

Goldschmiede Project Loom

Virtual Threads und structured Concurrency mit Java 19

31.03.2023, Hans Jörg Heßmann

⇒ Start

Vorstellung

Hans Jörg Heßmann

- Softwareentwickler
- Java seit 1997
- Schwerpunkte
 - Webanwendungen
 - Batch-Entwicklung
 - Build und Deployment
 - Codequalität und Testing
- Java, Spring, Vaadin, AsciiDoc, DessertJ, ...


Unternehmen

Individuelle Anwendungsentwicklung - Java Enterprise, Web, Mobile

- seit 2005 ♦ in Köln ♦ für alle Branchen ♦ **Goldschmiede**  **anderScore**
- nach Aufwand & zum Festpreis


Unternehmen

Individuelle Anwendungsentwicklung - Java Enterprise, Web, Mobile

- seit 2005 ♦ in Köln ♦ für alle Branchen ♦ **Goldschmiede**  anderScore
- nach Aufwand & zum Festpreis
- ✓ Digitalisierung / Prozesse / Integration
- ✓ Migration
- ✓ Neuentwicklung
- ✓ Notfall / kritische Situationen
- pragmatisch, zielgerichtet, zuverlässig

Unternehmen

Individuelle Anwendungsentwicklung - Java Enterprise, Web, Mobile

- seit 2005 ♦ in Köln ♦ für alle Branchen ♦ **Goldschmiede**  anderScore
- nach Aufwand & zum Festpreis
- ✓ Digitalisierung / Prozesse / Integration
- ✓ Migration
- ✓ Neuentwicklung
- ✓ Notfall / kritische Situationen
- pragmatisch, zielgerichtet, zuverlässig

Einführung neues
Versicherungsprodukt

µservices
Automotive

Onlinebanking
Naturschutz: Kartie-
rung bedrohter Arten

Migration
Energieversorger

Stabilisierung
Retail

Schnittstellen
Gesundheitswesen

Adressverwaltung
Logistik

Nutzerservices
Energieversorger


Börsenhandel
Wertpapiere

Security Härtung

EAM & Refactoring
Leasing

Unternehmen

Individuelle Anwendungsentwicklung - Java Enterprise, Web, Mobile

- seit 2005 ♦ in Köln ♦ für alle Branchen ♦ **Goldschmiede**  anderScore
- nach Aufwand & zum Festpreis
- ✓ Digitalisierung / Prozesse / Integration
- ✓ Migration
- ✓ Neuentwicklung
- ✓ Notfall / kritische Situationen
- pragmatisch, zielgerichtet, zuverlässig

Kompletter SW Life Cycle

- Projektmanagement / agile Methodik
- Anforderungsanalyse
- Architektur & SW-Design
- Implementierung & Testautomation
- Studien & Seminare



... und für Sie? Sprechen Sie uns an!

Überblick

- JEP 425: Virtual Threads
- JEP 428: Structured Concurrency
- JEP 429: Scoped Values

Problemstellung

Wer hat sowas schon mal erlebt?

- Performance der Anwendung ist schlecht
- Kein Engpass (CPU, Speicher, IO, Netzwerk) feststellbar
- Einfache Konfigurationsänderungen (Speicher, Anzahl Threads) bringen keine nennenswerte Verbesserung
- Kunde will schnellstmöglich Ursache und Gegenmaßnahmen erfahren

Virtual Threads

Threads in Java

Jahr	Java-Version	Features
1997	1.0	Green-Threads
1998	1.1	OS-Threads
2004	1.5	ExecutorService (z. B. ThreadPoolExecutor)
2014	1.8	CompletableFuture, Parallel Streams
2016	9	java.util.concurrent.Flow (Reactive Streams)
2022	19	Virtual Threads, structured Concurrency (Incubator)

Problem: Blockierende Aufrufe

Annahme:

1. Webserver verarbeitet Requests
2. Webservice-Aufruf dafür notwendig

⇒ Thread, der den Request verarbeitet ist für die Dauer des Webservice-Aufrufs blockiert

