# Usage of multithreading with Spring Boot

[**Usage of multithreading with Spring Boot 1**](#_33jmxr6cite2)

[**Usage of multithreading with Spring Boot 2**](#_1xyvbtnb15g6)

[**Key best practices for MultiThreading with Spring Boot 2**](#_391myvu1aoge)

[1. Use Spring's Thread Management Tools 2](#_ud2vfif8v44i)

[2. Configure an Appropriate Thread Pool 2](#_4vz3okyvdw0x)

[3. Handle Exceptions in Asynchronous Tasks 3](#_nq0cmsshsyhu)

[4. Limit Concurrent Access to Shared Resources 3](#_ctyztjaddrmb)

[5. Use Specific Tools for Complex Tasks 4](#_yckzazusk77j)

[6. Avoid Blocking Threads 4](#_cjiix0hkzr9)

[7. Monitor and Adjust Performance 4](#_85bd9qg2zql3)

[8. Testing and Validation 4](#_8kfrlwd71p4c)

[**A focus on things to avoid in MultiThreading with Spring boot : 4**](#_qcgvsc3rw6xl)

[1. Avoid unsynchronized concurrent access to shared resources 4](#_kl1gu2bjsrgy)

[2. Don’t use a poorly configured thread pool 5](#_8wfzh7vfjn7w)

[3. Avoid blocking threads 5](#_1jmzqirtcmb1)

[4. Don’t ignore exceptions in asynchronous threads 5](#_7loiz33avxth)

[5. Avoid manually creating threads 5](#_jpth1xyg7b66)

[6. Don’t overlook transactional context 5](#_l0t8auuuy5jq)

[7. Don’t ignore the global performance impact 6](#_d80ls667sbsb)

[8. Avoid using non-thread-safe classes 6](#_b47feupoqoee)

[9. Don’t neglect monitoring active threads 6](#_13u7ktsqc3l)

[**Key Points to Remember: 6**](#_iygd1xzbnbpg)

[Example / Practical case / Results 7](#_vgxe912keinf)

[Contexte 7](#_jwg1om9anomh)

[- Temps d’exécution 8](#_16qw46wjlwi0)

[- Exécution du processus : 1 Thread 9](#_au90fuc9sxzf)

[- Temps d’exécution 9](#_7tbmimcnzvqu)

[- Exécution du processus : 3 Threads 10](#_z4z3pg4yhdg1)

[USER 10](#_xefy8iyj9x03)

[Async Config 10](#_zay8crz4w7s9)

[UserRepository 11](#_r3fthu7z82xb)

[UserService 11](#_jr08qbqy4cf)

[UserController 13](#_8y8oy4qiyxaw)

# Usage of multithreading with Spring Boot

Using multithreading with Spring Boot can enhance application performance by processing parallel or time-consuming tasks. However, improper thread management may lead to synchronization, concurrency, or memory issues.

We will start by listing a range of best practices and end with a more concrete example in a practical case.

## Key best practices for MultiThreading with Spring Boot

## 

### **1. Use Spring's Thread Management Tools**

* **@Async** : Annotate methods to execute them asynchronously. It relies on a configurable thread pool.

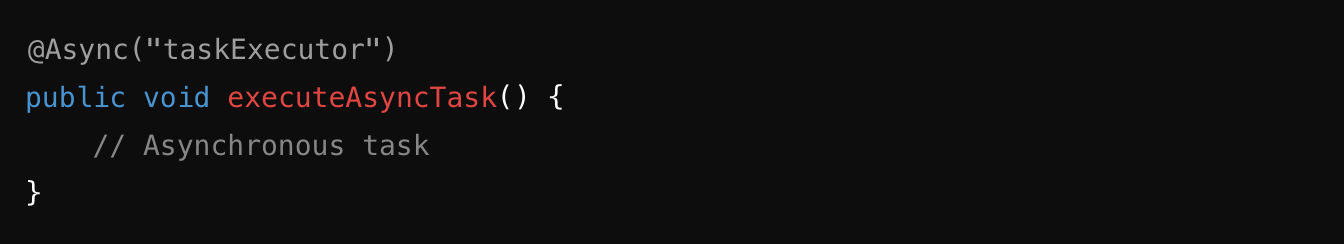


Enable asynchronous processing in your configuration:

### **2. Configure an Appropriate Thread Pool**

* Define a thread pool to control the number of active threads and prevent overloading.

