Microsoft

Microsoft Java Developer Conference 2024

Code. Cloud. Community.

Virtual Threads in Action

Daniel Kec

About me

Daniel Kec

Helidon developer
Oracle

@danielkec

@danielkec

@kec@mastodon.social

@kec.bsky.social

Agenda

- Quick Helidon introduction
- Optimizing server concurrency
- Helidon 3 – Reactive Programming
- Virtual Threads
- Helidon 4 – Virtual Threads in Action
- Pinning

Helidon Introduction

What is Helidon

- Framework for developing cloud-native Java (micro)services
- K8s friendly
- Helidon is 100% Open Source, available on GitHub
- Open source Support: GitHub, Slack, Stack Overflow

Helidon flavors

Helidon provides 2 programming models

Helidon SE

- Micro-framework
- Pure performance
- No Magic

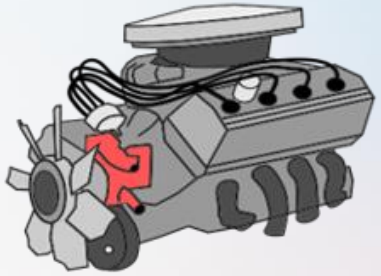
Helidon MP

- MicroProfile
- Declarative (IOC)
- CDI, JAXRS
- Jakarta APIs
- Helidon SE under the hood

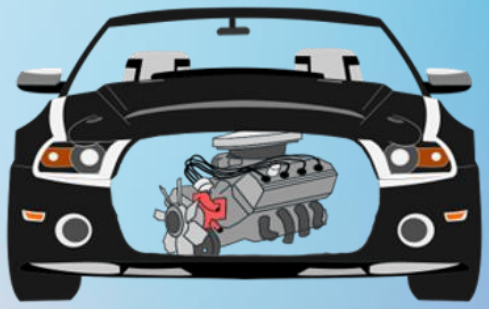
Helidon flavors

Helidon MP is under the hood powered by Helidon SE

Helidon SE



Helidon MP



Helidon flavors

Imperative vs. Declarative style

Helidon SE

```
WebServer.builder()
    .port(8080)
    .routing(r -> r
       .get("/greet", (req, res) ->
              res.send("Hello World!")))
    .build()
    .start();
```

Helidon MP

```
@Path("/greet")
public class GreetService {

    @GET
    public String getMsg() {
        return "Hello World!";
    }
}
```

Packaging

All easily containerizable and deployable to Kubernetes

Thin executable jar

```
COPY /target/libs ./libs
COPY /target/app.jar ./

CMD ["java", "-jar", "app.jar"]
```

Jlink image

74% size reduction

```
COPY /target/app-se-jri ./

ENTRYPOINT ["/bin/bash", "./bin/start"]
```

GraalVM

GraalVM Native image

88% size reduction

```
COPY /target/app .

ENTRYPOINT ["./app"]
```

Optimizing Concurrency

What problem do we solve?

- Heavily concurrent environment, usual for HTTP server
- Requirement to handle calls to other systems (database, messaging, other services [HTTP, grpc…])
- Requirement to return with low latency – requests are not designed to be long running
- Limited memory, CPU → limited number of platform threads
- Optimize, optimize, optimize …

Why is optimization so important?

Look at the bill from your cloud provider!

- **CPU cycles ….$$$**
- **Memory ……. $$$**
- **Storage …….. $$$**

Expensive Concurrency

- Java platform-threads are mapped one-to-one to the kernel threads
- Each kernel thread created by JVM needs megabytes of memory
- Kernel threads are scheduled by OS
- Starting new kernel thread is expensive!
- Context switching is expensive!

What can we do about it?

- Reusing threads - thread pools
- "Don't block the thread!" - Keep one thread busy, rather than multiple threads waiting

