

# KOROUTINEN MIT KOTLIN

JavaLand, 2018

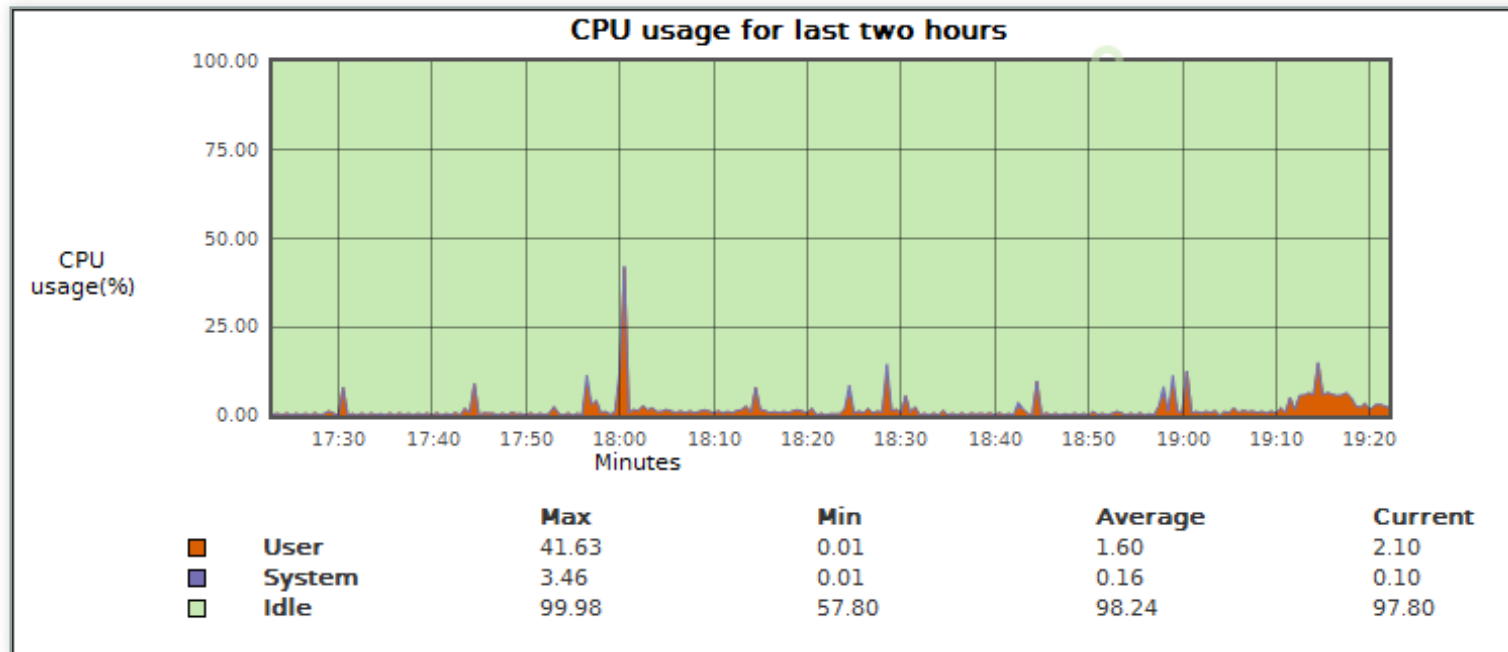
@RenePreissel

<https://github.com/rpreissel/kotlin-coroutine.git>

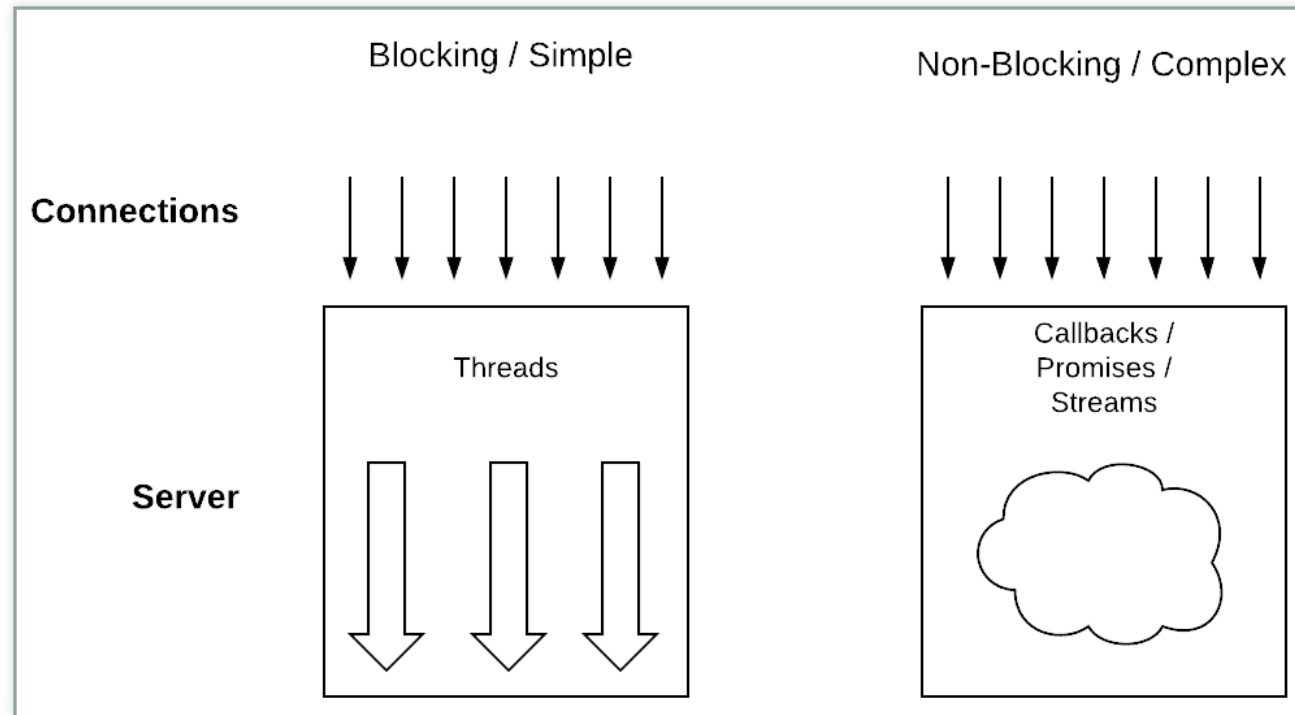
# INHALT

- Warum Koroutinen?
- Umsetzung von Koroutinen in Kotlin
- Asynchrone Kommunikationsmuster mit Koroutinen

# WARUM?



# ENTSCHEIDUNG?



# BEISPIEL



```
val collage = createCollage("turtle", 20)
```

# ARBEITEN MIT THREADS

```
fun loadOneImage(query: String): BufferedImage {  
    val url = requestImageUrl(query)  
    val image = requestImageData(url)  
    return image  
}
```

# CALLBACKS

```
fun loadOneImage(  
    query: String,  
    onFailure: OnFailure = {},  
    onSuccess: OnSuccess<BufferedImage>  
): Unit {  
    requestImageUrl(query, onFailure) { url ->  
        requestImageData(url, onFailure) { image ->  
            onSuccess(image)  
        }  
    }  
}
```

# FUTURES / PROMISES

```
fun loadOneImage(query: String): CompletableFuture<BufferedImage> {  
    return requestImageUrl(query)  
        .thenCompose{ requestImageData(it) }  
}
```



# KOMPLEXERES BEISPIEL - KOLLAGE

```
fun createCollage(query: String, count: Int): BufferedImage {  
    val urls = requestImageUrls(query, count)  
    val images = urls.map { requestImageData(it) }  
    val newImage = combineImages(images)  
    return newImage  
}
```

# KOLLAGE MIT CALLBACKS

```
fun createCollage(query: String, count: Int, onSuccess: OnSuccess<BufferedImage>) {  
    requestImageUrls(query, count) { urls ->  
        fun loadImages(  
            urlIter: Iterator<String>,  
            retrievedImages: List<BufferedImage>  
        ) {  
            if (urlIter.hasNext()) {  
                requestImageData(urlIter.next()) { image ->  
                    loadImages(urlIter, retrievedImages + image)  
                }  
            } else {  
                onSuccess(combineImages(retrievedImages))  
            }  
        }  
        loadImage(urls.iterator(), listOf())  
    }  
}
```

# KOLLAGE MIT FUTURES


```
fun createCollage(query: String, count: Int): CompletableFuture<BufferedImage> {  
    return requestImageUrls(query, count)  
        .thenCompose {urls ->  
            val startFuture = completedFuture<List<BufferedImage>>(listOf())  
            urls.fold(startFuture) { lastFuture, url ->  
                lastFuture.thenCompose { images ->  
                    requestImageData(url).thenApply { image ->  
                        images + image  
                    }  
                }  
            }  
        }.thenApply(::combineImages)  
}
```

