The definition from related JPE 425 (JDK Enhancement Proposal):  
“*Virtual threads are lightweight threads that dramatically reduce the effort of writing, maintaining, and observing high-throughput concurrent applications.”*

The page of JEP <https://openjdk.org/jeps/425> provides a great description of the feature, I highly recommend you to read it. Here I will just provide general information and an example of using “Virtual threads” with Spring Boot 2.7 and 3.0.

Let’s start with the definition of thread: a thread in Java is a wrapper of operating system (OS) thread. When we run a Java program (e.g Spring Boot app), the main method is executed by the “main” thread, which is created on the program launching. OS threads are costly, so we cannot have too many of them.

Besides a main thread, Spring Boot has many different threads created for different thread pools, such as:  
- thread pool of Tomcat to handle incoming HTTP requests  
- thread pool for asynchronous methods marked with @Async  
- thread pool for scheduled tasks  
- etc

Let’s configure the Tomcat thread pool to be a pool of virtual threads. We are going to test it in two different versions of Spring Boot: 2.7.10 and 3.0.5 and compare the results. Virtual threads were added in Java starting from Java19, thus we will use this version.

The test is simple, we have a simple rest endpoint that runs *Thread.sleep(5000)* operation inside. And we will call this endpoint 1000 times simultaneously and see how long it will take.

Why a sleep operation has been chosen? Because Virtual threads are good for not CPU bounded operations (e.g Network I/O operations), sleep is a good candidate to simulate such operations.

By default, Tomcat in Spring Boot has a thread pool consisting of a maximum of 200 threads. It means that If your web application simultaneously receives more than 200 HTTP requests, then some of them will be put in a queue waiting for available threads. If we run our test on this standard Tomcat configuration in Spring Boot the result will be predictable: we have only 200 threads available for 1000 requests, and each of the 200 threads will handle 5 requests (Number of requests / the Tomcat thread pool size = 1000 / 200) \* 5 seconds = 25 seconds.

Let’s try it by using Apache Bench to execute 1000 parallel requests:

*ab -n 1000 -c 1000*[*http://localhost:8085/test*](http://localhost:8085/test)

Results are the same for Spring Boot 2.7.10 and 3.0.5:

Finished 1000 requests

Concurrency Level: 1000  
Time taken for tests: 25.165 seconds  
Complete requests: 1000  
Failed requests: 0

The time taken is 25 seconds as we predicted.

Now we configure the Tomcat thread pool to use virtual threads for Spring Boot 2.7:

*Finished 1000 requests*

*Concurrency Level: 1000  
Time taken for tests: 420.380 seconds  
Complete requests: 1000  
Failed requests: 0*

It took quite a long, about 7 minutes. Why? Spring Boot 2.7 uses Tomcat 9 and it seems that it is not ready to work with virtual threads. The [JEP](https://openjdk.org/jeps/425) says about some cases when virtual threads might not work as expected:

*When it executes code inside a synchronized block or method, or  
When it executes a native method or a*[*foreign function*](https://openjdk.java.net/jeps/424)*.*

Needed changes related to it were done in Tomcat 10 <https://tomcat.apache.org/tomcat-10.1-doc/changelog.html>: “*Refactor synchronization blocks locking on SocketWrapper to use ReentrantLock to support users wishing to experiment with project Loom.“.*And by the way, Spring Boot 3 uses Tomcat 10.

So let’s run our test on Spring Boot 3.0.5 with the same configuration of the Tomcat thread pool:

*Finished 1000 requests*

*Concurrency Level: 1000  
Time taken for tests: 6.288 seconds  
Complete requests: 1000  
Failed requests: 0*

Much faster! Looks like virtual threads work well for Tomcat in Spring Boot 3.

But why do we have such different results? First of all, it’s important to understand that each virtual thread is linked to an OS thread, and it consumes that specific thread only while it performs calculations on the CPU. In our example, if we check logs we can see which threads were used by virtual threads:

*VirtualThread[#62]/runnable@ForkJoinPool-1-worker-4  
VirtualThread[#68]/runnable@ForkJoinPool-1-worker-7  
VirtualThread[#60]/runnable@ForkJoinPool-1-worker-2  
VirtualThread[#65]/runnable@ForkJoinPool-1-worker-6  
VirtualThread[#75]/runnable@ForkJoinPool-1-worker-10  
VirtualThread[#72]/runnable@ForkJoinPool-1-worker-9  
VirtualThread[#77]/runnable@ForkJoinPool-1-worker-11  
VirtualThread[#55]/runnable@ForkJoinPool-1-worker-1  
…*

We can see that they are linked to the common ForkJoinPool thread pool. My computer (which I used for tests) has a 12-core CPU and ForkJoinPool is created by default with a thread pool size equal to the number of cores.  
We can read the confirmation of it in the [JEP](https://openjdk.org/jeps/425):

*The JDK’s virtual thread scheduler is a work-stealing ForkJoinPool that operates in FIFO mode. The parallelism of the scheduler is the number of platform threads available for the purpose of scheduling virtual threads. By default it is equal to the number of available processors, but it can be tuned with the system property jdk.virtualThreadScheduler.parallelism.*

So, in the case of Spring Boot 2.7.10 and not working properly virtual threads as threads of Tomcat, virtual threads took all OS threads (ForkJoinPool) completely for all time of handling HTTP requests, since there were 12 available threads the time was

*84 (1000 / 12) batches \* 5 seconds = 420 seconds*

Which matches to result of the test we saw.

**Conclusion**

We can already start trying virtual threads with Spring Boot 3 on production. The feature is available in preview mode in Java 19 and 20 and hopefully will be available in Java 21 (the next LTS release). I’m excited about this feature, it is a good alternative to the asynchronous style.  
However, you have to take into consideration that not all libraries have already adopted their code to work properly with virtual threads (in most cases it is replacing “synchronize” blocks by ReentrantLock).

P.S.  
Here you can find sources of the repository with the code used in this post:  
<https://github.com/egor-ponomarev/virtualthread-in-spring-boot-example>

To play with different Spring Boot versions, specify the version in the *build.gradle* file (2.7.10, 3.0.5 or any other):

<https://github.com/egor-ponomarev/virtualthread-in-spring-boot-example/blob/main/build.gradle#L3>