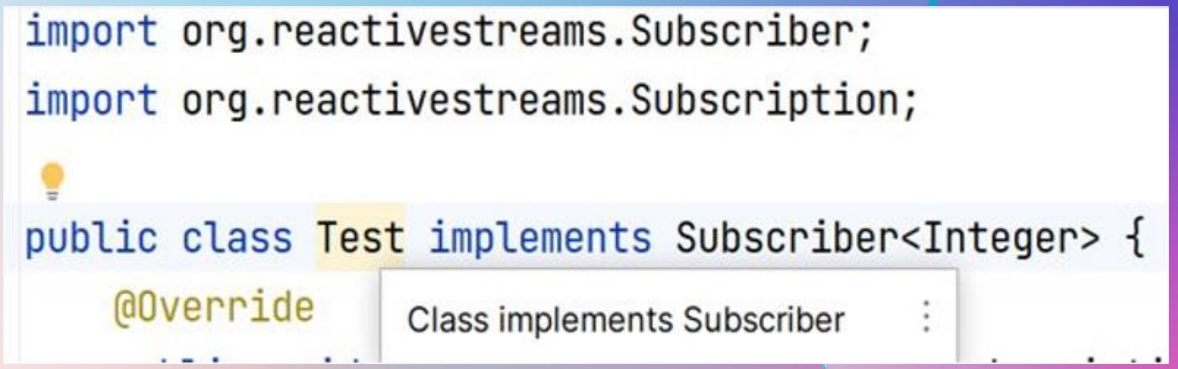
Reactive Programming

Reactive programming

- **Asynchronous** - we don’t wait for something to happen
- Just provide function to be called when it happens - callback function
- We have lost a flow control by giving up blocking, we need a means for backpressure control
- **Callback hell!**
- **Reactive Streams** API for callback orchestration

Reactive operators

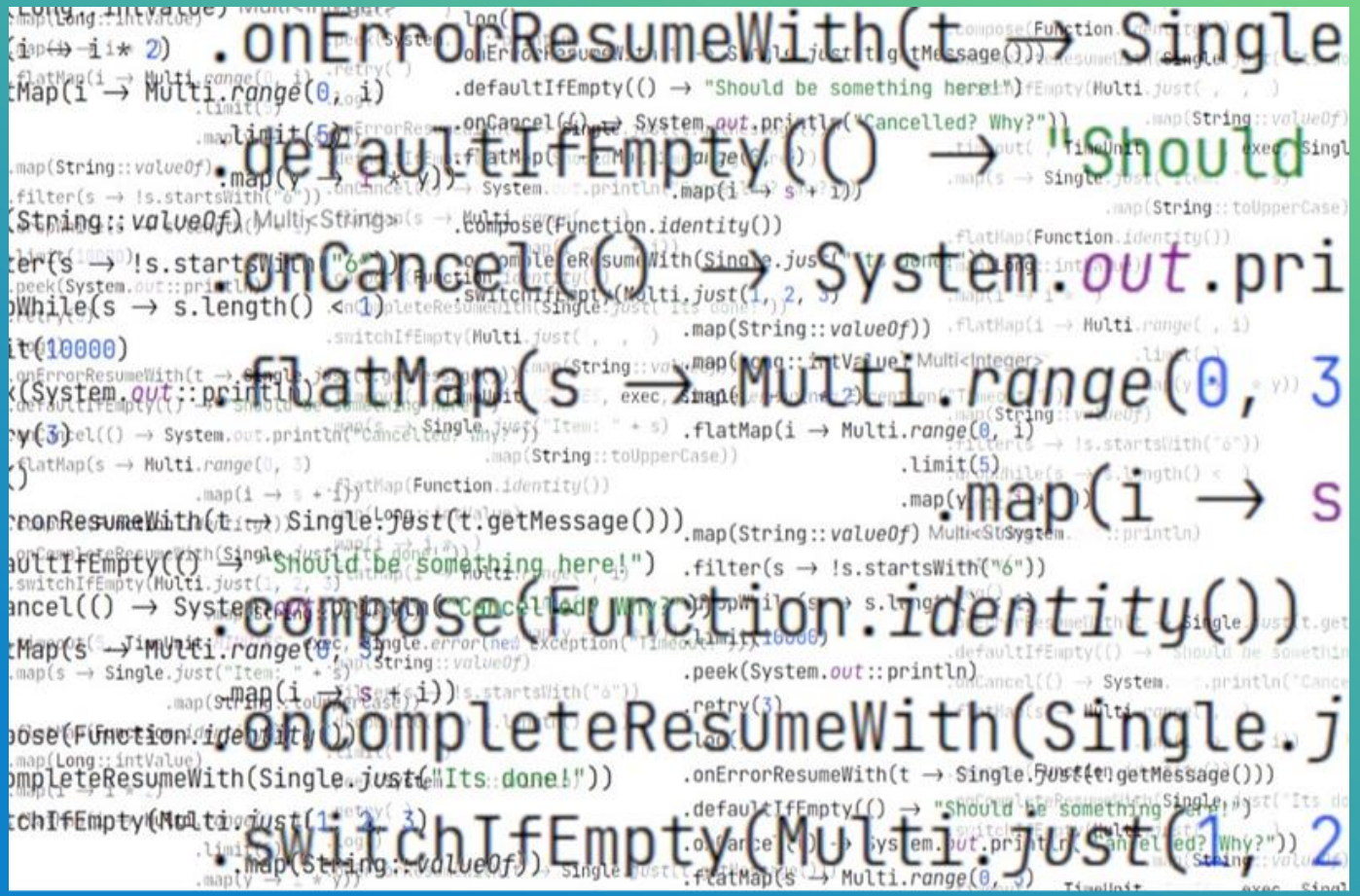
- **Reactive Streams** provides API for non-blocking back pressure control(request(1), request(5)...)
- Part of JDK since Java 9(Flow API)
- It's **hard to implement right**
- Reactive Streams spec rules are ridiculously complicated
- Even **IntelliJ warns you off!**