# KOLLAGE MIT KOROUTINEN

```
suspend fun requestImageUrls(query: String, count: Int = 20): List  
suspend fun requestImageData(imageUrl: String): BufferedImage
```

```
suspend fun createCollage(query: String, count: Int): BufferedImage {  
    val urls = requestImageUrls(query, count)  
    val images = urls.map { requestImageData(it) }  
    val newImage = combineImages(images)  
    return newImage  
}
```

# EINFACHE ASYNCHRONE SEQUENZEN



**Roman Elizarov**  
@relizarov

Folge ich

▼

4/ In Java we learn that our async methods should return `Single<T>` or `CompletableFuture<T>` or some other type of the future. We painstakingly learn all the ceremony around it like `flatMap` to imitate sequential composition. With Kotlin coroutines this is no longer needed.

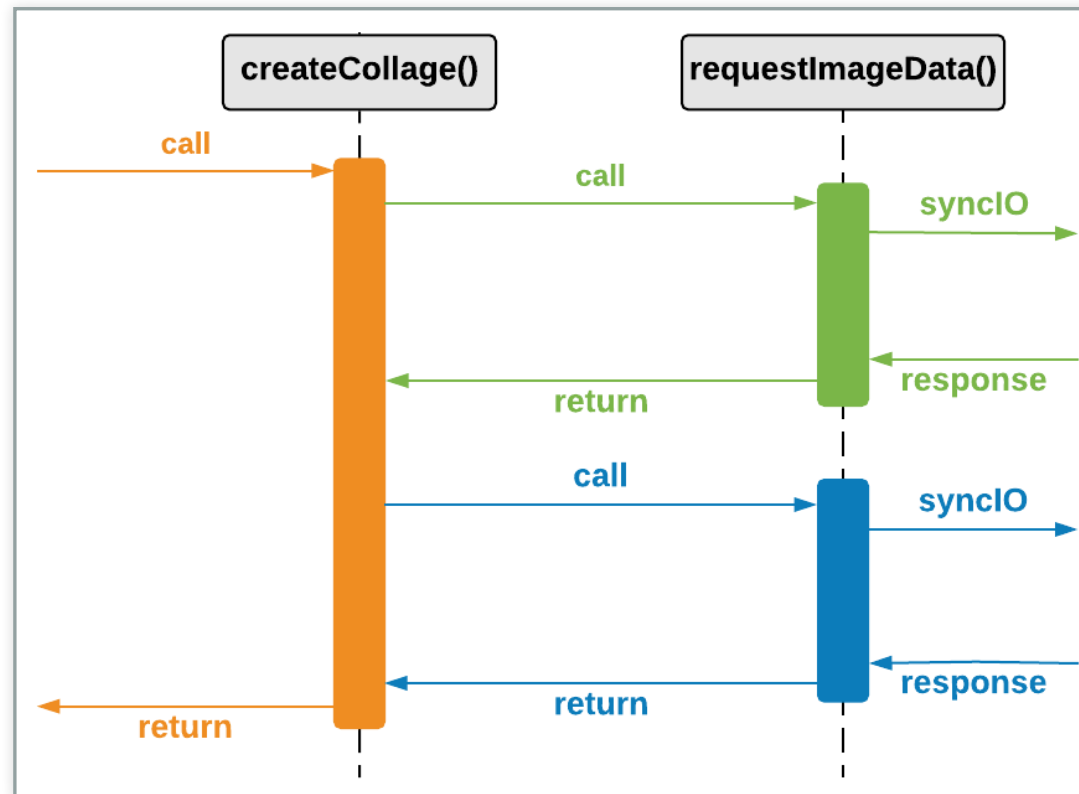
🌐 Original (Englisch) übersetzen

23:19 - 7. März 2018

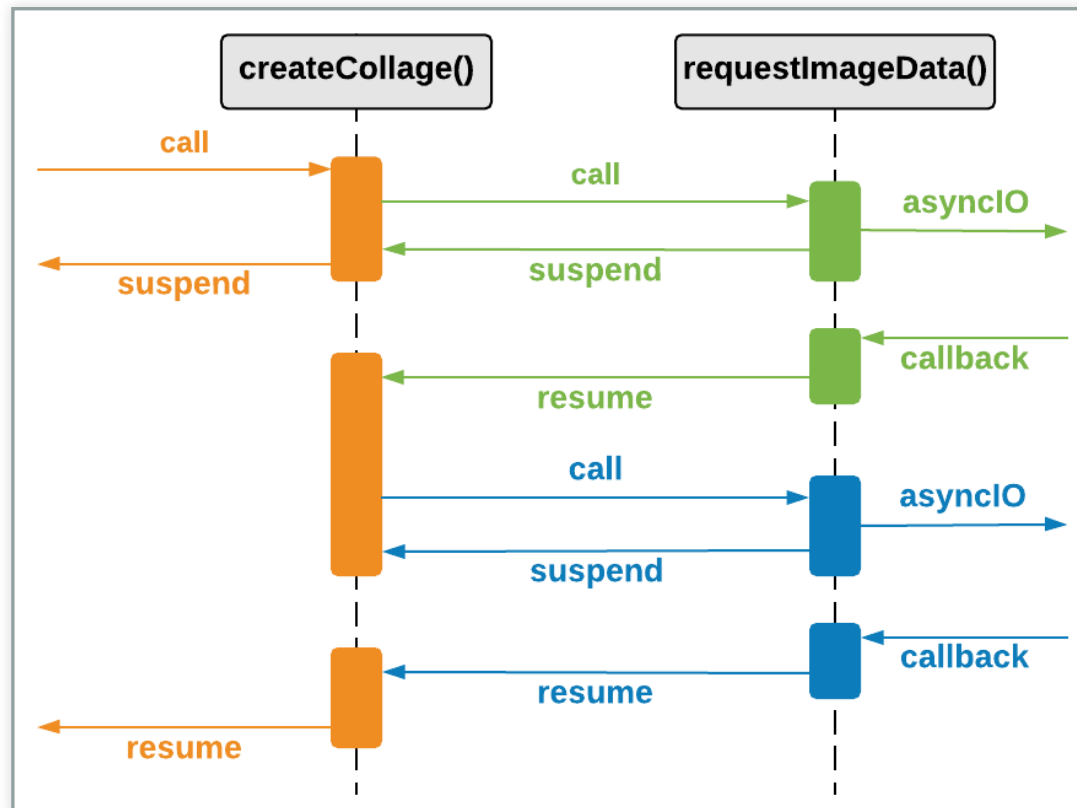
# KOROUTINEN

- Melvin Conway 1963