⇒ CPU wird nie zu 100% ausgenutzt

Lösung: Nichtblockierende Aufrufe

Umsetzung (z. B. mit `reactive Streams`):

- Webservice-Aufruf erfolgt asynchron
- Thread kann sofort den nächsten Request verarbeiten
- Beliebiger Thread reagiert auf Antwort via `Selector`

Vorteil:

- Durchsatz optimal

Nachteile:

- Stacktrace erlaubt keine Rückschlüsse auf Request
- Komplizierte Implementierung

Schön wäre, wenn für jeden Request der Stacktrace zwischengespeichert würde...

Was sind virtuelle Threads

Bisher:

1 Java-Thread — 1 Plattform-Thread

Thread-Scheduling ausschließlich durch Betriebssystem

Neu:

N Virtuelle Threads — M Plattform-Threads

Java-VM *scheduled* virtuelle Threads auf Plattform-Threads

Virtuelle Threads vs. Plattform-Threads

Plattform Threads	Virtuelle Threads
Betriebssystem-Aufrufe zum Erstellen und Beenden	Nur ein Java-Objekt für Metadaten und Stacktrace
Preemptives Multitasking (Thread kann jederzeit unterbrochen werden)	Kooperatives Multitasking (<i>work-stealing</i> ForkJoinPool)
Ein Thread ist ein Java-Thread und zugleich ein Plattform-Thread	Ein Thread ist ein Java-Thread läuft aber aus der Sicht von nativem Code auf einem völlig anderen Plattform-Thread
Priorität einstellbar und Ausführung als Dämon möglich	APIs für Priorität und Dämon sind no-ops.

Grenzen virtueller Threads

- Plattform-Thread wird blockiert durch:
 - Synchronized-Blöcke
(das soll sich in Zukunft ändern)
 - viele Systemaufrufe
- Rechenleistung bleibt unverändert

⇒ Es geht nur darum, Wartezeiten besser auszunutzen

Was ist bei virtuellen Threads zu beachten?

- Möglichst nicht blockierende API's verwenden
- synchronized vermeiden:
 - nicht blockierende Datenstrukturen
 - ReentrantLock

Ersetzen virtuelle Threads *reactive APIs*?

Nein:

- Stream-Verarbeitung ist mit reactive Streams einfacher
- Reactive Streams unterstützen Back-Pressure

Virtuellen Thread erzeugen

Plattform-Thread erzeugen:

```
Thread.ofPlatform().start(runnable);
```

ExecutorService erzeugen (Beispiel):

```
Executors.newFixedThreadPool(100);
```

Virtuellen Thread erzeugen:

```
Thread.ofVirtual().start(runnable);
```

ExecutorService erzeugen:

```
Executors.newVirtualThreadPerTaskExecutor();
```

Demo

Structured Concurrency

Einfaches Beispiel

Quelle: <https://www.thedevtavern.com/blog/posts/structured-concurrency-explained/>

```
var sum = new AtomicInteger();

// Create 3 concurrent tasks that compute the total sum in parallel
for (int i = 1; i <= 3; i++) {
    CompletableFuture.runAsync(() -> {
        var data = fetchData();
        sum.addAndGet(data);
    });
}

// Do something useful in the meantime while the sum is being computed
TimeUnit.SECONDS.sleep(2);

// And then use the sum
System.out.printf("The final sum is %d\n", sum.get());
```

Probleme:

Einfaches Beispiel

Quelle: <https://www.thedevtavern.com/blog/posts/structured-concurrency-explained/>

```
var sum = new AtomicInteger();

// Create 3 concurrent tasks that compute the total sum in parallel
for (int i = 1; i <= 3; i++) {
    CompletableFuture.runAsync(() -> {
        var data = fetchData();
        sum.addAndGet(data);
    });
}

// Do something useful in the meantime while the sum is being computed
TimeUnit.SECONDS.sleep(2);

// And then use the sum
System.out.printf("The final sum is %d\n", sum.get());
```

Probleme:

- Was passiert, wenn `fetchData()` lange braucht?

