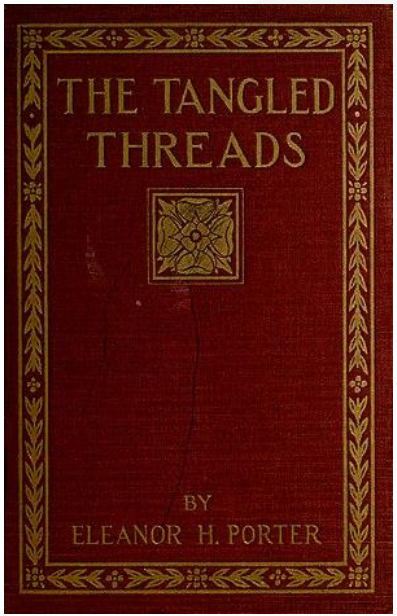
Herausforderungen

Thread ≠ Request

- Klasse *ThreadLocal*
- Spring Request-Scoped Beans

```
ThreadLocal<PerRequestData> perRequestData;
```

```
@Scope(WebApplicationContext.SCOPE_REQUEST)
```



commons.wikimedia.org/wiki/File:Cover--The_tangled_threads.jpg

Lösung: Zustand explizit durchreichen

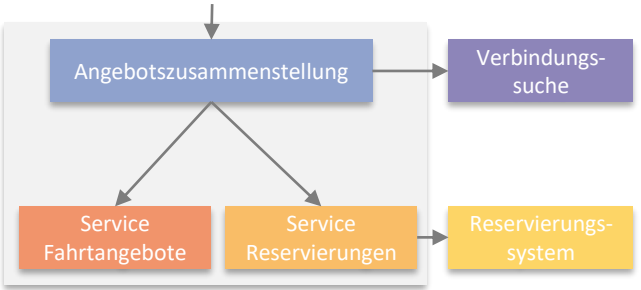
Lohnt sich der Umstieg?

Ja, wenn

- der Durchsatz ein Bottleneck darstellt
- die CPU-Last (zu) hoch ist
- viele ausgehende Requests gesendet werden
- ein neuer Service aufgebaut wird

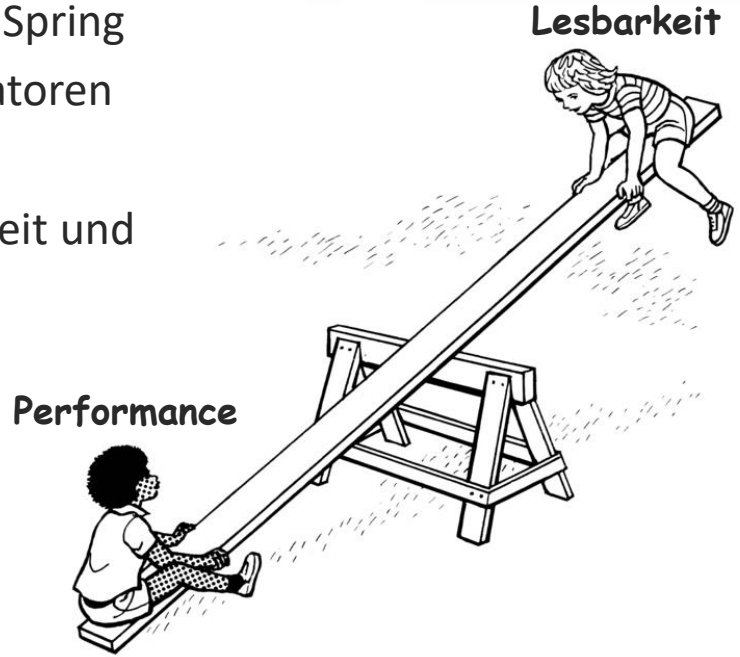
Nein, wenn

- Performance kein Problem darstellt
- wenig Wartezeiten durch I/O entstehen
- im Team keine Zeit für technische Umbauten zur Verfügung steht