- Kooperative Übergabe des Kontrollflusses
- Koroutinen sind *sequentiell* per Default

# FUNKTIONEN / ROUTINEN



# KOROUTINEN





# STACKLESS VS STACKFULL

- Stackless: Suspendierungen sind nur **direkt** in Koroutinen möglich
- Stackfull: Suspendierungen sind **überall** möglich
- Kotlin implementiert **stackless** Koroutinen

# CONTINUATIONS

```
suspend fun createCollage(query: String, count: Int): BufferedImage {  
    val urls = requestImageUrls(query, count) //Label 0  
    val images = mutableListOf<BufferedImage>() //Label 1  
    for (index in 0 until urls.size) {  
        val image = requestImageData(urls[index])  
        images += image //Label 2  
    }  
    val newImage = combineImages(images)  
    return newImage  
}
```

## Continuation

Label

2

Data

urls

A, B, C

images



index

2







# KOTLIN COMPILER

Aus:

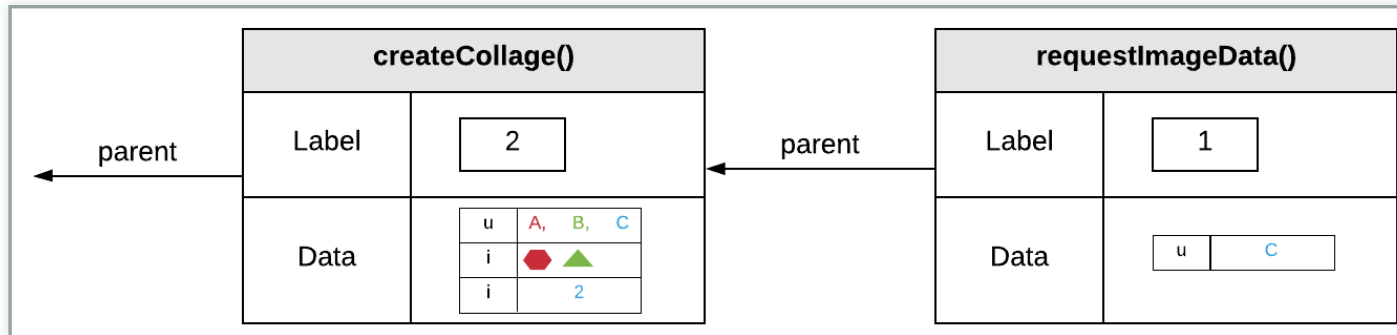
```
suspend fun createCollage(  
    query: String, count: Int  
): BufferedImage
```

wird:

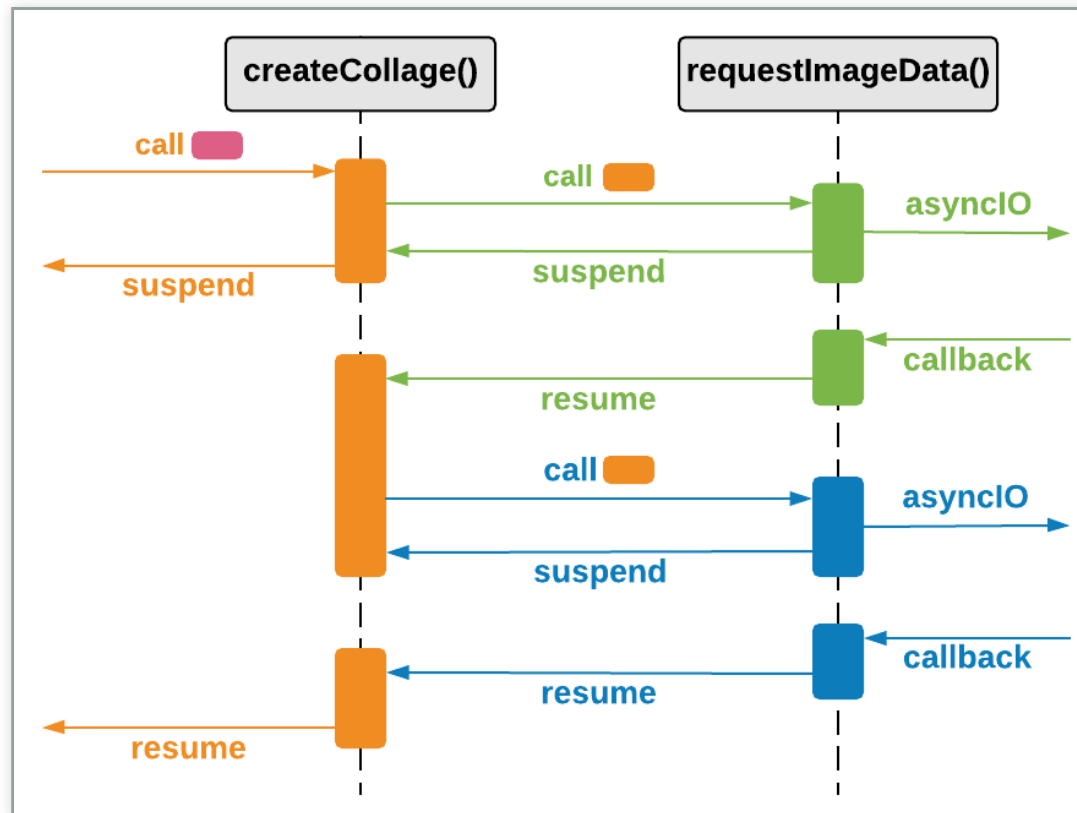
```
fun createCollage(  
    query: String, count: Int,  
    parentContinuation: Continuation<BufferedImage>  
): Any // BufferedImage | COROUTINE_SUSPENDED {  
    val cont = CoroutineImpl(parentContinuation) //Implements Continuation
```

CoroutineImpl: Continuation							
Label	<div>2</div>						
Data	<table><tr><td>urls</td><td>A, B, C</td></tr><tr><td>images</td><td> </td></tr><tr><td>index</td><td>2</td></tr></table>	urls	A, B, C	images	 	index	2
urls	A, B, C						
images	 						
index	2						
Parent	<div>Parent-Continuation</div>						
resume(value)							
resumeWithException(exception)							