Einfaches Beispiel

Quelle: <https://www.thedevtavern.com/blog/posts/structured-concurrency-explained/>

```
var sum = new AtomicInteger();

// Create 3 concurrent tasks that compute the total sum in parallel
for (int i = 1; i <= 3; i++) {
    CompletableFuture.runAsync(() -> {
        var data = fetchData();
        sum.addAndGet(data);
    });
}

// Do something useful in the meantime while the sum is being computed
TimeUnit.SECONDS.sleep(2);

// And then use the sum
System.out.printf("The final sum is %d\n", sum.get());
```

Probleme:

- Was passiert, wenn `fetchData()` lange braucht?
- Was passiert, wenn `fetchData()` eine Exception wirft?

Einfaches Beispiel

Quelle: <https://www.thedevtavern.com/blog/posts/structured-concurrency-explained/>

```
var sum = new AtomicInteger();

// Create 3 concurrent tasks that compute the total sum in parallel
for (int i = 1; i <= 3; i++) {
    CompletableFuture.runAsync(() -> {
        var data = fetchData();
        sum.addAndGet(data);
    });
}

// Do something useful in the meantime while the sum is being computed
TimeUnit.SECONDS.sleep(2);

// And then use the sum
System.out.printf("The final sum is %d\n", sum.get());
```

Probleme:

- Was passiert, wenn `fetchData()` lange braucht?
- Was passiert, wenn `fetchData()` eine Exception wirft?
- Was passiert, wenn `fetchData()` in eine Endlosschleife läuft?

Verbessertes Beispiel

```
var sum = new AtomicInteger();

// A list to keep track of the created futures
var futures = new ArrayList<Future<Void>>();

// Create 3 concurrent tasks that compute the total sum in parallel
for (int i = 1; i <= 3; i++) {
    futures.add(CompletableFuture.runAsync(() -> {
        var data = fetchData();
        sum.addAndGet(data);
    }));
}

// Do something useful in the meantime while the sum is being computed
TimeUnit.SECONDS.sleep(2);

// Wait for all the futures and propagate exceptions
for (Future future : futures) {
    future.get();
}

// And then use the sum
System.out.printf("The final sum is %d\n", sum.get());
```

Probleme:

Verbessertes Beispiel

```
var sum = new AtomicInteger();

// A list to keep track of the created futures
var futures = new ArrayList<Future<Void>>();

// Create 3 concurrent tasks that compute the total sum in parallel
for (int i = 1; i <= 3; i++) {
    futures.add(CompletableFuture.runAsync(() -> {
        var data = fetchData();
        sum.addAndGet(data);
    }));
}

// Do something useful in the meantime while the sum is being computed
TimeUnit.SECONDS.sleep(2);

// Wait for all the futures and propagate exceptions
for (Future future : futures) {
    future.get();
}

// And then use the sum
System.out.printf("The final sum is %d%n", sum.get());
```

Probleme:

- Threads laufen weiter, wenn 1. fetchData() Exception wirft

Verbessertes Beispiel

```
var sum = new AtomicInteger();

// A list to keep track of the created futures
var futures = new ArrayList<Future<Void>>();

// Create 3 concurrent tasks that compute the total sum in parallel
for (int i = 1; i <= 3; i++) {
    futures.add(CompletableFuture.runAsync(() -> {
        var data = fetchData();
        sum.addAndGet(data);
    }));
}

// Do something useful in the meantime while the sum is being computed
TimeUnit.SECONDS.sleep(2);

// Wait for all the futures and propagate exceptions
for (Future future : futures) {
    future.get();
}

// And then use the sum
System.out.printf("The final sum is %d%n", sum.get());
```

