Virtual threads

Quarkus

JoeDayz 2025


QUARKUS

Hilos Virtuales en Java 21

Introducción

La concurrencia es clave en el software moderno, especialmente para aplicaciones que manejan muchas tareas simultáneas. Java 21 introduce hilos virtuales, una alternativa escalable y eficiente a los hilos tradicionales.

Hilos Virtuales en Java 21

Problemas con los hilos tradicionales

- Alto consumo de memoria: Cada hilo del sistema operativo consume ~1MB.
- Bloqueos en I/O: Un hilo bloqueado ocupa recursos del OS.
- Escalabilidad limitada: Manejar miles de hilos es costoso por el cambio de contexto y la planificación.

Hilos Virtuales en Java 21

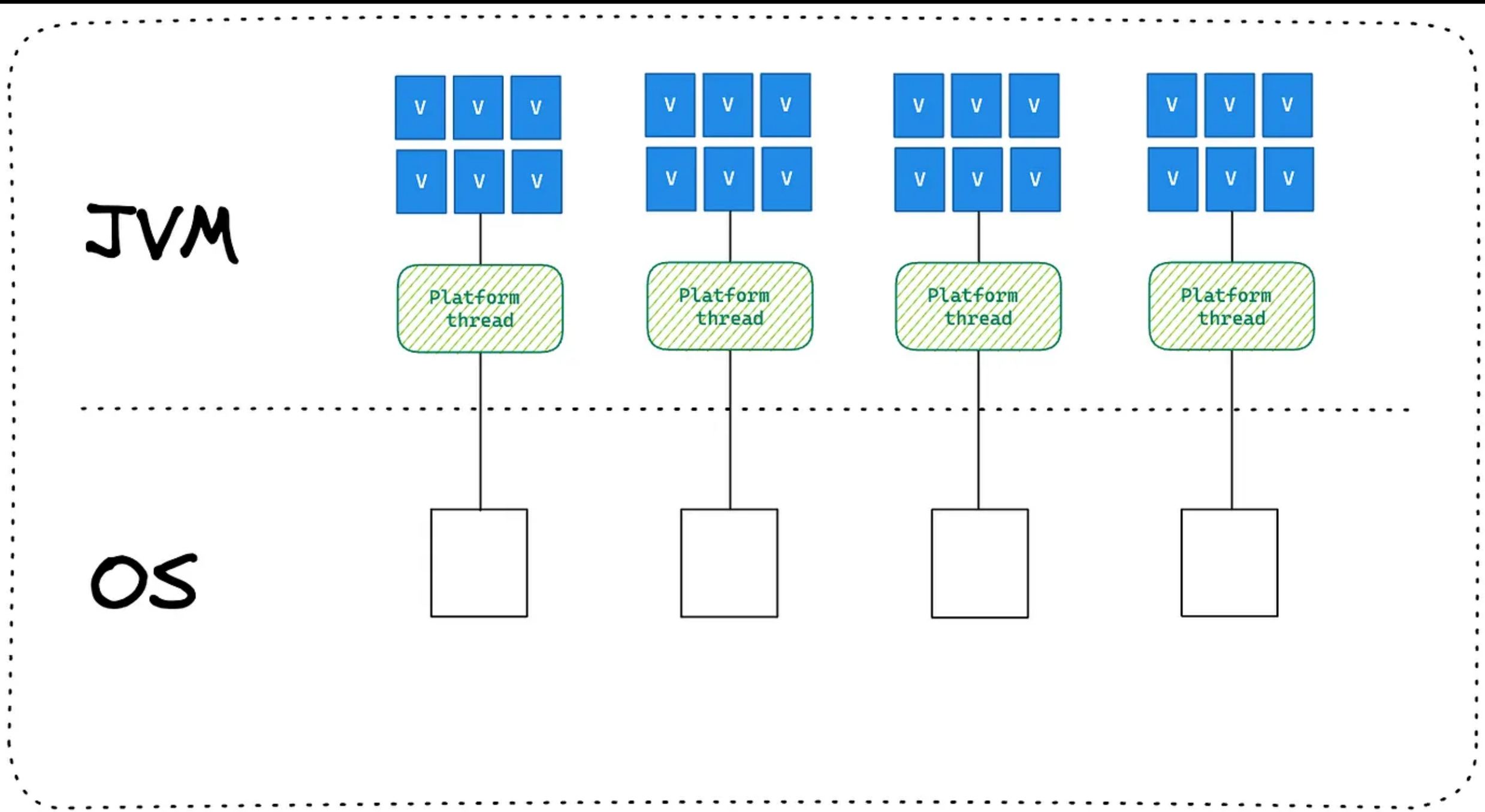
Ventajas de los Hilos Virtuales

- Bajo consumo de memoria: Permite millones de hilos concurrentes.
- Eficiencia en I/O: Bloquean sin usar hilos del OS.
- Gestión simple de concurrencia: Código sincrónico que se ejecuta de forma altamente escalable.

Hilos Virtuales en Java 21

¿Cómo Funcionan?

- Los hilos virtuales son gestionados por la JVM. Cuando uno bloquea en I/O, el hilo del OS subyacente se libera y otro hilo virtual puede ejecutarse, permitiendo una gran eficiencia y escalabilidad.


JVM
OS
Platform thread
Platform thread
Platform thread
Platform thread

Hilos Virtuales en Java 21

Hilos Tradicionales vs Hilos Virtuales

```java
for (int i = 0; i < 1000; i++) {
    new Thread(() -> {
        // Operación bloqueante
    }).start();
}
```

Vs

```java
for (int i = 0; i < 1000; i++) {
    Thread.startVirtualThread(() -> {
        // Operación bloqueante
    });
}
```

Puntos clave:

- Los hilos tradicionales consumen mucha memoria y no escalan bien.
- Los hilos virtuales son ligeros, eficientes y permiten millones de hilos concurrentes.

