

# VIRTUAL THREADS

SIMPLE YET SCALABLE CONCURRENCY IN JAVA 19

[BJORN.BESKOW@CALLISTAENTERPRISE.SE](mailto:BJORN.BESKOW@CALLISTAENTERPRISE.SE)

CADEC 2023.01.19 & 2023.01.25 | [CALLISTAENTERPRISE.SE](https://callistaenterprise.se)

CALLISTA

## THREADS IN JAVA: PROGRAMMING MODEL FROM JDK 1.1 (1997)

```
Thread t1 = new Thread(() -> {  
    ...  
});  
Thread t2 = new Thread(() -> {  
    ...  
});  
  
t1.run();  
t2.run();
```

## THREADS PROGRAMMING MODEL

```
public ProductValue getProduct(long productId) {
    Product product = productRepository.findByProductId(productId);
    if (product != null) {
        InventoryValue inventoryValue = inventoryService.getInventory(product.getSku());
        return ProductValue.fromEntity(product, inventoryValue.getStock());
    } else {
        throw new EntityNotFoundException("Product " + productId + " not found");
    }
}
```

Illusion of “sequential flow”

Call stack variables

Call stack exception handling

Single-step debugging

= fairly simple

## THREADS PROGRAMMING MODEL: THREAD CONTEXT

```
private static final InheritableThreadLocal<String> currentTenant =  
    new InheritableThreadLocal<>();
```

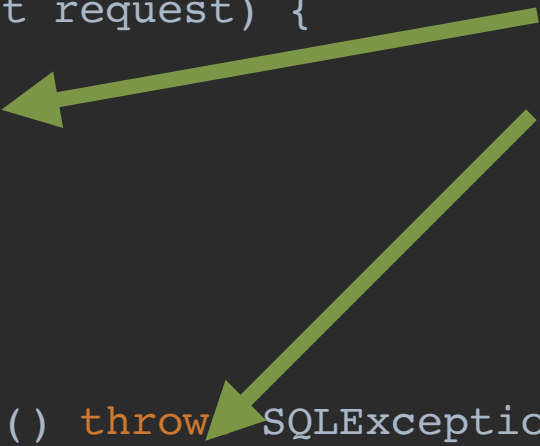
...

```
public void preHandle(WebRequest request) {  
    String tenantId = ...;  
    currentTenant.set(tenantId);  
}
```

...

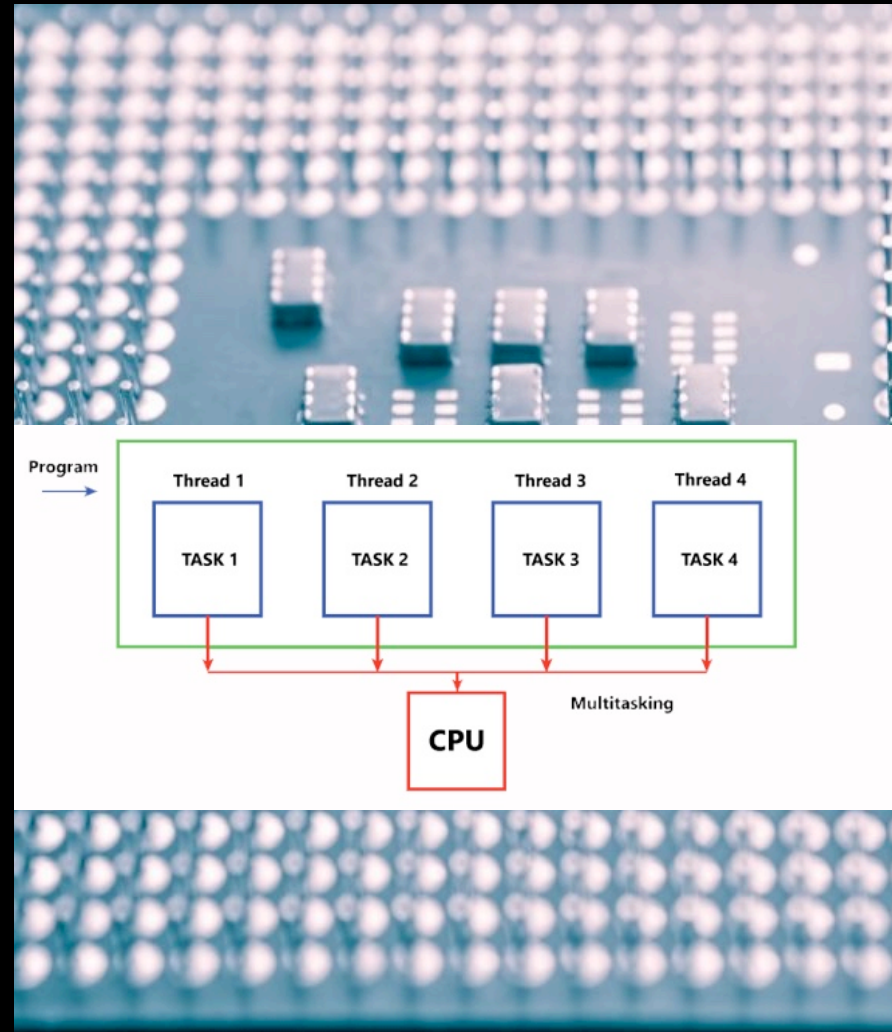
```
public Connection getConnection() throws SQLException {  
    String tenantId = currentTenant.get();  
    ...  
}
```

