GEDOPLAN

Reactive Programming
Königs- oder Irrweg?

GEDOPLAN GmbH

Dirk Weil

Dirk Weil

- GEDOPLAN GmbH, Bielefeld
  - GEDOPLAN IT Consulting
    Softwareentwicklung, Beratung, Konzepte, Reviews
  - GEDOPLAN IT Training
    Java, JEE, Tools u.v.a.m. in Berlin, Bielefeld, on-site
- JEE seit 1999
- Speaker und Autor

Darum geht's

Platform Thread
Context Propagation
Loom
Virtual Thread
Multitasking
Thread Pool
CompletionStage

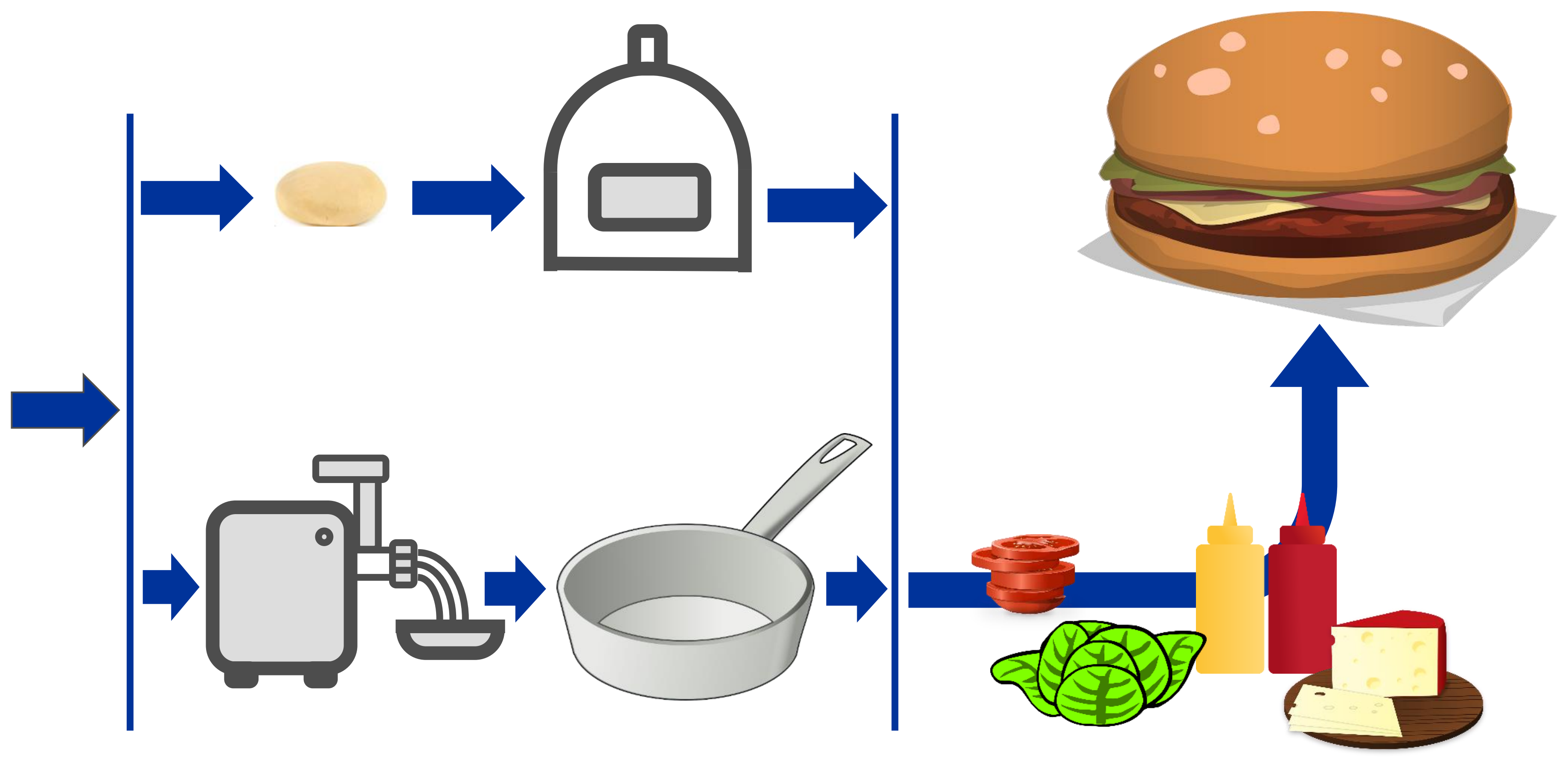
Das Problem zur Lösung

Burger-Service: 1) synchron

```java
@GET
@Produces(MediaType.APPLICATION_JSON)
public List<String> getBurger(…) {

  Bun bun = bakeBun(supplyBunDough(…));

  Patty patty = fryPattie(supplyPattyMeat(…));

  List<String> parts = List.of(
    bun.getUpperHalf(),
    getSauce(),
    getTomato(),
    getCheese(),
    pattie.toString(),
    getSalad(),
    bun.getLowerHalf());

  return parts;