Reactive Streams implementations

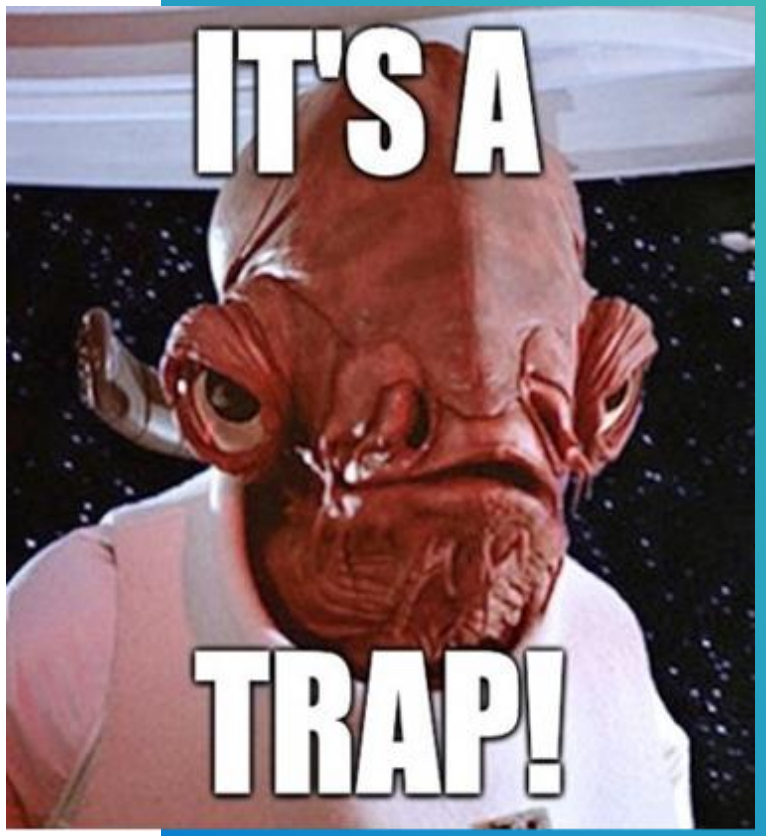
- Composable reactive operators
- RxJava
- Reactor
- Akka-Streams
- Service-Talk
- Helidon
- Mutiny
- So reactive operators are nice?

Reactive Operators



Reactive programming

- Steep learning curve
- Hard to get right™
  - Troubleshooting
  - No useful stack traces
  - More than one task in parallel is tough
- Using blocking code requires offloading
- “Callback Hell”