Transparently pass  
information along  
in the thread context



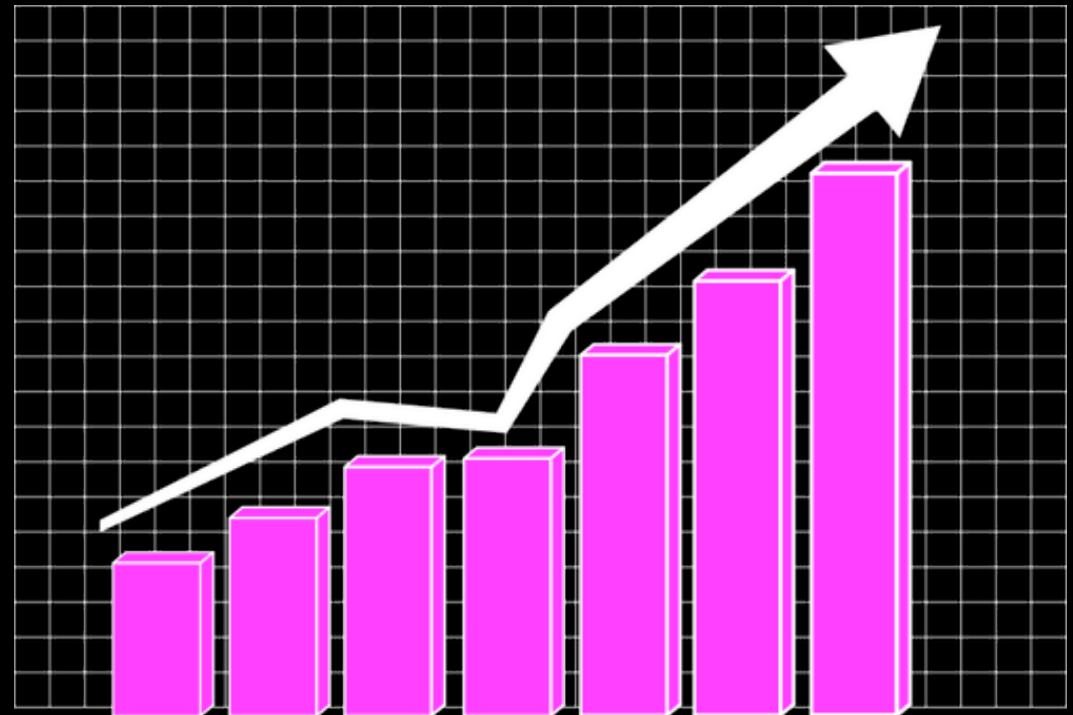
## PLATFORM / OS THREADS

- Reliable scheduling mechanism
- Thread stacks are monolithic blocks of memory
- Relatively expensive, and hence limited



## INTERNET SCALE

- 1999:
  - The Internet had 280 million users
- 2005:
  - 1 billion Internet users.
  - Facebook had 5.5 million users.
- 2014:
  - 2.95 billion Internet users.
  - Facebook 1.3 billion users.
  - Twitter 270 million users.
- 2022
  - 5 billion Internet users.
  - Facebook 2.9 billion users.