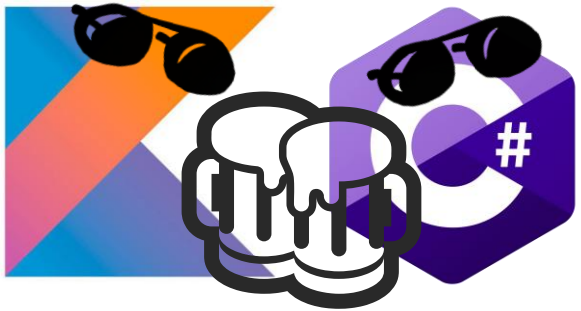


Fazit

- Gute Unterstützung für reaktive Webservices in Spring
- Reactor ist ausgereift und bietet sinnvolle Operatoren
- Leider keine Sprachunterstützung in Java
  - Tradeoff zwischen Performance/Skalierbarkeit und Lesbarkeit/einfachem Debugging
  - Steilere Lernkurve



commons.wikimedia.org/wiki/File:See-Saw_(PSF).png

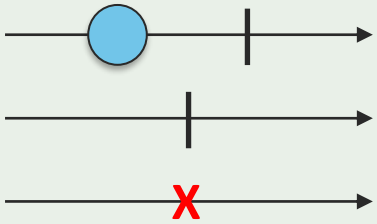
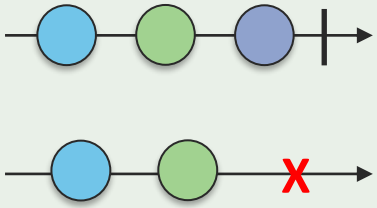


Vielen Dank fürs Zuhören!

Anhang

Deep Dive Reactor & Lasttest-Ergebnisse

Project Reactor

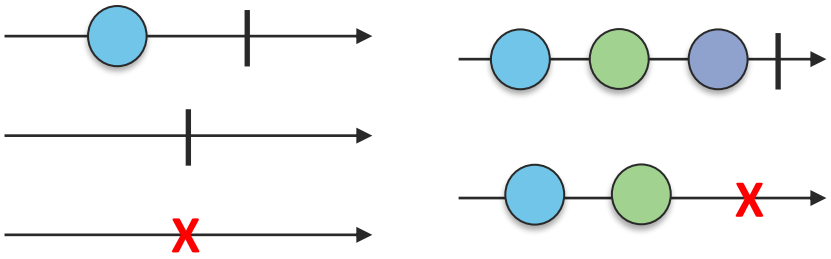
Mono	0..1 Elemente	
Flux	0..n Elemente	

Project Reactor

- Mono und Flux sind *Publisher*
- Werden erst ausgewertet, wenn *Subscriber* registriert sind

```
package org.reactivestreams;

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



```
package org.reactivestreams;