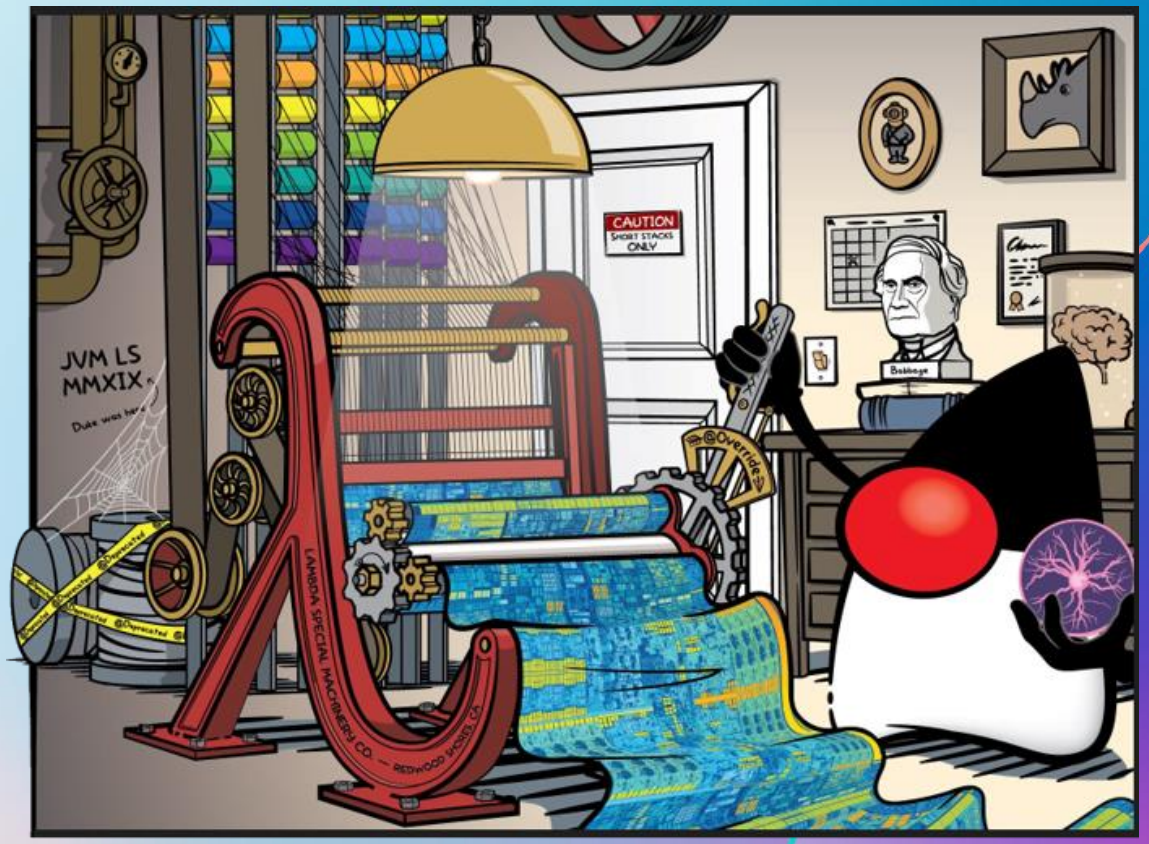
Virtual Threads

Better Solution?

- Virtual Threads (Part of project Loom)
  - JEP-425 Preview feature since Java 19
  - JEP-444 Delivered in Java 21 (September 2023)
- Threads can now be either Platform or Virtual
- Blocking operations do not block a platform/carrier thread
- Can have a huge number of virtual threads
- Useful stack traces
- “Naive” approach to coding Java is back (and safe)

Virtual Threads

- We can block cheaply!
- Imperative code can achieve performance comparable with reactive constructs
- Green threads again? - Not really!
- Yielding happens under the hood(sleep)



java.lang.Thread.sleep()

```
07.05.22 Bateman  498      public static void sleep(long millis) throws InterruptedException {
07.05.22 Bateman  499          if (millis < 0) {
07.05.22 Bateman  500              throw new IllegalArgumentException("timeout value is negative");
07.05.22 Bateman  501          }
07.05.22 Bateman  502
07.05.22 Bateman  503          long nanos = MILLISECONDS.toNanos(millis);
11.04.23 Bateman  504          ThreadSleepEvent event = beforeSleep(nanos);
07.05.22 Bateman  505          try {
11.04.23 Bateman  506              if (currentThread() instanceof VirtualThread vthread) {
11.04.23 Bateman  507                  vthread.sleepNanos(nanos);
07.05.22 Bateman  508              } else {
07.05.22 Bateman  509                  sleep0(millis);
07.05.22 Bateman  510              }
11.04.23 Bateman  511          } finally {
11.04.23 Bateman  512              afterSleep(event);
11.04.23 Bateman  513          }
07.05.22 Bateman  514      }
07.05.22 Bateman  515
```

java.lang.VirtualThread.sleepNanos(long nanos)

VirtualThread.java

```
791     void sleepNanos(long nanos) throws InterruptedException {
792         assert Thread.currentThread() == this && nanos >= 0;
793         if (getAndClearInterrupt())
794             throw new InterruptedException();
795         if (nanos == 0) {
796             tryYield();
797         } else {
798             // park for the sleep time
799             try {
800                 long remainingNanos = nanos;
801                 long startNanos = System.nanoTime();
802                 while (remainingNanos > 0) {
803                     parkNanos(remainingNanos);
804                     if (getAndClearInterrupt()) {
805                         throw new InterruptedException();
806                     }
807                     remainingNanos = nanos - (System.nanoTime() - startNanos);
808                 }
809             } finally {
810                 // may have been unparked while sleeping
811                 setParkPermit(true);
812             }
813         }
814     }
815
```

java.lang.VirtualThread.parkNanos(long nanos)