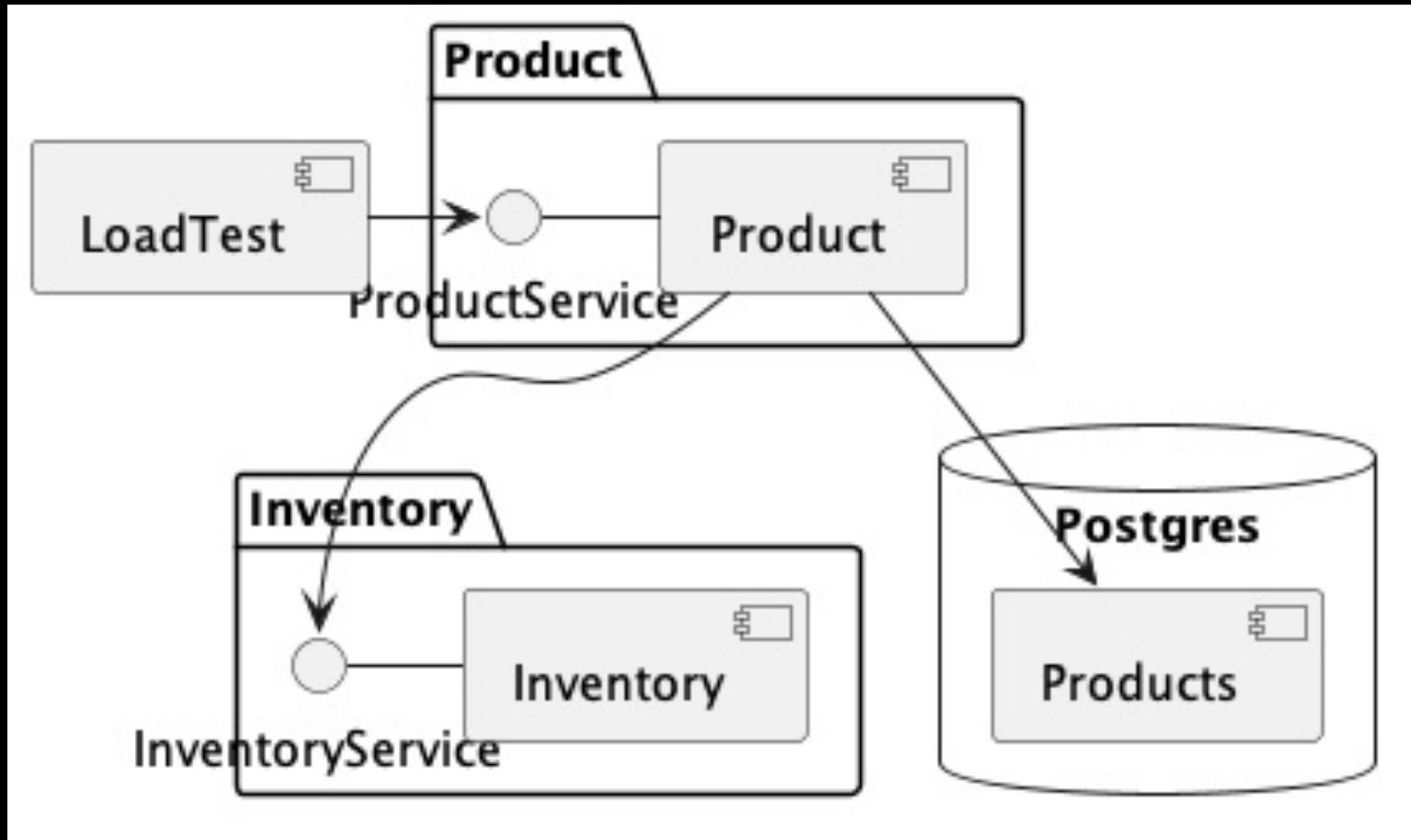


[HTTPS://WWW.STATISTA.COM/STATISTICS/617136/DIGITAL-POPULATION-WORLDWIDE/](https://www.statista.com/statistics/617136/digital-population-worldwide/)

## THREAD PER TASK

```
@GetMapping(value = "/products/{productId}")
public ResponseEntity<ProductValue> getProduct(@PathVariable long productId) {
    try {
        ProductValue product = productService.getProduct(productId);
        return new ResponseEntity<>(product, HttpStatus.OK);
    } catch (EntityNotFoundException e) {
        throw new NotFoundException(e.getMessage());
    }
}
```