Hilos Virtuales en Java 21

Casos de Uso

- **Microservicios y servidores web:**

  Manejan miles de conexiones concurrentes de manera eficiente, sin el overhead de hilos tradicionales.

- **Aplicaciones I/O-bound:**

  Ideal para operaciones que esperan datos de base de datos, archivos o red, permitiendo código bloqueante sin penalizar recursos del sistema.

- **Procesamiento por lotes:**

  Permite asignar un hilo virtual por tarea, logrando código más limpio y concurrente sin sacrificar rendimiento.

Hilos Virtuales en Java 21

Beneficios de Rendimiento

- **Menor consumo de memoria:**

  Hilos más ligeros que permiten escalar mejor que los hilos tradicionales.

- **Mejor rendimiento en I/O:**

  No bloquean hilos del OS, aumentando throughput y reduciendo latencia en aplicaciones como servidores web.

- **Mayor escalabilidad:**

  Permiten manejar millones de tareas concurrentes sin saturar los recursos del sistema.

Hilos Virtuales en Java 21

Retos y consideraciones

- Versión de JVM: Solo disponibles a partir de Java 21.
- Monitoreo y depuración: Las herramientas tradicionales pueden no manejar correctamente la gran cantidad de hilos virtuales.
- Revisar estrategias de concurrencia: Migrar desde thread pools o librerías reactivas puede requerir refactorización.

Hilos Virtuales en Java 21

Quarkus y Virtual Threads

- Conceptos Clave:

Virtual Thread: Hilo ligero gestionado por la JVM, no ligado a un hilo del OS.

Carrier Thread: Hilo de plataforma que ejecuta un hilo virtual.

Pinning: Situación donde un hilo virtual bloquea su carrier thread (ej. operaciones sincronizadas o nativas).

Monopolización: Hilos virtuales CPU-bound pueden crear demasiados carrier threads, aumentando consumo de memoria.