VirtualThread.java

```
616     void parkNanos(long nanos) {
617         assert Thread.currentThread() == this;
618
619         // complete immediately if parking permit available or interrupted
620         if (getAndSetParkPermit(false) || interrupted)
621             return;
622
623         // park the thread for the waiting time
624         if (nanos > 0) {
625             long startTime = System.nanoTime();
626
627             boolean yielded = false;
628             Future<?> unparker = scheduleUnpark(this::unpark, nanos);
629             setState(PARKING);
630             try {
631                 yielded = yieldContinuation();  // may throw
632             } finally {
633                 assert (Thread.currentThread() == this) && (yielded == (state() == RUNNING));
634                 if (!yielded) {
635                     assert state() == PARKING;
636                     setState(RUNNING);
637                 }
638                 cancel(unparker);
639             }
640
641             // park on carrier thread for remaining time when pinned
642             if (!yielded) {
643                 long deadline = startTime + nanos;
644                 if (deadline < 0L)
645                     deadline = Long.MAX_VALUE;
646                 parkOnCarrierThread(true, deadline - System.nanoTime());
647             }
648         }
649     }
658
```

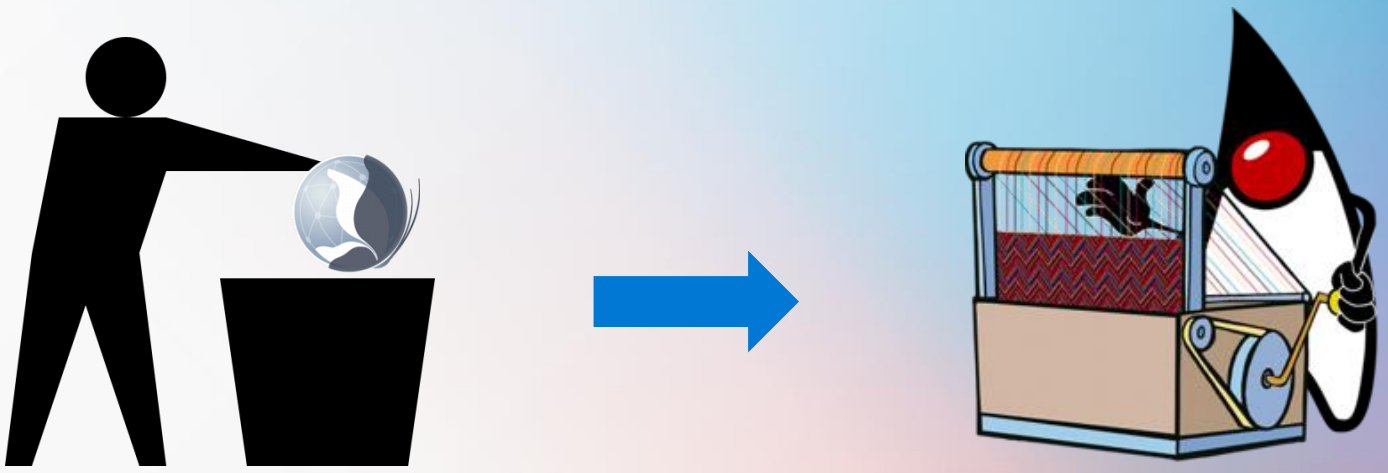
Continuations in Java!

