**How JVM Handles Concurrent Requests in a Spring Boot App: From Threads to Bytecode Execution  
  
  
  
What happens when the spring boot app starts?**

Before that we need to understand how thread gets created in our processes.

So for eg, we have an simple java app that creates three threads and prints something .  
Here what happens is when the app is executed ,process would start with the common fields/methods in heap memory and main thread would have a calls stack,which calls a method that creates three threads ,so when the method is stack gets processed ,three threads will be created for the task(this also explains why main shud be slept) and we can see the printed messages.

This is contrary to the older understanding that we had which is threads would be already present when the process starts..but discard it..only the main threads calls stack matters here after

Now coming to springboot  
  
We jar/war the app,when the executable is executed OS then creates a process

* A process would be created with a main thread that holds the main method in the call stack with other methods ,probably in spring boot it holds controller class methods and such along with GC thread, then if any thread pool that we might use to delegate a task would be created.
* SO HERE , main thread would kind of bootstrap component and other stuff,doesn't really do much
* But in this model, tomcat has pool of threads which would accept the incoming request ,now thread-1 would have a method that calls dispatcherservlet and handleradapter in their threads call stack , and this decides which controller,service and repo methods should be pushed in the thread call stack
* For me, this answer one of the famous doubt that I had, how each request get their own copy of code

If we think closely, for eg the controllers would be designed for not storing any state and it would be in shared heap memory which can be used by any threads(multiple users).

So when multiple users log in, each thread would have their own chain of methods to be executed in their call stack which accepted the incoming request.

Now even though the threads have independent methods in their call stack,it does not answer our question.

To understand this , the basic we should know is each of these shared methods our threads are gonna use would be in bytecode in a different location in heap

Our threads are only gonna point to the reference of that method , but still we could not connect if it refers the same method then obv this arises our lingering doubt how the users access (do they have own copy of code)  
  
Basically ,consider a static method that accepts a string and returns it.  
The bytecode will be like ,  
–accept a param  
–print the param  
  
And our thread’s PCounter knows these reference , when our thread gets it’s scheduled time, these bytecode is interpreted as the thread would have the variable that this method accepts and bytecode→machine code happens there ..and that is offloaded to CPU to further process the actual ask  
  
Since t2,t3 would also get their scheduled time and same happens to them  
And this also explains why only one thread is processed by a processor core(not explain tho)

So mostly if we understand this basic and flow ,we can start designing and understand why thread safety is needed in few logics because multiple threads can call the same static method reference which is in heap,if the method call proves to cause inconsistent data,race conditions(Wats that) and thread error we got to tweak the code mindfully by understanding the above flow

**CATered chatgpt response based on the notes**

* What happens when a **Spring Boot app starts**
* How threads and call stacks work per **incoming request**
* Your lingering doubt: “If all requests share the same method, how are users not clashing?”

Here’s the **clean cheat sheet** with some real-life flavored examples 👇

## **✅ Spring Boot Thread Execution – Cheat Sheet (For Sudharsan's Brain)**

### **🧠 1. What Happens When Spring Boot Starts?**

* You run the app → OS creates a **Java process**
* JVM starts:  
  + main() thread ➝ kicks off SpringApplication.run(...)
  + Bootstraps Beans, Configs, Controllers
  + Starts **embedded Tomcat**, which prepares a **thread pool** (e.g. 200 threads)

### **🔁 2. What Happens Per Request?**

* 🧍 User hits: GET /user/summary

One Tomcat thread (say exec-5) is **assigned** to handle it:  
  
  
exec-5 thread stack:

→ DispatcherServlet.service()

→ Controller.getSummary()

→ Service.getData()

→ Repository.fetchDb()

* All these methods are **shared** in memory (heap), but:  
  + Only **references** are pushed to the thread’s **own stack**
  + Each thread uses its **own method input**

### **🧩 3. Lingering Doubt Solved: “If method is shared, how do users not overwrite?”**

➡️ Example:

@GetMapping("/hello")

public String sayHi(@RequestParam String name) {

return "Hi " + name;

}

3 users hit:

* /hello?name=Ram
* /hello?name=Sam
* /hello?name=Sita

🧵 Each thread (Tomcat's) has this in stack:

sayHi("Ram") ← Thread exec-1

sayHi("Sam") ← Thread exec-2

sayHi("Sita") ← Thread exec-3

📦 The actual method sayHi() is in heap (as bytecode)  
 🧠 JVM doesn’t clone it — just **points to it** 🔁 But each thread has its **own input**, so there's **no clash**

### **💡 Under the Hood (Simplified):**

* Each thread has:  
  + 🧱 A call stack (with parameters, local variables)
  + 🧭 A **program counter (PC)** that points to **which bytecode instruction** to run next
* When the thread is scheduled by CPU:  
  + That PC-guided instruction is turned into **machine code**
  + Processed by **one core**
  + Other threads wait (context switch handles fairness)

### **🧪 Bonus Use Case:**

Suppose this controller:

@GetMapping("/calc")

public int square(@RequestParam int x) {

return x \* x;

}

* 100 users hit /calc?x=2, /calc?x=3… etc
* All share the **same method** in heap

But each request thread runs:  
  
  
square(2)

square(3)

square(4)

...