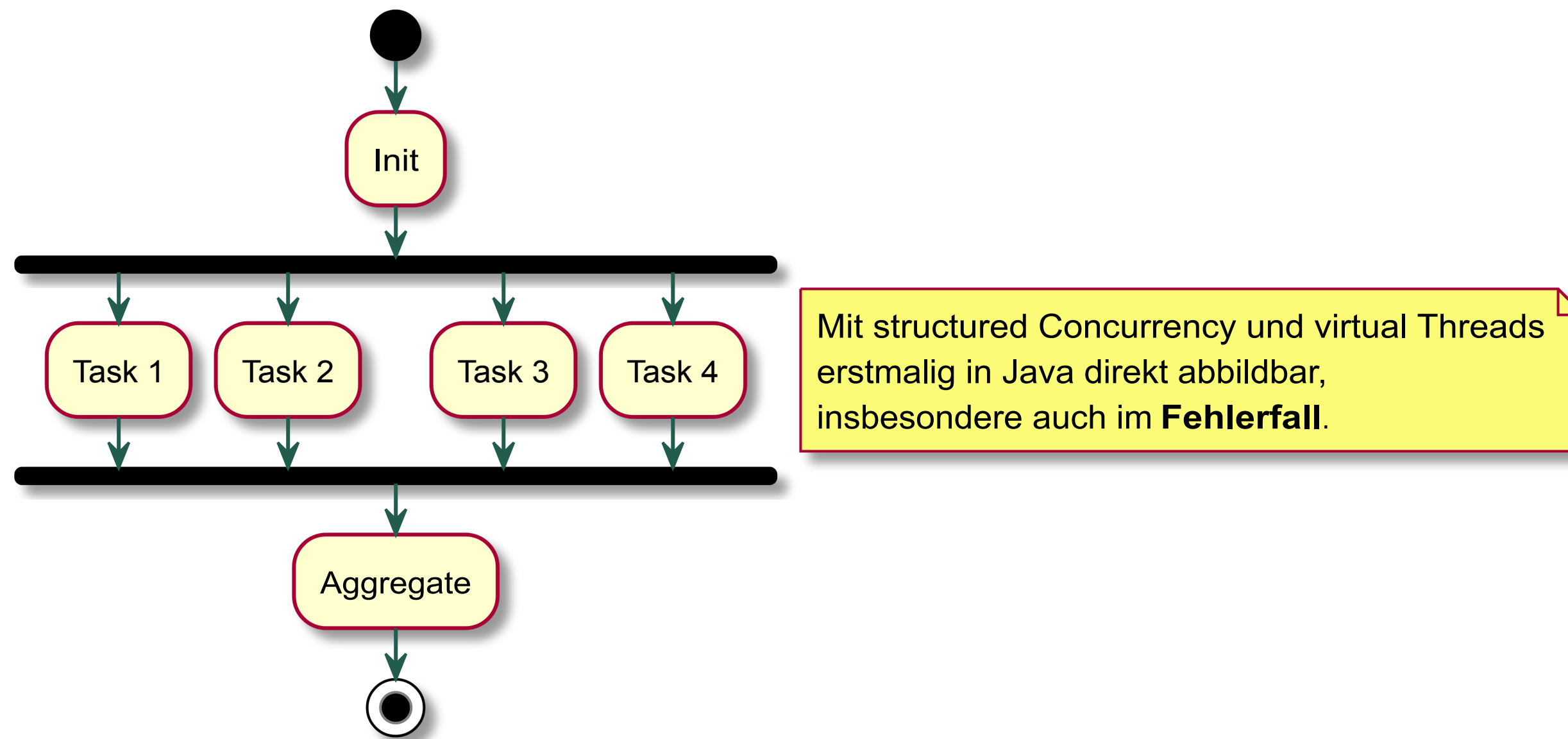
Probleme:

- Threads laufen weiter, wenn 1. fetchData() Exception wirft
- Berechnung lässt sich nicht abbrechen