## DEMO: EXAMPLE SYSTEM



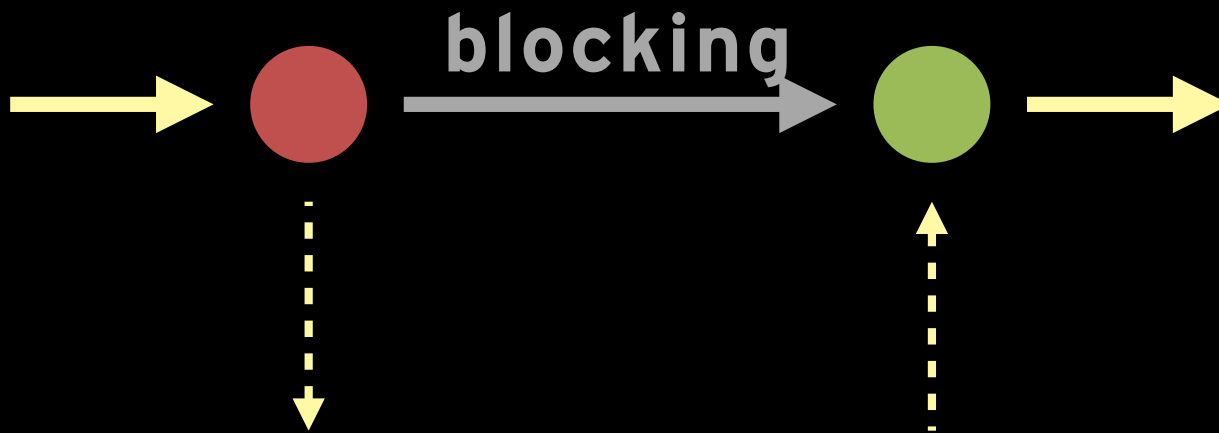
<https://github.com/callistaenterprise/cadec-2023-virtual-threads.git>



## BLOCKING THREADS

# Blocking!

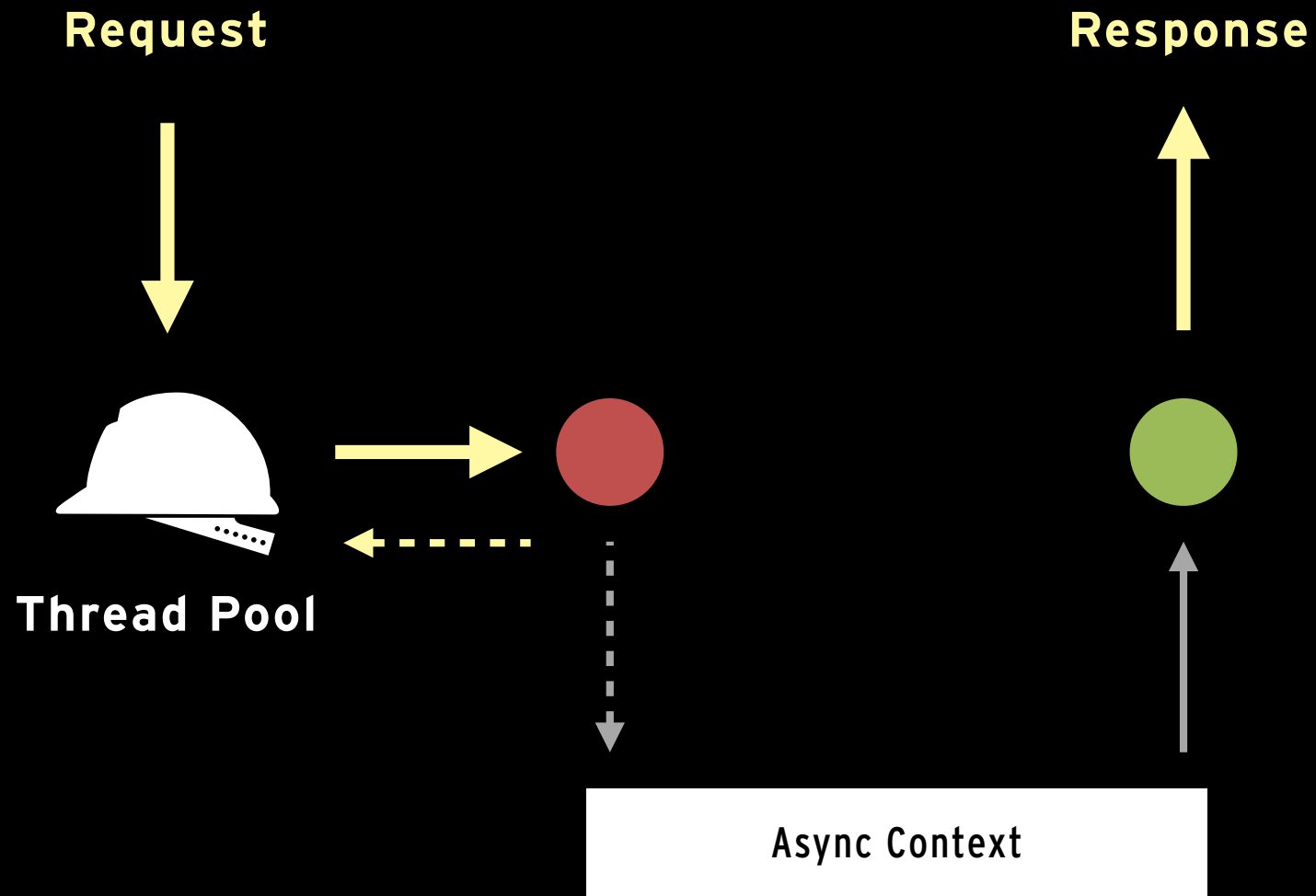
```
public ProductValue getProduct(long productId) {  
    // Database call will block  
    Product product = productRepository.findById(productId);  
    if (product != null) {  
        // Rest call will block  
        InventoryValue inventoryValue = inventoryService.getInventory(product.getSku());  
        return ProductValue.fromEntity(product, inventoryValue.getStock());  
    } else {  
        throw new EntityNotFoundException("Product " + productId + " not found");  
    }  
}
```



