➢ How JVM, Tomcat, Java achives multi-threading and concurrency?

➢ First, OS level:

- Lets say in our OS, only 1 core is there, and we have 2 threads that have to execute.
- *T1* contains some heavy IO bound task, and *T2* is a normal task.
- So, lets say CPU execute *T1* [as we have only 1 core, so only 1 thread can run at once]
- Now, *T1* will execute its task that do heavy IO operation. [Core executes Thread, Thread executes Task]
- So, *T1* is blocked now till the IO operation completes, but the core that was running *T1* has no work to do now.
- Now, Core will be assigned to execute *T2*.
- When the IO is completed by *T1*, it becomes ***runnable***.
- Now, depending upon the algorithm OS is chosing to allocate thread to core (priority based, round-robin …etc) *T1* will be executed.
- So, Cores are **limited, hardware** level. Threads are **unlimited, software** level.
- Now you might have question, what is the use of *Threads* where the parallelism is defined by *Core*?
  - There are 2 things: **Concurrency, Parallelism**.
  - Concurrency: many task **in progress**
  - Parallelism : many tasks ***running*** **at same time**.
- In the above case, if you had only one thread that performs both heavy IO task and normal task, then CPU will allocate that thread to core, then it'll go to perform IO bound task. And now the Core is free, idle, have nothing to execute.
  - That normal task will be executed after that heavy IO task is completed.
  - So, **to make sure CPU doesn't sit *idle* during the heavy IO bound task we need to create threads**.

➢ Now, JVM level

- JVM is nothing but a **virtual machine that runs Java byte-code on top of OS**.
- At the JVM / OS level, **all threads are the same**.

  There is no special difference between "Tomcat threads" and "Spring worker threads" at this level; they are all **JVM (OS) threads**.

  The difference exists only at the *application/framework level* (who created them and for what purpose).

- **Tomcat is also just Java code running inside the JVM.**

  It creates and manages its own request thread pool (default maxThreads ≈ 200, configurable).
- These Tomcat threads are used **only to serve HTTP requests.**

  Tomcat assigns **one thread per HTTP request.**
- Once a request thread is assigned, Tomcat *invokes Spring's DispatcherServlet.*

  The Spring application runs on the **same thread that Tomcat allocated for the request.**

  Let's call this thread *T1.*
- Note: At the JVM / OS level, this thread **T1 is just a normal thread.**

  It is simultaneously:
  - a Tomcat request-handling thread, and
  - the thread executing Spring application code.
- Now, suppose this request needs to perform a heavy or blocking task.

  If this heavy task runs on the same thread T1, **it will block the Tomcat request thread until the operation completes.**
- To avoid blocking the request thread, the Spring application may offload the heavy task **to a separate worker thread** (this happens *only if the application is explicitly designed* to do so).
- The worker thread is created or obtained by the application (via JVM + OS).

  This **worker thread is independent of Tomcat's request thread pool.**
- **Case 1: Waiting for the worker thread**
  - If the Spring application waits for the worker thread to finish (e.g., to include its result in the response), then:
    - the request thread T1 remains blocked anyway,
    - and the response is sent only after the heavy task completes.
- **Case 2: Fire-and-forget**
  - If the Spring application does not wait for the heavy task to finish:
    - the task runs in the worker thread,
    - the response is sent immediately,
    - and the request thread T1 is not blocked by the heavy operation.

- Http connection doesn't depends on the thread (in Tomcat) that is handling the request.
- **Normal request & response (synchronous)**
  - **Client → Socket opened → Thread-1 assigned → Filters → DispatcherServlet → Controller → Response created → Response written → Connection closed → Thread-1 released**
  - The **Socket** is the ***TCP*** connection that carries the HTTP request and response.
  - So, when client send a request, this Socket is in opened.
  - Tomcat assigns one thread to handle that request.
  - In this model, this thread will handle everything like
    - **Getting request from client.**
    - **Delegating that to dispatcher servlet.**
    - **Getting response from dispatcher servlet.**
    - **Sending response back to the client.**
  - After this,
    - HTTP response is completed.
    - **Socket** is closed (or returned to keep-alive, depending upon the HTTP version).
  - In this model
    - The **thread lifetime** and **HTTP connection lifetime** are ***almost*** **the same.**
  - The HTTP connection remains open as long as the Socket is open. It **doesn't depend upon the Thread's lifetime**.
- **Streaming request & response (asynchronous)**
  - **Client → Socket opened → Thread-1 assigned → Filters → DispatcherServlet → Controller returns Flux → request.startAsync() → Thread-1 released → HTTP connection kept open → (Flux emits data → Container borrows a thread → Data written to socket → Thread released) → (repeats) → Flux completes / client disconnects → Connection closed**
  - In this case, *up to dispatching the request to DispatcherServlet*, the flow is the same as the synchronous request–response model.
  - If the response is of **streaming** type, then ***request.startAsync()*** is called.

```
@GetMapping(value = "test", produces = MediaType.TEXT_EVENT_STREAM_VALUE)
```

  - The **Servlet container (Tomcat)** creates an ***AsyncContext***, keeps the **Socket** inside it, and releases the request thread.

