Virtual Threads in Spring Boot with Tomcat

Q&A; Document - October 22, 2025

Question 1: How do virtual threads work on Spring Boot with Tomcat?

Virtual threads in Spring Boot with Tomcat provide a way to handle massive concurrency efficiently.

The Basics

Virtual threads (from Project Loom, Java 21+) are lightweight threads managed by the JVM rather than the OS. Unlike traditional platform threads that map 1:1 to OS threads, thousands or even millions of virtual threads can run on a small number of platform threads.

How It Works in Spring Boot + Tomcat

1. Request Handling Flow

When you enable virtual threads in Spring Boot:

• Tomcat receives an HTTP request
• Instead of assigning it to a traditional platform thread from the thread pool, it creates/uses a virtual thread
• That virtual thread handles the entire request lifecycle
• When the thread hits a blocking operation (I/O, database call, sleep), it doesn't block a platform thread—it gets unmounted
• The underlying platform thread is freed to do other work
• When the I/O completes, the virtual thread remounts on any available platform thread

2. Configuration

In Spring Boot 3.2+, you can enable virtual threads simply:

```
spring.threads.virtual.enabled=true
```

This tells Spring Boot to use virtual threads for: Tomcat's request processing, @Async methods, and scheduled tasks.

3. Benefits with Tomcat

Traditional Tomcat uses a thread pool (default ~200 threads). With virtual threads:

• You can handle thousands of concurrent requests without increasing the thread pool
• No more thread pool exhaustion under high load
• Blocking I/O operations (database queries, HTTP calls, file I/O) no longer waste threads
• Particularly beneficial for I/O-heavy applications

4. What Doesn't Change

Your application code remains the same! You can still write blocking code naturally—no need for reactive programming. The virtual thread magic happens at the JVM level.

Key Point: Virtual threads shine when your app does lots of blocking I/O. For CPU-intensive work, they don't provide much benefit over platform threads.

Question 2: Performance Comparison Across Three Scenarios

Scenario: Mix of fast requests (~1ms) and rare slow requests (60s) on 64 vCPUs

Scenario A: Spring Boot 3.5.6 + Tomcat + Virtual Threads + 64 vCPUs

Scenario B: Spring Boot 3.5.6 + Tomcat (no virtual threads, thread-per-request model) + 64 vCPUs

Scenario C: Spring Boot 3.5.6 + Netty with I/O worker pool = 64 and Hikari CP pool = 64 + 64 vCPUs

Ranking: Best to Worst

■ Scenario A: Virtual Threads - BEST

Why it wins:

• When slow requests block (DB, I/O), virtual threads unmount and free the underlying platform thread
• Fast requests continue processing unimpeded on those freed platform threads
• Can handle thousands of concurrent slow requests without thread exhaustion
• Zero code changes needed—write simple blocking code
• With 64 vCPUs, you have 64 carrier threads that can efficiently multiplex thousands of virtual threads

Trade-offs:

• Slight overhead for virtual thread creation/scheduling (negligible)
• Requires Java 21+

■ Scenario C: Netty + Worker Pool - MIDDLE (with a BIG caveat)

The problem: Hikari CP is a JDBC connection pool—that's blocking I/O. Using blocking JDBC calls with Netty is an anti-pattern that kills performance.

IF using R2DBC (non-blocking DB driver):

• Good: 64 event loop threads efficiently handle thousands of concurrent requests
• Slow requests don't block worker threads (everything is async/reactive)
• Excellent throughput for I/O-bound workloads

IF using JDBC (blocking):

• Terrible: Each slow 60s request blocks one of your 64 worker threads
• Just a few slow requests can starve your entire event loop
• Fast requests get queued behind blocked workers
• This scenario becomes worse than platform threads

Complexity cost:

• Requires reactive programming (Reactor, WebFlux, R2DBC)
• Much harder to write and debug
• Any blocking call (JDBC, blocking APIs) breaks the model

■ Scenario B: Platform Threads - WORST

Why it struggles:

• Default Tomcat thread pool ≈ 200 threads
• Each 60s slow request holds a thread for the entire duration
• With only ~4 concurrent slow requests, you've tied up 240 seconds of thread time
• If slow requests spike even slightly, thread pool exhaustion occurs
• Fast requests get queued waiting for available threads

You could tune it:

• Increase thread pool to 1000+ threads
• But now you're wasting memory and context-switching overhead
• Still fundamentally inefficient—threads just sitting idle during I/O

The Verdict

For your scenario: A > C > B

Virtual Threads (A) is the clear winner because:

1. ✓ Handles slow requests gracefully without blocking threads
2. ✓ Fast requests stay fast
3. ✓ Simple blocking code (no reactive complexity)
4. ✓ Efficient resource utilization

Scenario C could be competitive if you're using a fully non-blocking stack (R2DBC, reactive HTTP clients), but the complexity cost is high. If you're using JDBC (Hikari suggests you are), it's actually worse than platform threads.

Scenario B is the worst here specifically because slow requests starve the thread pool, impacting fast requests.