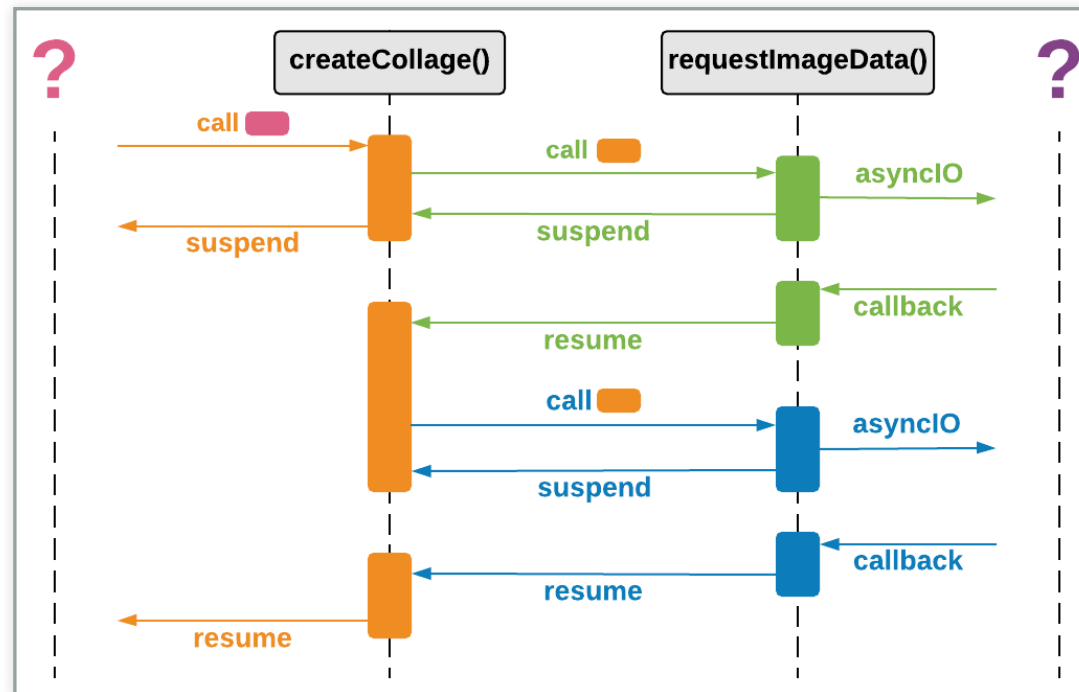
# CONTINUATIONS-STACK



# KOROUTINEN



# EINSTIEG UND ABSPRUNG?





# BUILDER - EINSTIEG IN KOROUTINEN

```
//Startet die Koroutine und "blockiert" den aktuellen Thread
val collage = runBlocking {
    createCollage("dogs", 20)
}

//Startet die Koroutine und setzt den aktuellen Thread fort
val job = launch {
    val collage = createCollage("dogs", 20)
    ImageIO.write(collage, "png", FileOutputStream("dogs.png"))
}

//Stoppt die Koroutine
job.cancel()
```

# COROUTINECONTEXT

```
//Den Fork-Join-Pool für die Koroutine nutzen
val collage = runBlocking(CommonPool) {
    createCollage("dogs", 20)
}

//Einen eigenen Thread-Pool für die Koroutine nutzen
val fixedThreadPoolContext = newFixedThreadPoolContext(1, "collage")
val job = launch(fixedThreadPoolContext) {
    val collage = createCollage("dogs", 20)

    // Wechsel in den UI-Thread und zurück
    withContext(UI) {
        ImageIO.write(collage, "png", FileOutputStream("dogs.png"))
    }
}
```

# INTEGRATION MIT ASYNCHRONEN LIBRARIES

```
JerseyClient.pixabay("q=$query&per_page=$count").request()  
    .async()  
    .get(object : InvocationCallback<String> {  
        override fun completed(response: String) {  
            val urls = JsonPath.read<List<String>>(response, "$..previewURL")  
            ...  
        }  
  
        override fun failed(throwable: Throwable) {  
            ...  
        }  
    })
```

# ABSPRUNG ZU ASYNCHRONEN LIBRARIES

```
suspendCoroutine<List<String>> { continuation ->
    JerseyClient.pixabay("q=$query&per_page=$count").request().async()
        .get(object : InvocationCallback<String> {
            override fun completed(response: String) {
                val urls = JsonPath.read<List<String>>(response, "$..previewURL")
                continuation.resume(urls)
            }

            override fun failed(throwable: Throwable) {
                continuation.resumeWithException(throwable)
            }
        })
    })
}
```