- Uso en Quarkus:

@RunOnVirtualThread: Ejecuta endpoints REST en hilos virtuales, permitiendo operaciones bloqueantes sin bloquear hilos de plataforma.

Clientes amigables con hilos virtuales: Extensiones Quarkus (REST Client, Redis, mailer) y drivers reactivas (Mutiny) evitan pinning.

Contexto duplicado: Métodos heredan contexto duplicado, pero ThreadLocals no se propagan.

Pruebas: junit5-virtual-threads detecta pinning en tests para mejorar confiabilidad.

Métricas: Micrometer Java 21 binder monitorea pinning y hilos virtuales fallidos.

Hilos Virtuales en Java 21

Quarkus y Virtual Threads

- Buenas prácticas:

Usar hilos virtuales solo para cargas I/O-bound.

Evitar código CPU-bound en hilos virtuales.

Revisar librerías que usan ThreadLocal o pooling.

Configurar nombres de hilos virtuales para depuración (quarkus-virtual-thread-).

Modelos Avanzados de Concurrencia en Quarkus

Modelos de Concurrencia

Introducción

1. **Hilo por Conexión (Thread per connection)**
   - Cada solicitud se ejecuta en un hilo de plataforma dedicado.
   - Código simple pero limitada escalabilidad
2. **Event Loop (Bucle de eventos)**
   - Uso de hilos mínimos para manejar muchas conexiones.
   - Alta concurrencia y bajo consumo de recursos.
   - Código más complejo (reactivo)
3. **Hilos Virtuales (Virtual threads)**
   - Cada solicitud puede ejecutarse en un hilo virtual ligero
   - Código sincrónico simple, escaladle y eficiente para I/O.
   - Riesgo de pinning y monopolización si se usa incorrectamente.

Hilo por conexión

Concepto:

- Cada solicitud del cliente recibe un hilo dedicado.
- Modelo clásico en servidores Java, soportado en Quarkus con el stack tradicional basado en servlets.

Cómo Funciona:

- El contenedor (p. ej., Undertow) asigna un hilo para cada nueva solicitud.
- El procesamiento es síncrono dentro del hilo asignado.

Pros:

- Programación intuitiva y sincrónica.
- Ecosistema maduro: muchas librerías Java asumen comportamiento bloqueante.

Contras:

- Riesgo de agotamiento del pool de hilos bajo alta carga.
- Mayor consumo de memoria (cada hilo necesita su stack).
- Costos de cambio de contexto pueden degradar rendimiento.

```java
@Path("/blocking")
@Produces(MediaType.TEXT_PLAIN)
public class BlockingResource {
    @GET
    public String blockingMethod() {
        try {
            // Simulate a blocking operation
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
        return "Blocking response";
    }
}
```

```
# Worker pool size
quarkus.vertx.worker-pool-size=200
# Thread pool settings
quarkus.thread-pool.core-threads=100
quarkus.thread-pool.max-threads=500
```

Bucle de Eventos

Concepto:

- Un pequeño número de hilos del bucle de eventos (generalmente uno por núcleo de CPU) maneja todas las operaciones de I/O de manera asíncrona.
- Quarkus usa Vert.x y Mutiny para habilitar este enfoque reactivo.

Cómo Funciona:

- I/O no bloqueante evita que los hilos del bucle se detengan.
- Las tareas bloqueantes deben derivarse a hilos worker.

Pros:

- Alta concurrencia y escalabilidad.
- Uso eficiente de CPU y memoria.

Contras:

- Patrón de callbacks o operadores reactivos más complejo.
- Operaciones bloqueantes mal manejadas pueden afectar el throughput.