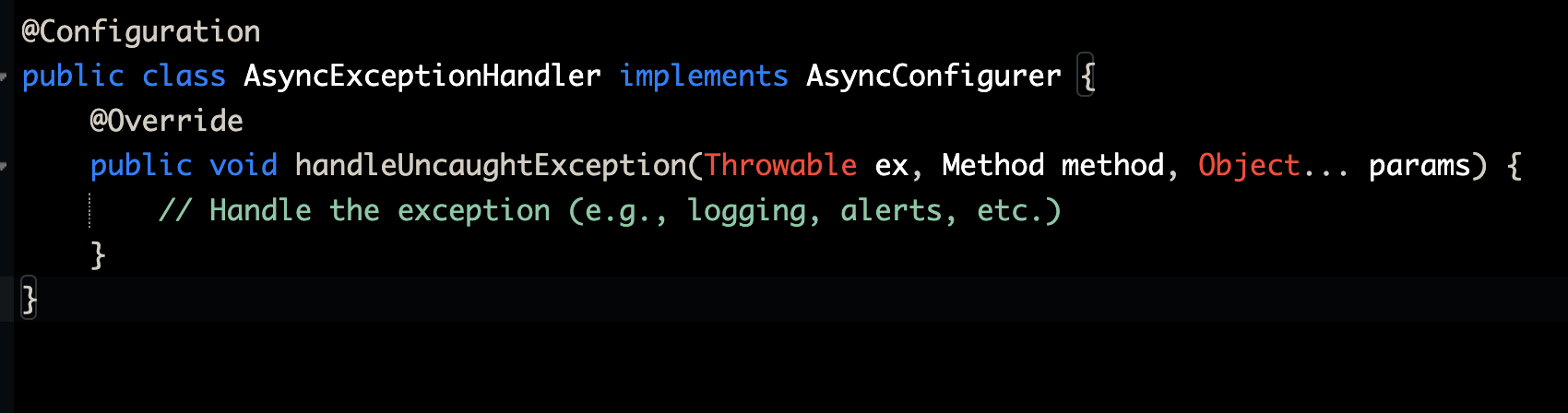


Specify this thread pool in @Async if necessary:

### **3. Handle Exceptions in Asynchronous Tasks**

Exceptions in asynchronous methods are not automatically propagated. Use a custom handler:

* Implement AsyncUncaughtExceptionHandler:



### **4. Limit Concurrent Access to Shared Resources**

* Use synchronization mechanisms like **synchronized**, **ReentrantLock**, or thread-safe collections (e.g., ConcurrentHashMap) to avoid concurrency issues.
* Minimize the use of shared variables between threads whenever possible.

### **5. Use Specific Tools for Complex Tasks**

* For advanced scenarios like parallel execution or task coordination, consider using:
  + **CompletableFuture** to compose asynchronous tasks.
  + **ExecutorService** for fine-grained thread management.

### **6. Avoid Blocking Threads**

* Avoid blocking operations (e.g., synchronous network calls) in non-main threads. Instead, adopt reactive approaches like Spring WebFlux where feasible.

### **7. Monitor and Adjust Performance**

* Use tools like **Postman**, **Spring Actuator**,
* Adjust thread pool parameters based on workload requirements.

In our example, we will see the time displayed by Postman. This timing solution is also possible with Loggers by calculating the total time taken by a particular request.

### **8. Testing and Validation**

* Perform load testing to ensure that multithreading actually improves performance.
* Test concurrency scenarios to identify race conditions or potential deadlocks.

## A focus on things to avoid in MultiThreading with Spring boot :

Although multithreading in a Spring Boot application can improve performance and responsiveness, but mishandling it can lead to issues that are difficult to debug. Here's what to avoid doing:

### **1. Avoid unsynchronized concurrent access to shared resources**

* If multiple threads access and modify shared objects (like static variables or collections), it can lead to **race conditions**.
* **Solution**: Use synchronization mechanisms or thread-safe data structures like ConcurrentHashMap.

### **2. Don’t use a poorly configured thread pool**

* Spring Boot uses a thread pool by default via the task manager (TaskExecutor). A poorly configured pool (e.g., too few or too many threads) can cause performance issues:
  + A small pool slows down execution.
  + A large pool overloads resources.
* **Solution**: Properly configure the thread pool via @EnableAsync and define its properties in the configuration file (application.properties or application.yml).

### **3. Avoid blocking threads**

* Blocking threads with blocking calls (e.g., Thread.sleep, synchronous I/O operations) limits performance.
* **Solution**: Use non-blocking APIs like **Project Reactor** or **CompletableFuture** for asynchronous operations.

### **4. Don’t ignore exceptions in asynchronous threads**

* Exceptions in tasks executed by threads are often lost if not handled.
* **Solution**: Use specific exception handlers to capture errors (AsyncUncaughtExceptionHandler).

### **5. Avoid manually creating threads**

* Creating threads with new Thread() or a custom ExecutorService can lead to resource leaks or improper scaling.
* **Solution**: Use Spring abstractions like **@Async** and **TaskExecutor**.