# ASYNCHRONE MUSTER / KONZEPTE IN KOTLIN

- Sequential by default
- Asynchronous explicitly
- Libraries not language

# SEQUENTIAL BY DEFAULT

```
suspend fun createCollage(query: String, count: Int): BufferedImage {  
    val urls = requestImageUrls(query, count)  
    val images = urls.map { requestImageData(it) }  
    val newImage = combineImages(images)  
    return newImage  
}
```

# ASYNC / AWAIT EXPLIZIT

```
suspend fun createCollageAsyncAwait(
    query: String, count: Int
): BufferedImage {
    val urls = requestImageUrls(query, count)
    val deferredImages: List<Deferred<BufferedImage>> = urls.map {
        async {
            requestImageData(it)
        }
    }

    val images: List<BufferedImage> = deferredImages.map { it.await() }

    val newImage = combineImages(images)
    return newImage
}
```

# COMMUNICATING SEQUENTIAL PROCESSES / CSP

- Concurrency Theory
- Pragmatisch:  
Kommunikation per Nachrichten über Kanäle
- In Kotlin: `Channel`
- Ein `Channel` entspricht einer `BlockingQueue`  
nur ohne blockieren



# NACHRICHTEN SENDEN

```
suspend fun retrieveImages(query: String, channel: SendChannel<BufferedImage>) {  
    while (true) {  
        val url = requestImageUrl(query)  
        val image = requestImageData(url)  
        channel.send(image)  
        delay(2, TimeUnit.SECONDS)  
    }  
}
```

# NACHRICHTEN EMPFANGEN

```
suspend fun createCollage(channel: ReceiveChannel<BufferedImage>, count: Int) {  
    var imageId = 0  
    while (true) {  
        val images = (1..count).map {  
            channel.receive()  
        }  
        val collage = combineImages(images)  
        ImageIO.write(collage, "png", FileOutputStream("image-${imageId++}.png"));  
    }  
}
```

# CHANNEL

```
val channel = Channel<BufferedImage>()
launch(Unconfined) {
    retrieveImages("dogs", channel)
}

launch(Unconfined) {
    retrieveImages("cats", channel)
}

launch(Unconfined) {
    createCollage(channel, 4)
}
```

# UNCONFINED UND RENDEZVOUS

```
"jersey-client-async-executor-2@3321" prio=5 tid=0x13 nid=NA runnable
  java.lang.Thread.State: RUNNABLE
    at ...CSPChannelKt.createCollage(CSPChannel.kt:47)
    at ...CSPChannelKt$createCollage$1.doResume(CSPChannel.kt:-1)
    at ...CoroutineImpl.resume(CoroutineImpl.kt:54)
    at ...ResumeModeKt.resumeMode(Dispatched.kt:87)
    at ...DispatchedKt.dispatch(Dispatched.kt:193)
    at ...AbstractContinuation.afterCompletion(AbstractContinuation.kt:86)
    at ...JobSupport.completeUpdateState$kotlinx_coroutines_core(Job.kt:719)
    at ...CancellableContinuationImpl.completeResume(CancellableContinuation.kt:2
    at ...AbstractChannel$ReceiveElement.completeResumeReceive(AbstractChannel.kt
    at ...AbstractSendChannel.offerInternal(AbstractChannel.kt:64)
    at ...AbstractSendChannel.offer(AbstractChannel.kt:186)
    at ...AbstractSendChannel.send(AbstractChannel.kt:180)
    at ...CSPChannelKt.retrieveImages(CSPChannel.kt:59)
```

# ACTOR

- Aktoren sind nebenläufige Einheiten
- Kommunizieren nur über Nachrichten
- Arbeiten alle Nachrichten sequentiell ab
- Verwalten eigenen Zustand

# ACTOR - NACHRICHTEN

```
sealed class PixabayMsg
data class RequestImageUrlMsg(
    val query: String,
    val resultChannel: SendChannel<String>
) : PixabayMsg()
```

# ACTOR - VERHALTEN

```
val PixabayActor: SendChannel<PixabayMsg> = actor<PixabayMsg> {  
    for (msg in channel) {  
        when (msg) {  
            is RequestImageUrlMsg -> msg.apply {  
                resultChannel.send(requestImageUrl(query))  
            }  
        }  
        delay(100)  
    }  
}
```