Ruta Reactiva

```java
@Route(path = "/reactive", methods = HttpMethod.GET)
public void reactiveRoute(RoutingContext rc) {
    Uni.createFrom().item(() -> expensiveCalculation())
        // Transform the result
        .onItem().transform(result -> "Reactive result: " + result)
        // Send back to client
        .subscribe().with(
            response -> rc.response().end(response),
            failure -> rc.response().setStatusCode(500).end("Error: " + failure.g
        );
}
private String expensiveCalculation() {
    // Simulate a CPU/IO heavy task
    try {
        Thread.sleep(500);
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    }
    return "42";
}
```

Offloading Blocking Work

```java
vertx.executeBlocking(promise -> {
    String result = blockingService.performLongTask();
    promise.complete(result);
}, asyncResult -> {
    if (asyncResult.succeeded()) {
        rc.response().end("Result: " + asyncResult.result());
    } else {
        rc.response().setStatusCode(500).end("Failure");
    }
});
```

Virtual Threads

Concepto:

- Los hilos virtuales (introducidos con Project Loom en versiones recientes de Java) permiten escribir código síncrono mientras soportan alta concurrencia.
- Cada hilo virtual es mucho más ligero que un hilo del sistema operativo, reduciendo drásticamente el overhead de cambio de contexto y el uso de memoria.

Cómo funciona:

- Los hilos virtuales son programados por la JVM en lugar del SO, por lo que miles de hilos virtuales pueden coexistir sin agotar los recursos del sistema.
- El código permanece síncrono (puedes usar Thread.sleep(), I/O bloqueante, etc.) con un impacto mínimo en el rendimiento.

Ventajas:

- Mantiene un estilo síncrono familiar sin necesidad de enormes pools de hilos.
- Simplifica la depuración y la estructura del código (sin callbacks reactivos complejos).

Desventajas:

- Tecnología aún nueva, puede haber riesgos de compatibilidad con algunas librerías.
- Requiere Java 19+ (o Java 21 LTS) con características de Loom habilitadas.

Virtual Thread Executor

```java
// Using Java Loom preview features
ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor();
public void runVirtualThreadExample() {
    // Submit tasks to virtual threads
    for (int i = 0; i < 10000; i++) {
        executor.submit(() -> {
            // Simulate blocking
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
            return "Result from virtual thread";
        });
    }
    executor.shutdown();
}
```

Hoy!!

```java
package org.acme.rest;

import org.acme.fortune.model.Fortune;
import org.acme.fortune.repository.FortuneRepository;
import io.smallrye.common.annotation.RunOnVirtualThread;
import io.smallrye.mutiny.Uni;

import jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;
import java.util.List;
import java.util.Random;


@Path("")
public class FortuneResource {

    @Inject FortuneRepository repository;

    @GET
    @Path("/blocking")
    public Fortune blocking() {
        // Runs on a worker (platform) thread
        var list = repository.findAllBlocking();
        return pickOne(list);
    }

    @GET
    @Path("/reactive")
    public Uni<Fortune> reactive() {
        // Runs on the event loop
        return repository.findAllAsync()
                .map(this::pickOne);
    }

    @GET
    @Path("/virtual")
    @RunOnVirtualThread
    public Fortune virtualThread() {
        // Runs on a virtual thread
        var list = repository.findAllAsyncAndAwait();
        return pickOne(list);
    }

}
```

Model	Example of signature	Pros	Cons
Synchronous code on worker thread	`Fortune blocking()`	Simple code	Use worker thread (limit concurrency)
Reactive code on event loop	`Uni<Fortune> reactive()`	High concurrency and low resource usage	More complex code
Synchronous code on virtual thread	`@RunOnVirtualThread Fortune vt()`	Simple code	Risk of pinning, monopolization and under-efficient object pooling

Usando clientes compatibles con VT

Contexto:
El ecosistema Java aún no está completamente listo para hilos virtuales, por lo que se debe tener cuidado, especialmente con librerías que realizan I/O.