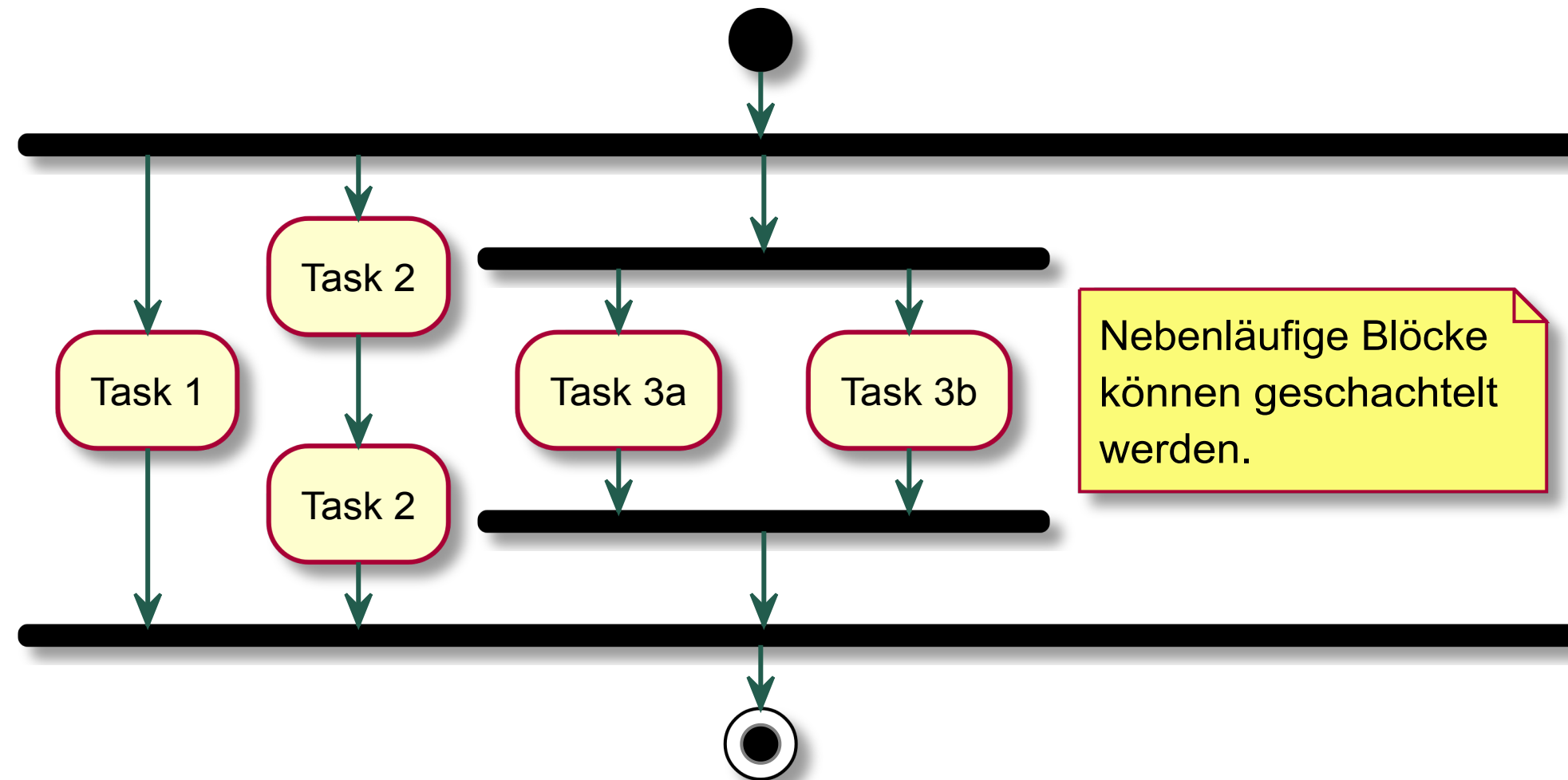
Beispiel mit structured Concurrency

```
try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {  
    var sum = new AtomicInteger();  
  
    // Create 3 concurrent tasks that compute the total sum in parallel  
    for (int i = 1; i <= 3; i++) {  
        scope.fork(Executors.callable(() -> {  
            var data = fetchData();  
            sum.addAndGet(data);  
        }));  
    }  
  
    // Do something useful in the meantime while the sum is being computed  
    TimeUnit.SECONDS.sleep(2);  
  
    // Wait for all the futures and propagate exceptions  
    scope.join();  
    scope.throwIfFailed();  
  
    // And then use the sum  
    System.out.printf("The final sum is %d%n", sum.get());  
}
```