https://youtu.be/6nRS6UiN7X0

Helidon 4

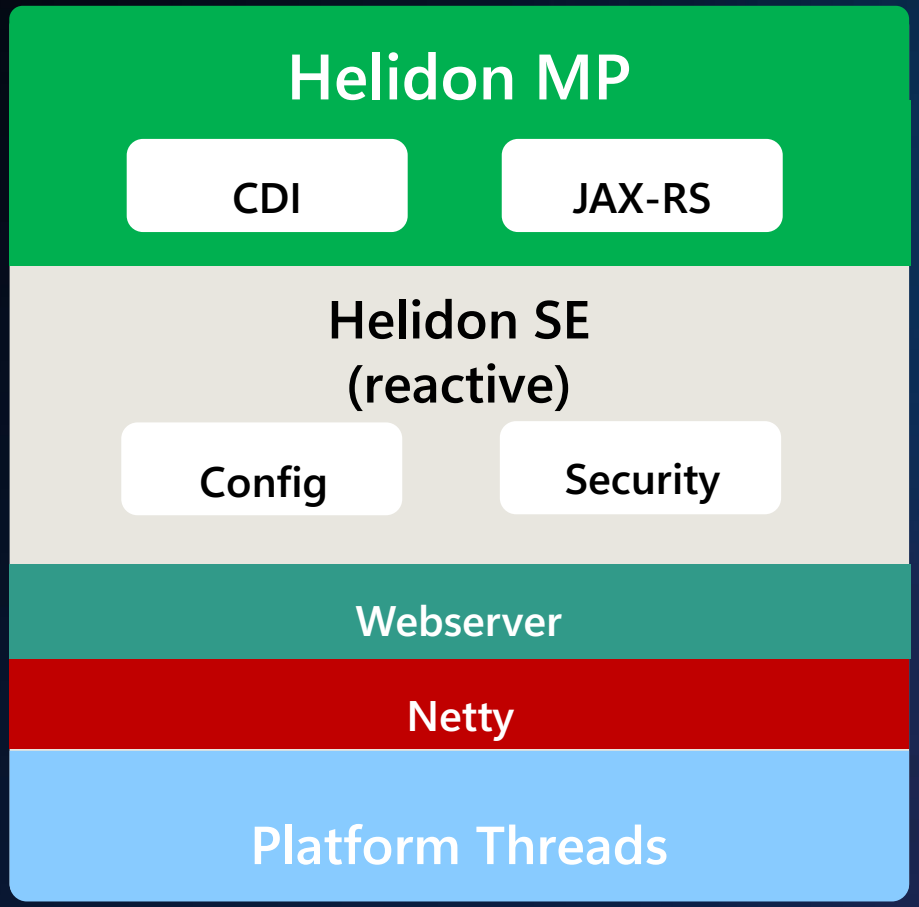
Helidon 4

- Requires - Java 21
- Netty replaced with custom Web Server (Project Níma)
  - Designed for Virtual Threads
  - Created in cooperation with the Java team
  - Performance comparable to Netty
  - Heart of Helidon 4 release

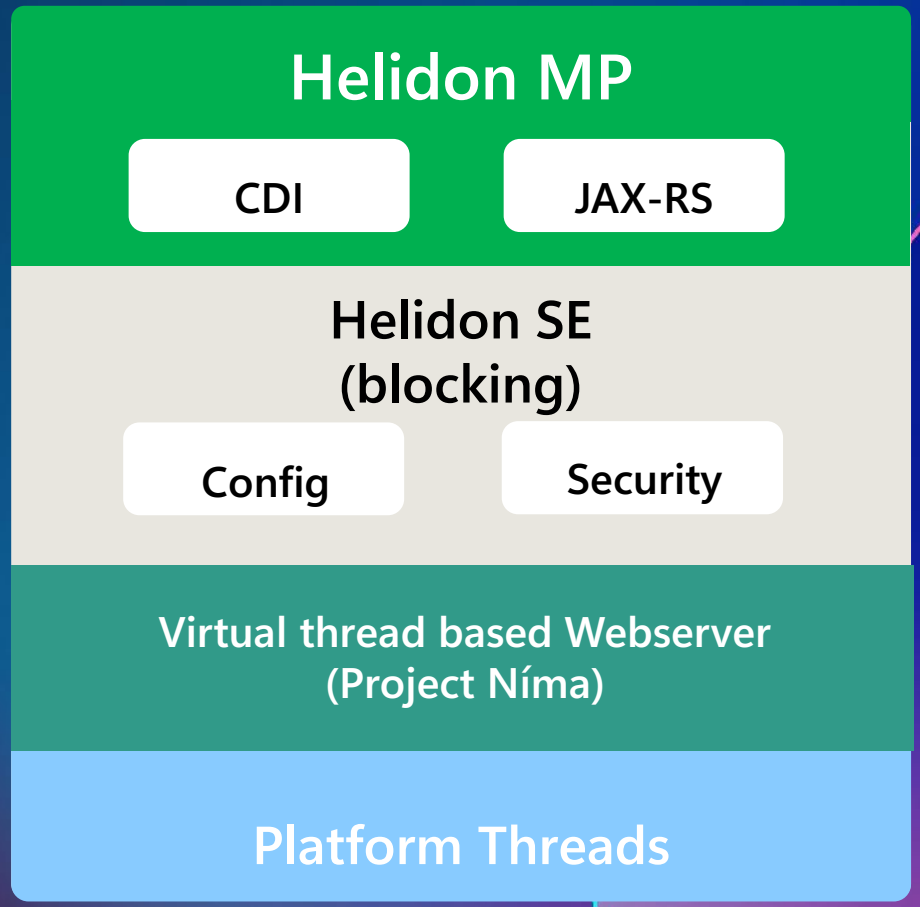


Architecture

Helidon 1.x, 2.x, 3.x



Helidon 4.x



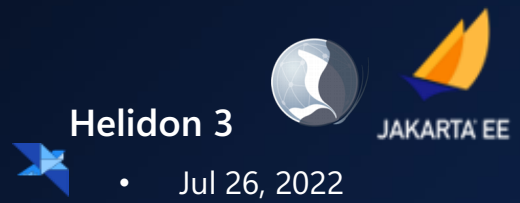
Helidon features timeline

Helidon 1

- Feb 14, 2019
- **Netty** based Web Server
- JDK >8
- **Javax** based MP
  - MicroProfile 3.2
  - Java EE 8