Quarkus ofrece un ecosistema preparado para hilos virtuales:

- Mutiny y los bindings Vert.x Mutiny permiten escribir código bloqueante sin bloquear el hilo carrier.
- Extensiones Quarkus con APIs bloqueantes sobre APIs reactivas pueden usarse en hilos virtuales: REST Client, Redis client, mailer, etc.

Prácticas recomendadas:

- APIs que retornan Uni pueden usarse con uni.await().atMost(...), bloqueando solo el hilo virtual y mejorando la resiliencia con timeouts no bloqueantes.
- Clientes Vert.x con bindings Mutiny: usar andAwait() para bloquear hasta obtener resultados sin pinnear el hilo carrier, incluyendo drivers SQL reactivos.

Detectar hilos “pinned” en tests

Contexto:
Al ejecutar tests en aplicaciones que usan **hilos virtuales**, algunos hilos pueden **quedarse “pinned”** en el hilo carrier, afectando el rendimiento y consumo de memoria.

Recomendación:
Configurar Maven Surefire para **registrar trazas de hilos pinned** durante los tests. Esto **no falla los tests**, pero permite identificar problemas.

```
<plugin>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>${surefire-plugin.version}</version>
  <configuration>
      <systemPropertyVariables>
        <java.util.logging.manager>org.jboss.logmanager.LogManager</java.util.logging.manager>
        <maven.home>${maven.home}</maven.home>
      </systemPropertyVariables>
      <argLine>-Djdk.tracePinnedThreads</argLine>
  </configuration>
</plugin>
```

Ejecutar aplicaciones usando virtual thread

```
java -jar target/quarkus-app/quarkus-run.jar
```

Construir contenedores para aplicaciones con VT

Contexto:
Al ejecutar tu aplicación en **modo JVM** (no nativo), puedes **containerizarla** siguiendo la guía de Quarkus. Aquí usamos **JIB** para crear la imagen.

Configuración en application.properties:

```
quarkus.container-image.build=true
quarkus.container-image.group=<tu-grupo>
quarkus.container-image.name=<nombre-de-tu-contenedor>
quarkus.jib.base-jvm-image=registry.access.redhat.com/ubi9/openjdk-21-runtime
quarkus.jib.platforms=linux/amd64,linux/arm64
```

NOTAS IMPORTANTES:
- Usa una imagen base compatible con hilos virtuales (Java 21+)
- Selecciona la arquitectura objetivo; se pueden construir imágenes multi-arch.

```
mvn package
```

Construye tu contenedor

Nombres de virtual threads

- Por defecto, los hilos virtuales se crean sin nombre, lo que dificulta el debugging y Logging.
- Quarkus gestiona los hilos virtuales y les asigna un prefijo.

```
quarkus-virtual-thread-
```

- Es posible personalizar el prefijo o desactivar la asignación de nombres configurando un valor vacío:

```
quarkus.virtual-threads.name-prefix=
```

- Esto ayuda a identificar fácilmente los hilos virtuales en logs y herramientas de monitoreo.

Inyectar el virtual thread executor

- Quarkus gestiona internamente un ThreadPerTaskExecutor para ejecutar tareas en hilos virtuales.
- En casos rara vez necesarios, se puede inyectar directamente usando el qualifier @VirtualThreads de CDI.
- Nota: La inyección del ExecutorService de hilos virtuales es experimental y podría cambiar en futuras versiones.

```java
package org.acme;

import org.acme.fortune.repository.FortuneRepository;

import java.util.concurrent.ExecutorService;

import jakarta.enterprise.event.Observes;
import jakarta.inject.Inject;
import jakarta.transaction.Transactional;

import io.quarkus.logging.Log;
import io.quarkus.runtime.StartupEvent;
import io.quarkus.virtual.threads.VirtualThreads;

public class MyApplication {

    @Inject
    FortuneRepository repository;

    @Inject
    @VirtualThreads
    ExecutorService vThreads;

    void onEvent(@Observes StartupEvent event) {
        vThreads.execute(this::findAll);
    }

    @Transactional
    void findAll() {
        Log.info(repository.findAllBlocking());
    }

}
```