```

Demo

Multitasking



Threads in Java

- **`java.lang.Thread`**
- führt ein **`Runnable`** aus

```
// Start doSomething in new (platform) thread
new Thread(this::doSomething).start();
```

- Best practice: Executor / Thread Pool

```
ExecutorService executor = Executors.newCachedThreadPool();

// Start doSomething in new (platform) thread
executor.execute(this::doSomething);
```

  - Auch für **`Callable`**

Burger-Service: 2) mit Thread Pool

- Context Propagation -> ManagedExecutor

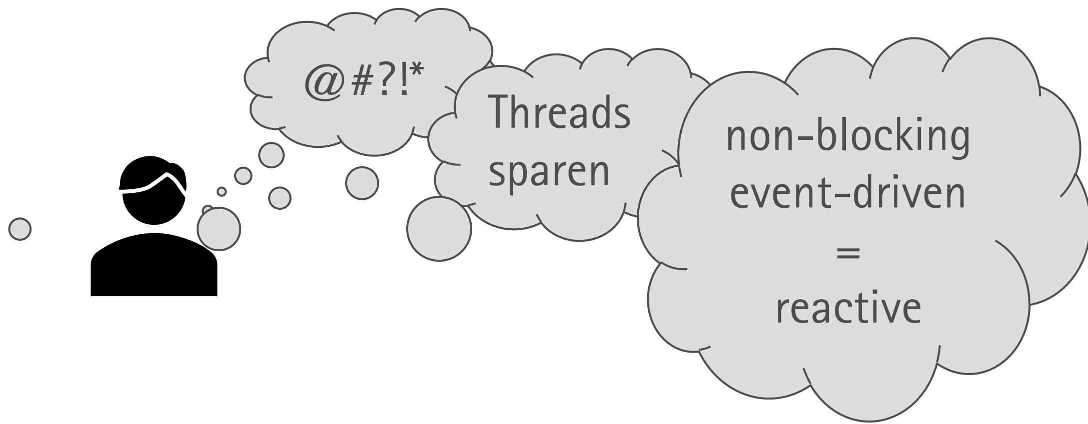
```
@Inject
ManagedExecutor executor;

@GET
@Produces(MediaType.APPLICATION_JSON)
public List<String> getBurger(…)
  throws ExecutionException, InterruptedException {

  Future<Bun> bunFuture = executor.submit(() -> bakeBun(supplyBunDough(…)));

  …

  List<String> parts = List.of(
    bunFuture.get().getUpperHalf(),
    getSauce(),
```