## LITTLE'S LAW

$$T = N / d$$

# NON-BLOCKING / ASYNC PROGRAMMING MODEL



## NON-BLOCKING PROGRAMMING MODEL: callbacks

```
public void getProduct(long productId, ProductServiceCallback callback) {
    productRepository.findById(productId, product -> {
        inventoryService.getInventory(product.getSku(), inventory -> {
            ProductValue result = ProductValue.fromEntity(product, inventory.getStock());
            callback.handle(result);
        });
    });
}
```

## NON-BLOCKING PROGRAMMING MODEL: `CompletableFuture`

```
public CompletableFuture<ProductValue> getProduct(long productId) {
    return productRepository.findById(productId).thenCompose(product -> {
        return inventoryService.getInventory(product.getSku()).thenApply(inventory -> {
            return ProductValue.fromEntity(product, inventory.getStock());
        });
    });
}
```

## NON-BLOCKING PROGRAMMING MODEL: RxJava

```
public Single<ProductValue> getProduct(long productId) {  
    return productRepository.findById(productId).flatMap(product ->  
        inventoryService.getInventory(product.getSku()).map(inventory ->  
            ProductValue.fromEntity(product, inventory.getStock()))  
    )  
};  
}
```

## NON-BLOCKING PROGRAMMING MODEL: Spring WebFlux

```
public Mono<ProductValue> getProduct(long productId) {  
    return productRepository.findById(productId).flatMap(product ->  
        inventoryService.getInventory(product.getSku()).map(inventory ->  
            ProductValue.fromEntity(product, inventory.getStock()))  
    )  
};  
}
```

## NON-BLOCKING PROGRAMMING MODEL: Shortcomings

```
public Mono<ProductValue> getProduct(long productId) {  
    return productRepository.findById(productId).flatMap(product ->  
        inventoryService.getInventory(product.getSku()).map(inventory ->  
            ProductValue.fromEntity(product, inventory.getstock()))  
    )  
};  
}
```

Obscure syntax

No sequential flow

No ThreadLocal

= Complexity

Akward error handling

No single-step debugging



## IMPACTS ALL APIS: e.g. JDBC/HIBERNATE/JPA -> R2DBC

```
import io.r2dbc.spi.ConnectionFactory;

...

@Bean
public ConnectionFactory connectionFactory() {
    ConnectionFactory connectionFactory = ConnectionFactoryBuilder.build();
    return new TenantAwareConnectionFactory(connectionFactory);
}

@Bean
@Primary
ReactiveTransactionManager connectionFactoryTransactionManager() {
    return new R2dbcTransactionManager(connectionFactory());
}
```