public interface Subscriber<T> {
   void onSubscribe(Subscription subscription);
   void onNext(T item);
   void onError(Throwable error);
   void onComplete();
}
```

Normalerweise abstrahiert hinter bekannten Streaming-Operationen

Operationen: Brücke zur synchronen Welt

Operation	Beispielcode
Erzeugen eines *Mono* aus einem gegebenen Wert	Mono<String> myMono = Mono.*just*("Hello world");
Erzeugen eines leeren *Mono*	Mono<String> myMono = Mono.*empty*();
Erzeugen eines fehlerhaften *Mono*	Mono<String> myMono = Mono.*error*(new RuntimeException("I am Error"));
(Synchron) auf das Ergebnis eines *Mono* warten	String result = myMono.block();

Operationen: Komposition

Operation	Synchroner Beispielcode	Reaktiver Beispielcode

Transformation	String myValue = "hello";
	return process(myValue);
		Mono<String> myMono = Mono.*just*("hello");
		return myMono.map(this::processSync);
		
		Mono<String> myMono = Mono.*just*("hello");
		return myMono.flatMap(this::processAsync);

Seiteneffekte	String myValue = "hello"; System.*out*.println(myValue); return myValue;	Mono<String> myMono = Mono.*just*("hello"); return myMono.doOnNext(System.*out*::println);

Operationen: Fehlerbehandlung

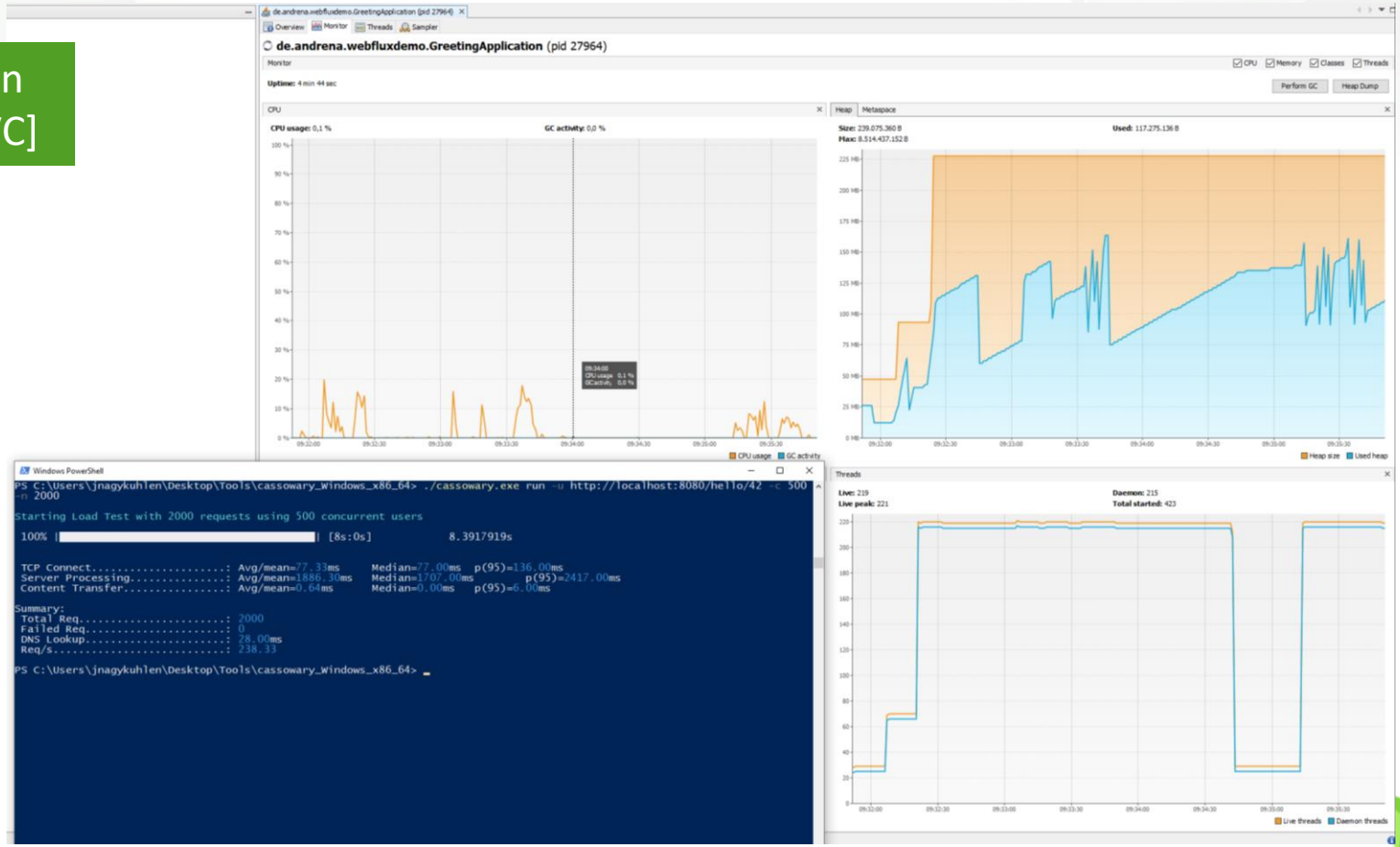
Operation	Synchroner Beispielcode	Reaktiver Beispielcode

Exception wrappen	try {
	  return process(myValue);
	} catch (Exception ex) {
	  throw new MyException(ex);
	}
		return myMono
		  .map(this::process)
		  .onErrorMap(Exception.class, ex ->
		  new MyException(ex));

Standardwert zurückliefern	try {
	  return process(myValue);
	} catch (Exception ex) {
	  return *DEFAULT*;
	}
		return myMono
		  .map(this::process)
		  .onErrorReturn(Exception.class, *DEFAULT*);

Komplexere Behandlungslogik	try {
	  return process(myValue);
	} catch (Exception ex) {
	  System.*out*.println(ex);
	  return processError(ex);
	}
		return myMono
		  .map(this::process)
		  .onErrorResume(Exception.class, ex -> {
		  System.*out*.println(ex);
		  return Mono.*just*(processError(ex));
		  });

Synchron [WebMVC]



Reaktiv [Webflux]