### **6. Don’t overlook transactional context**

* Threads executing asynchronous tasks do not share the same transactional context as the main thread.
* **Solution**: Explicitly pass the necessary data or use independent transactions if needed.

### **7. Don’t ignore the global performance impact**

* Too many threads can cause contention issues, CPU overload, or memory saturation.
* **Solution**: Perform load testing and adjust configuration according to real-world demand.

### **8. Avoid using non-thread-safe classes**

* Some classes like SimpleDateFormat or ArrayList are not thread-safe.
* **Solution**: Use thread-safe alternatives like DateTimeFormatter or synchronized collections.

### **9. Don’t neglect monitoring active threads**

* A lack of thread supervision can lead to issues like thread leaks or deadlocks.
* **Solution**: Use monitoring tools like **Spring Actuator**.

## Key Points to Remember:

* **Dependency Injection**: Ensure that @Async methods are called from another bean and not from the same bean instance.
* **Executor Tuning**: Properly configure the thread pool size for optimal performance.
* **Thread-Safe Beans**: Ensure your beans are thread-safe when accessed by multiple threads concurrently.

## **Example / Practical case / Results**

### Contexte

Sur une architecture microservice avec Spring Boot, disposant d’un dataset disponible en base de données, réalisons deux appels d'API @GetMapping qui récupèrent 2000 utilisateurs (User) .

* Premier Appel (hors multithread)

@GetMapping(value = "/users", produces = {MediaType.*APPLICATION\_JSON\_VALUE*})

public CompletableFuture<ResponseEntity> findAllUsers () {

return userService.findAllUsers().thenApply(ResponseEntity::*ok*);

}

* Deuxième Appel (contexte multithread)

@GetMapping(value = "/getUsersByMultiThreads", produces = {MediaType.*APPLICATION\_JSON\_VALUE*})

public ResponseEntity findAllUsersMultiThreads () {

CompletableFuture<List<User>> usersFuture1 = userService.findAllUsers();

CompletableFuture<List<User>> usersFuture2 = userService.findAllUsers();

CompletableFuture<List<User>> usersFuture3 = userService.findAllUsers();

CompletableFuture.*allOf*(usersFuture1, usersFuture2, usersFuture3).join();

return ResponseEntity.*status*(HttpStatus.*OK*).build();

}

Constat :

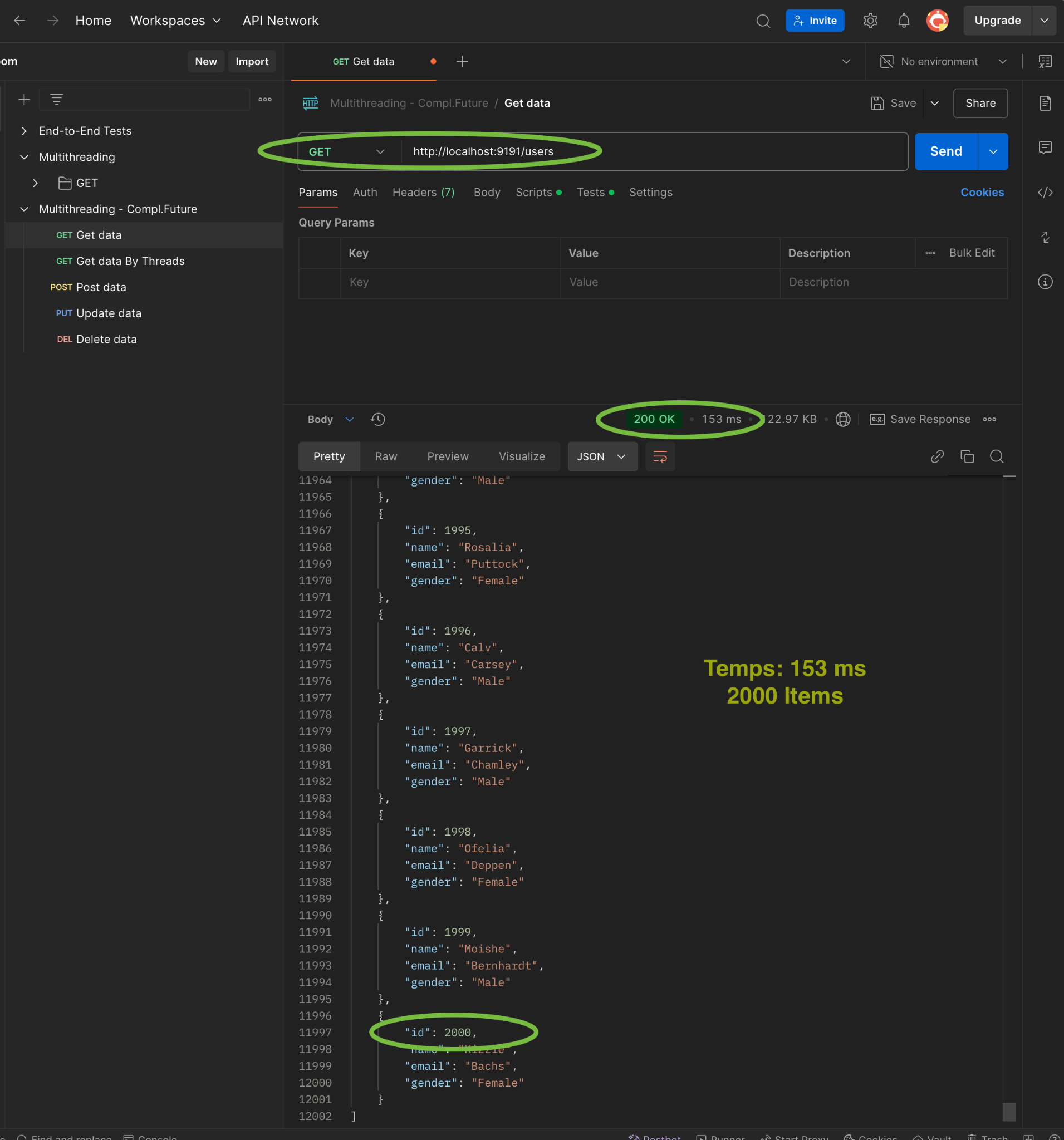
* Bien que les temps obtenus dépendent fortement de la configuration du matériel utilisé.
* Avec une configuration qui autorise l’’utilisation de jusqu'à 3 threads en traitement simultané.