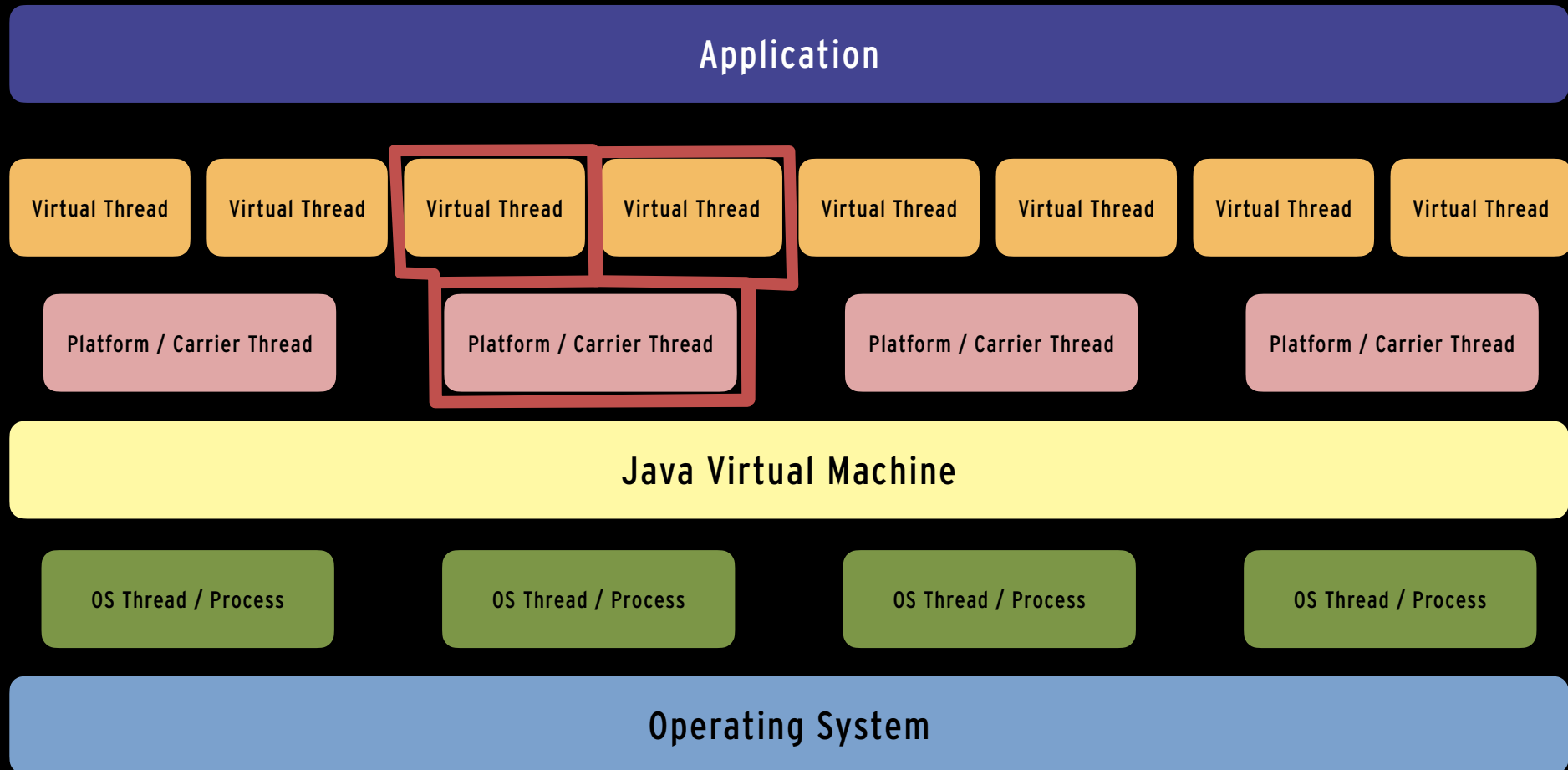


## JEP 425: Virtual Threads (Preview)

- Installing
- Contributing
- Sponsoring
- Developers' Guide
- Vulnerabilities
- JDK GA/EA Builds
- Mailing lists
- Wiki · IRC
- Bylaws · Census
- Legal
- JEP Process**
- Source code**
- Mercurial
- GitHub
- Tools**
- Mercurial
- Git
- jtreg harness