Testeando aplicaciones con virtual threads

Detección de hilos “pinned” en tests con Quarkus

- Los hilos virtuales tienen limitaciones que pueden afectar el rendimiento y uso de memoria.
- La extensión junit5-virtual-threads permite detectar hilos carrier “pinned” durante los tests.
- Configuración:

```xml
<dependency>
    <groupId>io.quarkus.junit5</groupId>
    <artifactId>junit5-virtual-threads</artifactId>
    <scope>test</scope>
</dependency>
```

- Anotar las clases de test con:

```java
@QuarkusTest
@TestMethodOrder(MethodOrderer.OrderAnnotation.class)
@VirtualThreadUnit    // Usar la extensión
@ShouldNotPin         // Detectar hilos pinned
class TodoResourceTest { ... }
```

- Si se detecta un hilo pinned el test falla.
- Se pueden usar anotaciones en métodos directamente:
  - @ShouldNotPin: el hilo no debe quedarse pinned.
  - @ShouldPin (atMost=1): permite un pinning máximo aceptable.

- Esto ayuda a prevenir problemas de rendimiento y memoria relacionados con hilos virtuales.

Metricas de Virtual threads

- Se puede habilitar el binder de hilos virtuales agregando la dependencia:

```xml
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-java21</artifactId>
</dependency>
```

- Este binder monitorea:
  - Número de eventos de pinning.
  - Hilos virtuales que no se pudieron iniciar o reactivar.

- Se puede desactivar explícitamente en application.properties.

```
quarkus.micrometer.binder.virtual-threads.enabled=false
```

Metricas de Virtual threads

- Si se ejecuta en una JVM anterior a Java 21, el binder se desactiva automáticamente.
- Se pueden asociar tags a las métricas recolectadas:

```
quarkus.micrometer.binder.virtual-threads.tags=tag_1=value_1, tag_2=value_2
```

Soporte de Hilos Virtuales en Quarkus con Reactive Messaging

Hilos Virtuales con Reactive Messaging

Objetivo: Procesar mensajes usando hilos virtuales en lugar de hilos de event-loop o worker threads.

Situación típica: Por defecto, Reactive Messaging ejecuta métodos de procesamiento en hilos de event-loop.

Problema: Algunas operaciones, como llamadas a servicios externos o bases de datos, son bloqueantes.

Solución:

- Usar @Blocking indica que el procesamiento es bloqueante y debe ejecutarse en un worker thread.
- Con hilos virtuales, se puede usar @RunOnVirtualThread para offload del procesamiento a un hilo virtual.

Requisitos: Java 19+, se recomienda Java 21.

Beneficio: Permite operaciones bloqueantes sin bloquear el hilo de plataforma donde se monta el hilo virtual.

Ejemplo usando Reactive Messaging Kafka

```xml
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-messaging-kafka</artifactId>
</dependency>
```

```xml
<properties>
    <maven.compiler.source>21</maven.compiler.source>
    <maven.compiler.target>21</maven.compiler.target>
</properties>
```

Ejemplo usando Reactive Messaging Kafka

```java
import org.eclipse.microprofile.reactive.messaging.Incoming;
import org.eclipse.microprofile.reactive.messaging.Outgoing;
import org.eclipse.microprofile.rest.client.inject.RestClient;

import io.smallrye.common.annotation.RunOnVirtualThread;

import jakarta.enterprise.context.ApplicationScoped;

@ApplicationScoped
public class PriceConsumer {

    @RestClient 2
    PriceAlertService alertService;


    @Incoming("prices")
    @RunOnVirtualThread 1
    public void consume(double price) {
        if (price > 90.0) {
            alertService.alert(price); 3
        }
    }

    @Outgoing("prices-out") 4
    public Multi<Double> randomPriceGenerator() {
        return Multi.createFrom().<Random, Double>generator(Random::new, (r, e) -> {
            e.emit(r.nextDouble(100));
            return r;
        });
    }


}
```

- @RunOnVirtualThread en el método @Incoming asegura que el método será llamado en un virtual thread.
- El REST client stub es inyectado con la anotación @RestClient.
- El método alert bloquea el virtual thread hasta que la llamada REST retorne.
- El método @Outgoing genera precios aleatorios y escribe ellos en un tópico Kafka para ser consumido luego por la aplicación.