Demo

Threads in Java

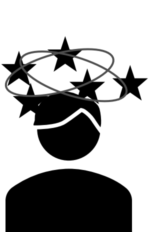
- OS Thread
- teuer
- Limitiert

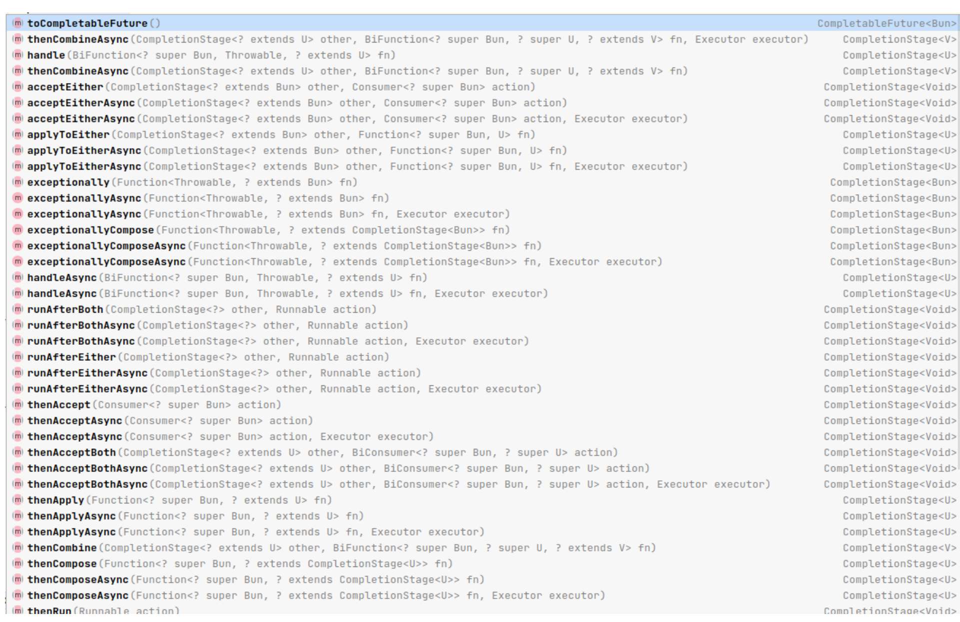
```
13:35:55,932 INFO [d.g.s.ThreadDemo] 28,000 (561 ns/thread) Thread[#28036,Thread-27999,5,main]
13:35:56,798 INFO [d.g.s.ThreadDemo] 29,000 (572 ns/thread) Thread[#29036,Thread-28999,5,main]
[17.306s][warning][os,thread] Failed to start thread "Unknown thread" - pthread_create failed (EAGAIN) for attributes: stacksize:
[17.306s][warning][os,thread] Failed to start the native thread for java.lang.Thread "Thread-29143"
java.lang.OutOfMemoryError: unable to create native thread: possibly out of memory or process/resource limits reached
  at java.base/java.lang.Thread.start0(Native Method)
  at java.base/java.lang.Thread.start(Thread.java:1535)
  at de.gedoplan.showcase.ThreadDemo.thread(ThreadDemo.java:28)
```



Completion Stage

> CompletionStage ist ein Interface in der Java Standardbibliothek, das mit der Einführung von Java 8 im Paket java.util.concurrent hinzugefügt wurde. Das Interface ist Teil des neuen asynchronen Programmiermodells, das mit Java 8 eingeführt wurde, um die Parallelverarbeitung und die asynchrone Ausführung von Aufgaben zu erleichtern.
>
> Eine CompletionStage repräsentiert eine asynchrone Berechnung, die möglicherweise noch nicht abgeschlossen ist. Sie kann als ein Versprechen (Promise) betrachtet werden, das eine zukünftige Aktion oder einen zukünftigen Wert darstellt. Eine CompletionStage kann mit anderen CompletionStages verknüpft werden, um asynchrone Verarbeitungspipelines aufzubauen.
>
> Einige der Methoden, die in der CompletionStage-Schnittstelle definiert sind, umfassen das Anhängen von Aktionen, die nach Abschluss der asynchronen Berechnung ausgeführt werden sollen, und das Kombinieren von CompletionStages, um eine neue CompletionStage zu erstellen, die auf den Abschluss mehrerer anderer CompletionStages wartet.