Structured Concurrency als Aktivitiendiagramm



Schachtelung



Eigenschaften von Structured Concurrency

- Logische Klammer um parallel ausgeführte Aufgaben (ähnlich einer Methode)
- Unbehandelte Exception beendet die Ausführung aller Threads:
 - wenn eine Teilaufgabe fehlschlägt
 - wenn es zu Fehler in logischer Klammer kommt
- Logische Klammer kann als ganzes abgebrochen werden (shutdown)

Structured Concurrency API

StructuredTaskScope.ShutdownOnFailure

- Bricht den ganzen Block ab, sobald ein Subtask eine Exception wirft
- Nützlich, wenn das Endergebnis jedes Teilergebnis erfordert (Normalfall)

StructuredTaskScope.ShutdownOnSuccess<T>

- Bricht den ganzen Block ab, sobald ein Subtask ein Ergebnis liefert
- Nützlich, wenn jeder Subtask das gleiche Ergebnis ermittelt (der Schnellste gewinnt)

Durch Schachtelung kann man beide Kombinieren

Verwenden von Structured Concurrency

Structured Concurrency ist noch im *Incubator* Status:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>3.10.1</version>
  <configuration>
    <compilerArgs>
      <arg>--enable-preview</arg>
      <arg>--add-modules</arg>
      <arg>jdk.incubator.concurrent</arg>
    </compilerArgs>
    <source>${java.version}</source>
    <target>${java.version}</target>
  </configuration>
</plugin>
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>3.0.0-M9</version>
  <configuration>