- When the servlet produces streaming data:
  - The Servlet container (Tomcat) temporarily borrows an available thread
  - Uses that thread to write the data to the Socket
  - Then releases the thread back to the pool
- This process repeats for each streamed chunk until:
  - The stream completes, or
  - The client disconnects
- After the streaming is done,
  - the **Socket** is closed and the AsyncContext is removed/discarded.
  - All references to request, response, socket are released.
- Tomcat does not keep a fixed thread for the connection — it only borrows threads when there is data to write.

- So, in simple terms:
  - **Flux** or **Mono** are not asynchronous by default. They are just *reactive* data types.
  - When Flux is used as a ***streaming HTTP response*** (e.g. **text/event-stream**), Spring switches to **Servlet async mode**, which allows Tomcat to *release the initial request thread* and *borrow threads later* when it needs to write streaming data to the response.
  - This is NOT multithreading in the concurrent sense, because:
    - Multithreading usually means multiple threads executing at the same time.
    - Here, threads are released and reused over time, not running concurrently for the same request.
  - The execution is sequential, even though:
    - Multiple threads may be involved at different times
    - For a given HTTP stream, only one thread writes to the connection at a time.
- My blog link: Medium Blog Link
- When **DispatcherType.ASYNC** is set?
  - When request has been put to **Servlet Async Mode**.

    ```
    request.startAsync();
    ```
  - The original thread has returned (after sending first chunk).
  - The container later re-dispatch the request to continue processing.
  -

- Re-Dispatching of request
  - First request comes from the client; Tomcat receives it and delegates it to Spring.
  - Spring checks whether the response is a streaming type (for example *text/event-stream*).

    If yes, Spring calls *request.startAsync( )*
  - The original Tomcat request thread is released back to the thread pool.
  - Spring subscribes to the **Flux**
  - Whenever a new element (chunk) is emitted by the Flux
    - Spring writes the chunk to the **ServletOutputStream**
    - Tomcat picks any free *http-nio-** thread
    - The chunk is written to the client
    - The thread is released again
  - This process continues for every emitted chunk.
  - After the last chunk:
    - If the Flux *completes* OR
    - An *error* occurs OR
    - Async *timeout* happens OR
    - Client *closes* the connection
  - The async request must now be formally completed, because it was previously put into async mode using *request.startAsync( )*
  - Tomcat performs an ASYNC **re-dispatch**
    - **DispatcherType** is set to **ASYNC**
    - Filters configured for ASYNC are invoked
    - DispatcherServlet resumes async processing
    - Controller method is NOT called again
  - Spring calls **AsyncContext.complete()**
  - Tomcat:
    - Finalizes the response
    - Triggers async listeners (onComplete, onError, onTimeout)
    - Cleans up request/response resources
    - Ends the request lifecycle

- Dispatcher.ASYNC allowance in security filter chain.
  - By default, Spring Security authorization rules are applied mainly to REQUEST dispatcher type (here the context is the dispatcher, not the user).
  - If permitAll() is configured for a route, that route is allowed for all dispatcher types, including ASYNC.
  - If a route is protected:
    - The user is authenticated during the initial REQUEST dispatch
    - Authentication is not performed again during async processing because the SecurityContext already contains the Authentication object for the duration of the request
  - In case of streaming responses, Spring enters async mode and at the end performs an ASYNC re-dispatch
  - This ASYNC re-dispatch targets the same protected route, but by default only REQUEST dispatcher type is permitted, so the ASYNC dispatch may be blocked by authorization rules.
  - Therefore, DispatcherType.ASYNC must be explicitly permitted to allow the async lifecycle to complete successfully

```
.dispatcherTypeMatchers( ...dispatcherTypes: DispatcherType.ASYNC).permitAll()
```

  - In simple term: **DispatcherType.ASYNC** must be explicitly allowed *only for protected routes* using async/streaming, because async re-dispatch is a lifecycle continuation that is otherwise *blocked by default request-only authorization rules*.
-