<i>Authors</i>	Ron Pressler, Alan Bateman
<i>Owner</i>	Alan Bateman
<i>Type</i>	Feature
<i>Scope</i>	SE
<i>Status</i>	Candidate
<i>Component</i>	core-libs
<i>Discussion</i>	loom dash dev at openjdk dot java dot net
<i>Effort</i>	XL
<i>Reviewed by</i>	Alex Buckley, Brian Goetz, Chris Hegarty
<i>Created</i>	2021/11/15 16:43
<i>Updated</i>	2022/04/11 11:03
<i>Issue</i>	<a href="#">8277131</a>

# VIRTUAL THREADS



## VIRTUAL THREADS: SAME PROGRAMMING MODEL

```
Thread.startVirtualThread(() -> {  
    ...  
});
```

```
var executor = Executors.newVirtualThreadPerTaskExecutor();  
executor.submit(() -> {  
    ...  
});
```

## VIRTUAL THREADS IN SPRING BOOT 3

```
@Bean
AsyncTaskExecutor applicationTaskExecutor() {

    ExecutorService executorService = Executors.newVirtualThreadPerTaskExecutor();
    return new TaskExecutorAdapter(executorService::execute);
}
```

## VIRTUAL THREADS IN TOMCAT 10.1.X

```
@Bean
TomcatProtocolHandlerCustomizer<?> protocolHandlerExecutorCustomizer() {

    return protocolHandler ->
        protocolHandler.setExecutor(Executors.newVirtualThreadPerTaskExecutor());
}
```

## BEST PRACTICES: DON'T USE `synchronised`

- The current implementation of virtual threads cause `synchronised` blocks to be *pinned* to the underlying carrier thread, which may cause blocking
- Use `ReentrantLock` instead, if needed

```
private final ReentrantLock REENTRANT_LOCK = new ReentrantLock();

public InventoryValue getInventory(String sku) {
    try {
        REENTRANT_LOCK.lock();
        ...
    } finally {
        REENTRANT_LOCK.unlock();
    }
}
```

```
<jvmArguments>-Djdk.tracePinnedThreads=full</jvmArguments>
```

## BEST PRACTICES: DON'T USE THREAD POOLS

- Virtual Threads are cheap - don't try to reuse them
- Use **Semaphores** instead, if there is a need to limit use of other scarce resources

```
private final Semaphore INVENTORY_SEMAPHORE;  
  
public InventoryServiceImpl(@Value("${inventory.session.max:1000}") int permits) {  
    INVENTORY_SEMAPHORE = new Semaphore(permits);  
}  
  
public InventoryValue getInventory(String sku) {  
    try {  
        INVENTORY_SEMAPHORE.acquire();  
        ...  
    } finally {  
        INVENTORY_SEMAPHORE.release();  
    }  
}
```



## BEST PRACTICES: DON'T OVERUSE `ThreadLocal`

- While Virtual Threads are plentiful, multiple or heavy-weight `ThreadLocal` variables may alter the equation and become a bottleneck
- Harmless when used wisely

```
private static final InheritableThreadLocal<String> currentTenant =  
    new InheritableThreadLocal<>();  
  
...  
  
public Connection getConnection() throws SQLException {  
    String tenantId = currentTenant.get();  
    ...  
}
```

## VIRTUAL THREADS: CURRENT STATUS

- First Preview release in Java 19:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-compiler-plugin</artifactId>
  <configuration>
    <compilerArgs>--enable-preview</compilerArgs>
  </configuration>
</plugin>
```

- Second Preview in Java 20 (March 2023), final release in Java 21? (September 2023)
- Fully functional support in Spring Boot 3 / Spring Framework 6
- Fully functional support in Tomcat 10.1
- Fully functional support in Oracle JDBC 21c
- Preliminary support in Quarkus, Microsoft JDBC
- Ongoing work in many supporting libraries/frameworks (HikariCP, PostgresJDBC, ...)

## SUMMARY

- Virtual Threads allow applications using the familiar “Thread-per-Task” model to achieve the **same high scalability** as *Async/Non-blocking* applications, but **without the complexity of a new programming model** and developer experience.
- The Reactive stack may **still be a better option** when the unique features of e.g. Reactive Streams (**parallelism, back-pressure, cancellation** etc.) are required.



# Time for questions?



[BJORN.BESKOW@CALLISTAENTERPRISE.SE](mailto:BJORN.BESKOW@CALLISTAENTERPRISE.SE)