Completion Stage

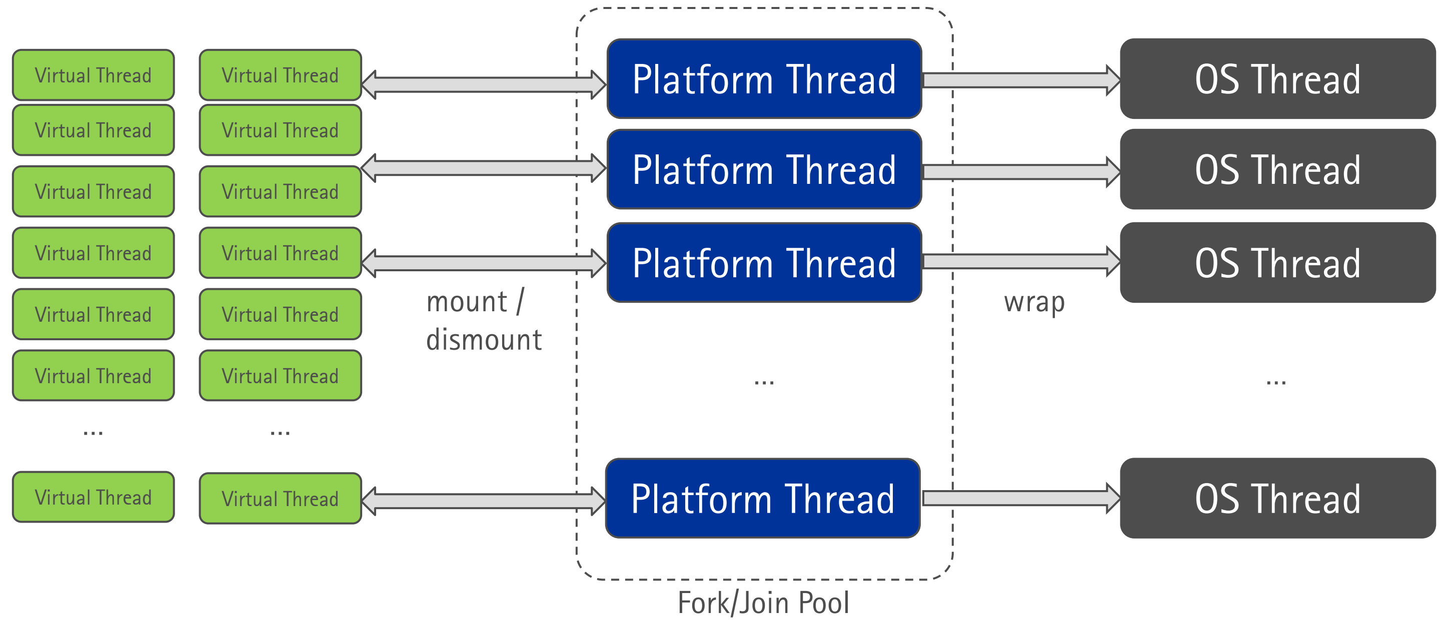
- DSL für potenziell asynchrone Abläufe
  - **`thenCompose`** ≈ führe nacheinander aus
  - **`thenCombine`** ≈ führe parallel aus
  - ...
- modelliert Kontrollstrukturen
  - ... einige, nicht alle, die in Java möglich sind

Burger-Service: 3) mit Completion Stages

Project Loom: Virtual Threads

- Terminologie

(Platform) Thread	Wrapper für OS Thread
Virtual Thread	JVM-managed Thread
Carrier Thread	Platform Thread, auf den ein virtual Thread montiert werden kann



Project Loom: Virtual Threads

```
// Start doSomething in new virtual thread
Thread.ofVirtual().start(this::doSomething);
```

```
Thread unstarted = Thread.ofVirtual().unstarted(this::doSomething);
// ...
unstarted.start();
```

```
ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor();
// ...
executor.execute(this::doSomething);
```