    <argLine>--enable-preview --add-modules jdk.incubator.concurrent</argLine>
    <trimStackTrace>>false</trimStackTrace>
  </configuration>
</plugin>
```

Scoped Values

Teaser

*The enterprise bean must not attempt to manage threads. **The enterprise bean must not attempt to start, stop, suspend, or resume a thread**, or to change a thread's priority or name. The enterprise bean must not attempt to manage thread groups.*

Quelle: Jakarta® Enterprise Beans, Core Features, Version 4.0 Final

Warum?

Teaser

*The enterprise bean must not attempt to manage threads. **The enterprise bean must not attempt to start, stop, suspend, or resume a thread**, or to change a thread's priority or name. The enterprise bean must not attempt to manage thread groups.*

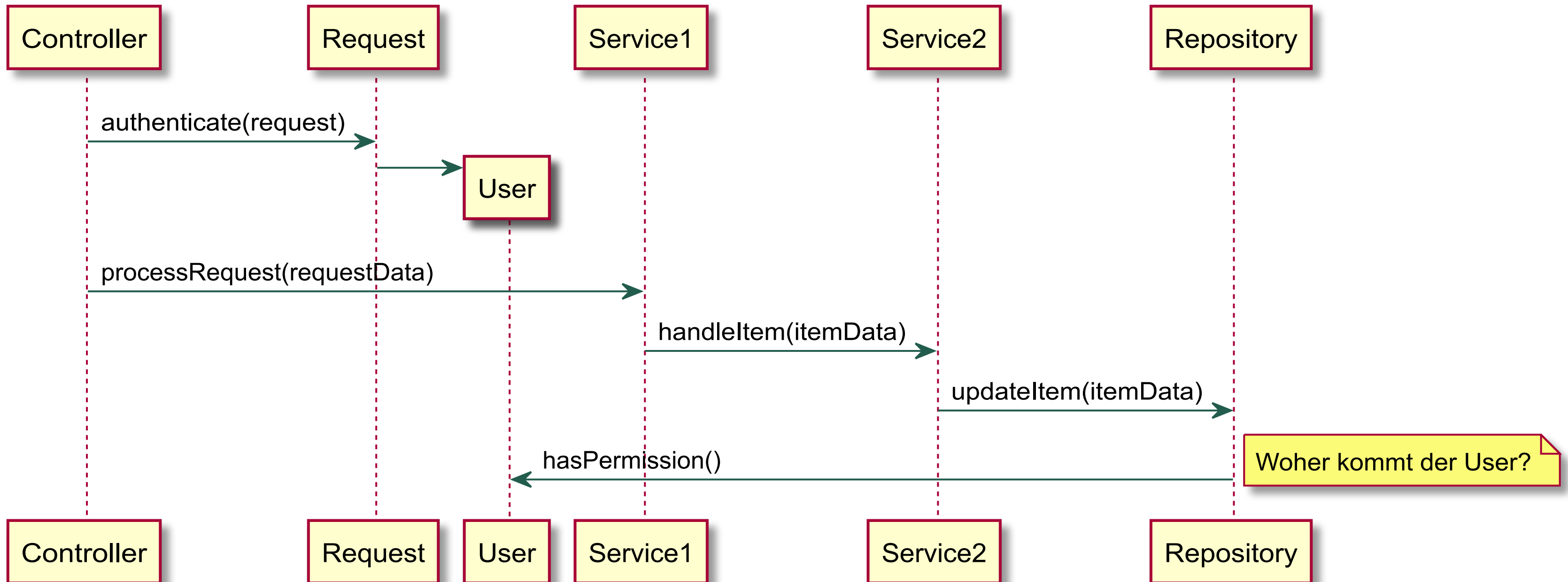
Quelle: Jakarta® Enterprise Beans, Core Features, Version 4.0 Final

Warum?

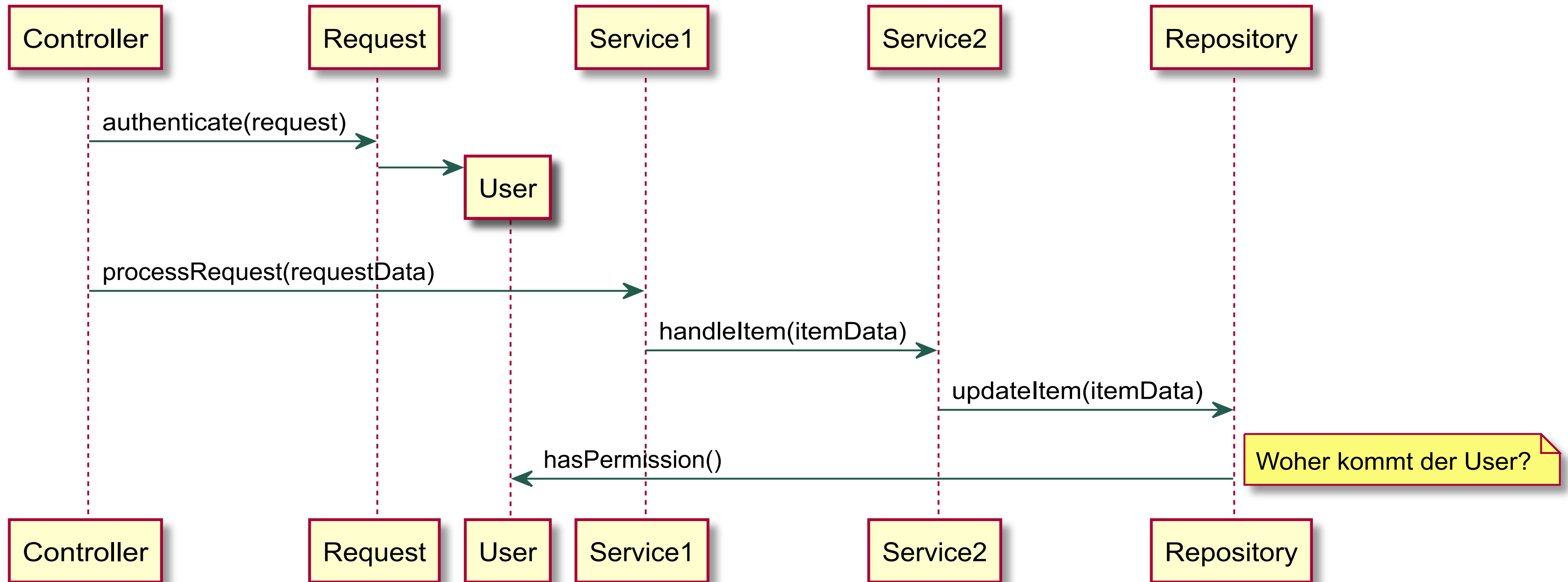
- Antwort (ebendort):

These functions are reserved for the Enterprise Beans container. Allowing the enterprise bean to manage threads would decrease the container's ability to properly manage the runtime environment.

Problemstellung



Problemstellung



- Lösung: ThreadLocal

Thread lokal Variables

- Lebenszeit an Thread gebunden
(Vorsicht: Memory-Leak)
- Ein Exemplar je Thread
wg. set-Methode
(teuer)
- Für Kontextinformationen
(z. B. aktueller User)

Was gibt der Test aus?