# ACTOR - BENUTZEN

```
suspend fun retrieveImages(query: String, channel: SendChannel<BufferedImage>) {  
    val resultChannel = Channel<String>(1)  
    val requestImageUrlMsg = RequestImageUrlMsg(query, resultChannel)  
    while (true) {  
        PixabayActor.send(requestImageUrlMsg)  
        val url = resultChannel.receive()  
        val image = requestImageData(url)  
        channel.send(image)  
        delay(2, TimeUnit.SECONDS)  
    }  
}
```



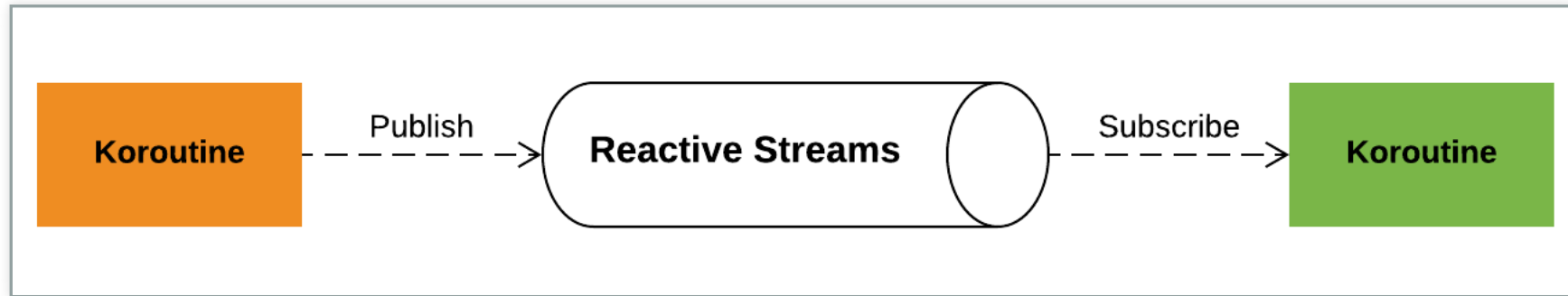
# ACTOR - EINSCHRÄNKUNGEN

- Kein Supervisor bzw. keine Child-Hierarchie
- Keine implizite Fehlerbehandlung
- Keine Verteilung
- Vollständigere Actor Implementierung: <http://proto.actor>

# REACTIVE STREAMS

- Nachrichtenbasierend
- Asynchron / Nicht-Blockierend
- Unterstützung von Back-Pressure
- API in Java 9 enthalten
- Verschiedene Implementierungen:  
Reactor, RxJava, Akka Streams

# REACTIVE STREAMS UND KOROUTINEN



# VON SUSPEND ZUM REAKTIVEM STREAM (PUBLISH)

```
fun retrieveImagesAsFlux(
    query: String,
    batchSize: Int
): Flux<BufferedImage> = flux {
    while (isActive) {
        val urls = requestImageUrls(query, batchSize)
        for (url in urls) {
            val image = requestImageData(url)
            send(image)
        }
        delay(2, TimeUnit.SECONDS)
    }
}
```

# VOM REAKTIVEM STREAM ZU SUSPEND (SUBSCRIBE)

```
suspend fun requestImageUrls(query: String, count: Int = 20): List<String> {  
    return ReactorClient  
        .pixabay("q=$query&per_page=$count")  
        .retrieve()  
        .bodyToMono<String>()  
        .map { response ->  
            JsonPath.read<List<String>>(response, "$..previewURL")  
        }.awaitSingle()  
}
```

# ZUSAMMENFASSUNG

- **suspend** konvertiert Funktionen zu Koroutinen
- Sequential by Default / Asynchronous explicitly
- Asynchronen Kommunikationsmustern als Library
- Einfache Integration in vorhandene asynchrone APIs
- Stackless: Suspendierungen nur in Koroutinen möglich  
(Red/Blue Code Problem)
- Tooling / Debugging muss noch verbessert werden

# AUSBLICK

- Stackfull-Koroutinen durch **Quasar**:  
<https://github.com/puniverse/quasar>
- Oder durch Project **Loom**:  
<http://cr.openjdk.java.net/~rpressler/loom/Loom-Proposal.html>  
<https://www.youtube.com/watch?v=fpyub8fbrVE>

# FRAGEN?

@RenePreissel

rene.preissel@etosquare.de

www.etosquare.de

<https://github.com/rpreissel/kotlin-coroutine.git>