- Procesamiento de mensajes concurrentes con hilos virtuales
  - Por defecto, Reactive Messaging procesa los mensajes de forma secuencial, preservando el orden.
  - La anotación @Blocking(ordered=false) permite procesamiento no secuencial.
  - Usar @RunOnVirtualThread permite procesar mensajes concurrentemente, sin preservar el orden.

Usando la anotación @RunOnVirtualThread

- `@Outgoing("channel-out") O generator()`
- `@Outgoing("channel-out") Message<O> generator()`
- `@Incoming("channel-in") @Outgoing("channel-out") O process(I in)`
- `@Incoming("channel-in") @Outgoing("channel-out") Message<O> process(I in)`
- `@Incoming("channel-in") void consume(I in)`
- `@Incoming("channel-in") Uni<Void> consume(I in)`
- `@Incoming("channel-in") Uni<Void> consume(Message<I> msg)`
- `@Incoming("channel-in") CompletionStage<Void> consume(I in)`
- `@Incoming("channel-in") CompletionStage<Void> consume(Message<I> msg)`

- Estas son firmas elegibles para `@RunOnVirtualThread`
- Sólo los métodos que pueden ser anotados con `@Blocking` pueden usar `@RunOnVirtualThread`

Usar @RunOnVirtualThread en métodos y clases

- Tu puedes usar @RunOnVirtualThread:
  - Directamente en un método reactive messaging – este método será considerado bloqueante y ejecutado en un virtual thread.
  - En la clase que contiene método reactive messaging – los métodos de esta clase anotada con @Blocking serán ejecutados en virtual thread, excepto si la anotación define un nombre de pool configurado para usar worker threads.

```java
@ApplicationScoped
public class MyBean {

    @Incoming("in")
    @Outgoing("out")
    @RunOnVirtualThread
    public String process(String s) {
        // Called on a new virtual thread for every incoming message
    }
}
```

En el mismo método

```java
@ApplicationScoped
@RunOnVirtualThread
public class MyBean {

    @Incoming("in1")
    @Outgoing("out1")
    public String process(String s) {
        // Called on the event loop - no @Blocking annotation
    }

    @Incoming("in2")
    @Outgoing("out2")
    @Blocking
    public String process(String s) {
        // Call on a new virtual thread for every incoming message
    }

    @Incoming("in3")
    @Outgoing("out3")
    @Blocking("my-worker-pool")
    public String process(String s) {
        // Called on a regular worker thread from the pool named "my-worker-pool"
    }
}
```

En la clase

Controlar el máximo de concurrencia

- Por defecto el máximo de concurrencia para métodos anotados con @RunOnVirtualThread es 1024.
- A diferencia de los platform threads, los virtual threads no se reutilizan y se crean por cada mensaje.
- El límite de concurrencia aplica independientemente a todos los métodos anotados con @RunOnVirtualThread.
- Se puede usar @RunOnVirtualThread junto con @Blocking para especificar un nombre de worker y ajustar la concurrencia.

```
@Incoming("prices")
@RunOnVirtualThread
@Blocking("my-worker")
public void consume(double price) {
    //...
}
```

smallrye.messaging.worker.my-worker.max-concurrency=30 //Ajusta el maximo de concurrencia a 30.

smallrye.messaging.worker.<virtual-thread>.max-concurrency= //Para cada método anotado con @RunOnVirtualThread que no tenga un nombre de worker configurado, se puede usar esta propiedad.