Helidon 2

- Jun 25, 2020
- **Netty** based Web Server
- JDK >11
- **Javax** based MP
  - MicroProfile 3.3
  - Jakarta EE 8



Helidon 3

- Jul 26, 2022
- **Netty** based Web Server
- JDK >17
- **Jakarta** based MP
  - MicroProfile 5
  - Jakarta EE 9.1

Java 21

- Sep 19, 2023
- JEP 444 – **Virtual Threads**

Helidon 4

- Oct 24, 2023
- **Virtual Thread based** Web Server (Project Níma)
- **Jakarta** based MP

Helidon 4 Performance

TechEmpower Web Framework Benchmark

TechEmpower

2023-10-17
Round 22

Composite Framework Scores

Each framework's peak performance in each test type (shown in the colored columns below) is multiplied by the weights shown above. The results are then summed to yield a weighted score. Only frameworks that implement all test types are included. 159 total frameworks ranked, 5 visible, 154 hidden by filters. See filter panel above.

Rnk	Framework	JSON	1-query	20-query	Fortunes	Updates	Plaintext	Weighted score
37	helidon	429,240	268,833	30,291	238,545	9,390	3,035,006	**3,664** 45.3%
38	quarkus	903,185	318,897	17,610	214,275	6,697	2,861,479	**3,637** 45.0%
40	micronaut	568,955	221,741	28,171	179,741	15,209	1,327,013	**3,555** 44.0%
81	dropwizard	170,910	75,821	17,933	54,065	9,674	208,744	**1,608** 19.9%
88	spring	236,259	147,907	15,932	24,082	7,131	506,087	**1,507** 18.6%

https://www.techempower.com/benchmarks/#hw=ph&test=composite§ion=data-r22&f=zijunz-zik0zj-zik0zj-zik0zj-zik0zj-zik0zj-zik0zj-zik0zj-v2qiv3-xamxa7-zik0zj-zik0zj-zik0zj-zik0zj-zik0zj-1ekf