```
private ThreadLocal<Integer> threadLocalValue =  
    new InheritableThreadLocal<>();  
  
@Test  
public void testThreadLocal() {  
    assertThat(threadLocalValue.get()).isNull();  
    threadLocalValue.set(0);  
}
```

```
for (int i = 1; i <= 5; i++) {
    Thread.ofVirtual()
        .name("my-thread-", i)
        .allowSetThreadLocals(true) // default
        .inheritInheritableThreadLocals(true) // default
        .start(this::dumpThreadLocal);
}
dumpThreadLocal();
dumpThreadLocal();
threadLocalValue.remove();
dumpThreadLocal();
}

private void dumpThreadLocal() {
    System.out.printf("[%s] threadLocalValue = %d%n",
        Thread.currentThread().getName(),
        threadLocalValue.get());
    threadLocalValue.set(99);
}
```

Scoped Values

- Gleicher Verwendungszweck wie `ThreadLocal`
- Lebenszeit ist an Ausführungskontext gebunden
- Wird **ausschließlich** innerhalb von `StructuredTaskScope` vererbt
- Unveränderlich (kein set)
- Sollte immutable sein (z. B. Record)
- Rebinding für geschachtelten Ausführungskontext möglich (z. B. für `sudo`)

Verwendung von Scoped Values

Scoped Value deklarieren

```
private static final ScopedValue<String> CONTEXT = ScopedValue.newInstance();  
private static final ScopedValue<String> USERNAME = ScopedValue.newInstance();
```

Scoped Value setzen

```
ScopedValue.where(USERNAME, "duke")  
    .where(CONTEXT, "test")  
    .run(this::doNested);
```

Scoped Value auslesen (ähnlich *Optional*)

```
CONTEXT.orElse("undefined"),  
USERNAME.orElse("undefined"),
```

Einschränkungen

Datenbankzugriff (JDBC)

- DB-Transaktion ist an Connection gebunden
- Jeder Thread benötigt eigene Connection
 - \Rightarrow Verteilte Transaktionen
 - \Rightarrow Eventual (schließlich) Consistency
- Anzahl Threads ist durch Connection-Pool begrenzt
- Alternative: **R2DBC**
 - \rightarrow <https://spring.io/blog/2019/05/16/reactive-transactions-with-spring>

Fragen?

Sourcecode: <https://github.com/goldschmiede/loom>