Burger-Service: 4) mit virtuellen Threads

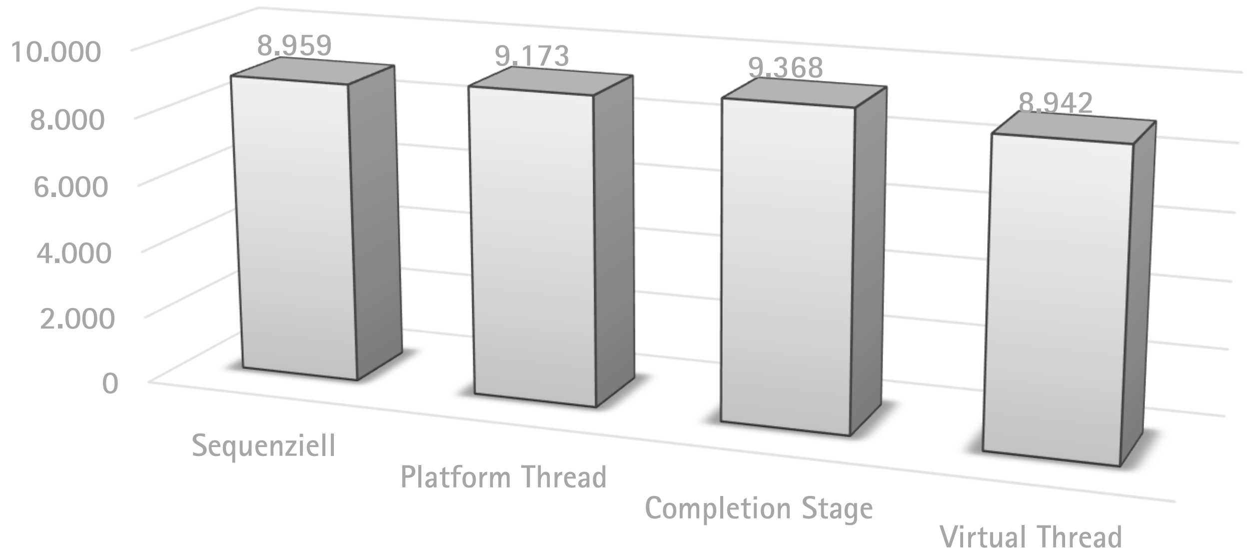
```
@GET
@Produces(MediaType.APPLICATION_JSON)
public List<String> getBurger(…)
  throws ExecutionException, InterruptedException {

  ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor();

  // Rest wie mit Standard-Threads
```

Demo

- Achtung: Derzeit kein Managed Executor

Schneller, besser, schöner?

- JMH Benchmark: µs pro Burger Call (ohne Delays)



Project Loom: Virtual Threads

- Java 21 (Java 19/20 Preview)

☺ leichtgewichtig, klein, schnell
- Millionen von Threads pro JVM

☺ kein Pooling nötig / sinnvoll

☹ ~~kein Time Slicing~~
- nicht für CPU-intensive Tasks

☹ können Platform Threads blockieren
- **`synchronized`**
- Native Calls

Unterstützung virtueller Threads in Frameworks

- Quarkus
  - HTTP-based only
    - RestEasy Reactive ➔ `@RunOnVirtualThread`
  - Pinning problems
    - Reactive Subsystems nutzen (z. B. JDBC)

- Helidon Níma
  - Ersetzt Netty durch Virtual Thread based Web Server
  - Derzeit Alpha, Release für Ende 2023 geplant

- Spring Boot
  - AsyncTaskExecutor, Tomcat können auf Virtual Threads umkonfiguriert werden

- Insgesamt: Work in Progress!

Weitere Lösungsansätze

- Structured Concurrency (Java 21 Preview, Java 19 Incubate)

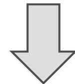
```
try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
  SubTask<Job1Result> task1 = scope.fork(this::job1);
  SubTask<Job2Resut> task2 = scope.fork(this::job2);
  scope.join();
```

- Kotlin Coroutines

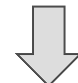
```
val job1 = GlobalScope.launch {
  // do something asynchronously
}

val job2 = GlobalScope.launch {
  // do another thing asynchronously
}

runBlocking {
  joinAll(job1, job2);
```

Königs- oder Irrweg?

- Reactive Programming erzeugt „schwierigen“ Code
- Frameworks (REST Provider, REST Client, JDBC Driver, ...)
  - hoher Nutzungsgrad (wenige Entwickelnde / viele Nutzer)
  - intern häufig ohnehin Event-basiert



  - Reactive akzeptabel
- Anwendungen
  - geringer Nutzungsgrad (individuell, viele verschiedene Anw.)



  - Reactive ungünstig

More

- Demo-Projekte in `github.com/GEDOPLAN`
  - `quarkus-demo/quarkus-virtual-threads`: Burger-Demo
  - `virtual-thread-demo`: Allgemeine Thread-Demo
- `gedoplan.de`
  Trainings in Berlin, Bielefeld, Köln, inhouse
  - Jakarta EE Intensiv
  - Microservices mit Quarkus
  - …
- `gedoplan.de`
  Reviews, Coaching, …
  Blog

- `dirk.weil@gedoplan.de`
- `@dirkweil`