No Reactive layer

Helidon Webserver

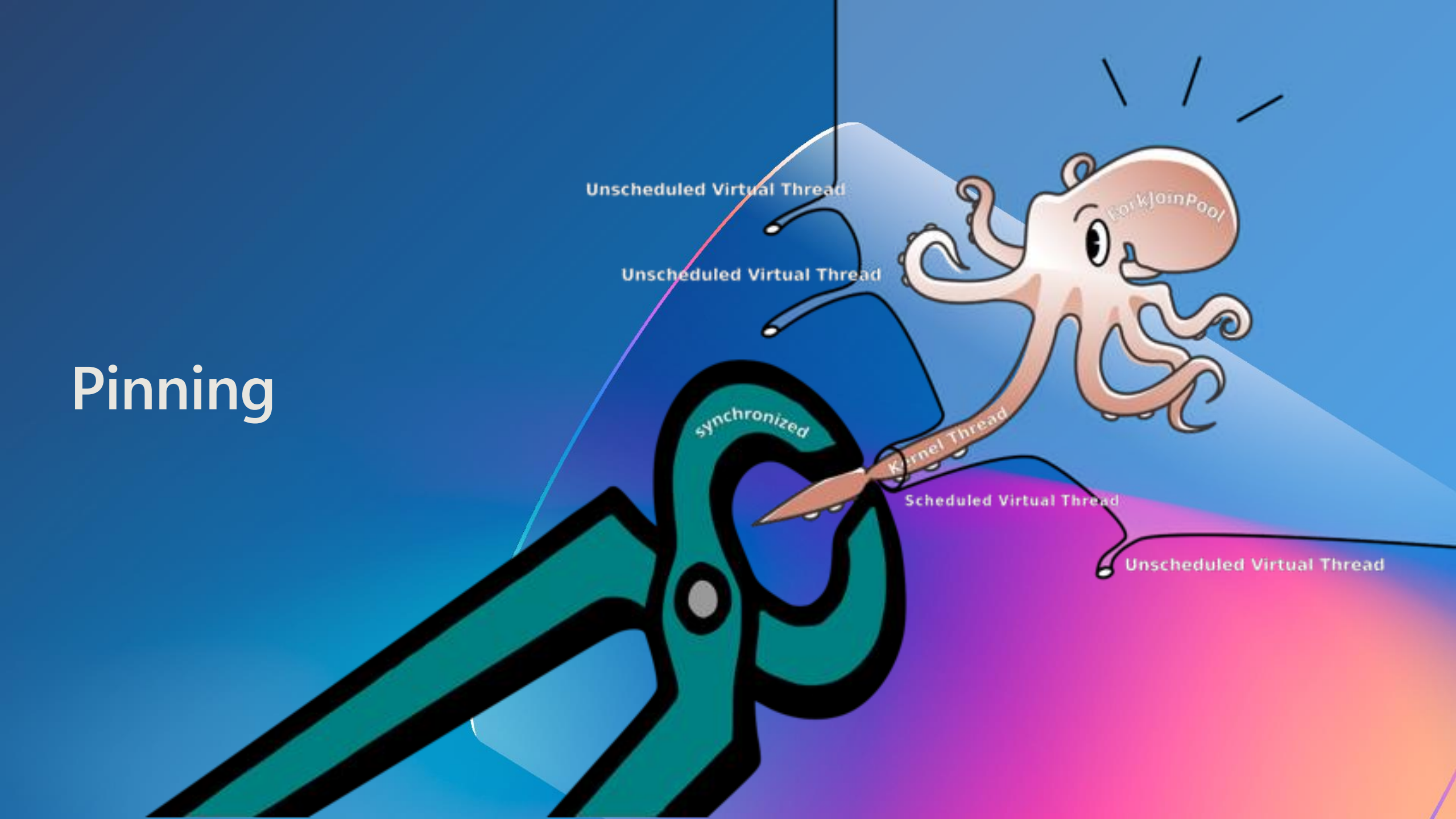
Netty

Project Níma

No Netty!

- Got rid of numerous Netty CVEs

Pinning


Unscheduled Virtual Thread
Unscheduled Virtual Thread
ForkJoinPool
synchronized
Kernel Thread
Scheduled Virtual Thread
Unscheduled Virtual Thread

Pinning

- Usual suspect is usage of synchronized
  - Not always harmful
  - Short-lived operations like in-memory operations are not harmful
- Carrier thread pool compensates by adding new carrier thread
  - Leads to degraded performance in case it happens frequently
- Usage of ReentrantLock does NOT cause pinning
  - ReentrantLock is VirtualThread friendly

Pinning example

```
public class Main {
    public static void main(String[] args) throws InterruptedException {
        Thread.ofVirtual().start(() -> {
            synchronized (new Main()) {
                try {
                    Thread.sleep(100);
                } catch (InterruptedException e) {}
            }
        }).join();
    }
}
```

Pinning Detection #1

jdk.tracePinnedThreads system property

- Easy to use
- **-Djdk.tracePinnedThreads=short** prints just problematic frame
- Not recommended for production use with Helidon

```
➜ java -Djdk.tracePinnedThreads Main.java
Thread[#29,ForkJoinPool-1-worker-1,5,CarrierThreads]
    java.base/java.lang.VirtualThread$VThreadContinuation.onPinned(VirtualThread.java:183)
    java.base/jdk.internal.vm.Continuation.onPinned0(Continuation.java:393)
    java.base/java.lang.VirtualThread.parkNanos(VirtualThread.java:621)
    java.base/java.lang.VirtualThread.sleepNanos(VirtualThread.java:793)
    java.base/java.lang.Thread.sleep(Thread.java:507)
    me.daniel.se.quickstart.Main.lambda$main$0(Main.java:8) <== monitors:1
    java.base/java.lang.VirtualThread.run(VirtualThread.java:309)
```

Pinning Detection #2

JDK Flight Recorder (JFR) **jdk.VirtualThreadPinned** event

- Easy to use
- Enabled by default on when operation takes longer 20ms

```
➜ java -XX:StartFlightRecording:jdk.VirtualThreadPinned#enabled=true,filename=pinning.jfr Main.java

➜ jfr print --events jdk.VirtualThreadPinned pinning.jfr
jdk.VirtualThreadPinned {
  startTime = 15:28:37.594 (2024-03-01)
  duration = 99.1 ms
  eventThread = "" (javaThreadId = 32, virtual)
  stackTrace = [
    java.lang.VirtualThread.parkOnCarrierThread(boolean, long) line: 677
    java.lang.VirtualThread.parkNanos(long) line: 636
    java.lang.VirtualThread.sleepNanos(long) line: 793
    java.lang.Thread.sleep(long) line: 507
    me.daniel.se.quickstart.Main.lambda$main$0() line: 8
    ...
```

Future of synchronized

- Frameworks and libraries are replacing synchronized
- Pinning-less synchronize in Java is just around the corner

Thank you!

@helidon_project

helidon.io/nima

medium.com/helidon

github.com/helidon-io/helidon

youtube.com/Helidon_Project

helidon.slack.com