✔️ Works flawlessly — no conflict, because each thread uses its **own param**

### **🧠 Scenario:**

You have a Spring Boot controller method like:

@GetMapping("/user/summary")

public UserSummary getSummary(@RequestParam String username) {

String name = fetchNameFromDB(username); // DB call 1

String address = fetchAddressFromDB(username); // DB call 2

String balance = fetchBalanceFromDB(username); // DB call 3

return new UserSummary(name, address, balance);

}

Now imagine **3 users** hit this API at the same time. Let's focus on 2 of them: **T1 (UserA)** and **T2 (UserB)**.

### **⚙️ What Happens Under the Hood (Step-by-step):**

#### **✅ 1. Request Accepted:**

* Tomcat thread http-nio-8080-exec-1 handles UserA’s request (T1).
* Tomcat thread http-nio-8080-exec-2 handles UserB’s request (T2).
* These threads are **mapped to JVM-level threads**, and JVM gives each thread its **own call stack**.

#### **✅ 2. Same Method, Different Data:**

* Both threads **point to the same controller method bytecode** in JVM memory.
* But they pass different values: username="UserA" and username="UserB".

#### **✅ 3. Processor Scheduling:**

* CPU core #1 picks T1
* CPU core #2 picks T2  
   (Assuming a multi-core processor)

Now:

* Both threads **refer to the same method**, but with their own **input values**.
* **JVM interprets or executes JIT-compiled code independently** for each thread.
* The execution is **truly parallel**, thanks to separate CPU cores.

#### **✅ 4. Execution Path (for each thread):**

Each thread has:

* Its **own Program Counter (PC)** → pointing to current instruction (e.g., call fetchName()).
* Its **own local variables** (like username, name, etc.).
* Its **own stack of method calls**.

**🔄 Summary Points for Notes:**

* ✅ JVM creates a **new thread** per incoming request (handled by Tomcat).
* ✅ All threads **share the same code (bytecode)**, but use **different local data**.
* ✅ Threads run **in parallel** if CPU allows — each processed by **separate cores**.
* ✅ JVM either **interprets bytecode or runs JIT’d machine code**.
* ✅ Each thread's **Program Counter (PC)** tracks where it is in the method.
* ✅ This is **how 2 users can run the same method at the same time safely**, provided there’s **no shared mutable state**.

**Now lets have some examples on how to design thread safe apps**

We had one more doubt ,where even we have a method that sets a value for common field and use that to fetch or do some action (like calling a db,updating a map with the value that set in common field even map being static and thread safe)  
  
Since each thread gets their time and field set should be only for that particular processing of the thread ri8..but here we have a concept called interleaving  
  
  
**⚠️ Shared State Doesn’t Care About Threads or CPUs**

Let’s say you have this in your service:

@Service

public class UserService {

private String userFromRequest; // ❌ shared mutable state

public UserSummary getSummary(String username) {

this.userFromRequest = username;

String name = fetchNameFromDB(); // <-- Reads the shared variable

String address = fetchAddressFromDB();

String balance = fetchBalanceFromDB();

return new UserSummary(name, address, balance);

}

private String fetchNameFromDB() {

return "Name for: " + userFromRequest;

}

}

### **🔄 What Actually Happens:**

Now imagine this **interleaving** due to CPU scheduling:

#### **🧵 Thread T1 (User A)**

1. Sets userFromRequest = "A"
2. **(Before it calls fetchNameFromDB(), it gets paused)**

#### **🧵 Thread T2 (User B)**

1. Sets userFromRequest = "B" ← **overwrites the same field**
2. Fetches name → "Name for: B"
3. Completes.

#### **🧵 Thread T1 resumes:**

1. Fetches name → "Name for: B" ❌
2. Even though it **started** with "A", it now reads "B" because it's reading the same shared field.

### **❗Why This Is a Problem**

* Threads **don’t get a "copy" of the field** — it's one field in heap memory.
* Even if two threads **run on two cores** and get CPU time **simultaneously**, they still **access the same variable in memory**.
* Java Memory Model doesn’t guarantee **atomic visibility** or **consistency** unless you synchronize or isolate the state.

### **✅ Real Fix**

Keep request state **inside method arguments** or use **ThreadLocal** (carefully).

public UserSummary getSummary(String username) {

// username is now safely passed around, not stored in a shared field

}

### **🧠 In Short:**

Even though:

* Threads are separate
* CPU cores are separate
* They have their own call stacks

**They still touch the same field in memory** if it's not isolated, and **CPU scheduling/interleaving makes timing unpredictable**.