threadPoolTaskExecutor.setCorePoolSize(3);

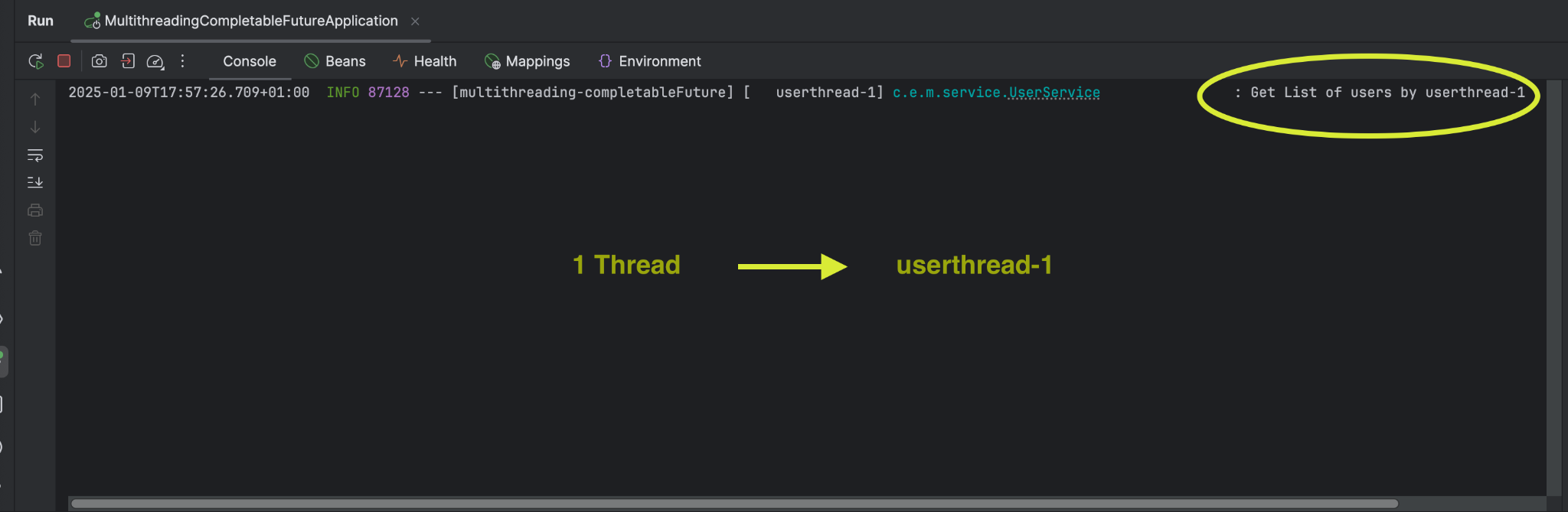
threadPoolTaskExecutor.setMaxPoolSize(3);

* Je constate un traitement **4,63 fois plus rapide** en faveur du multithreading.

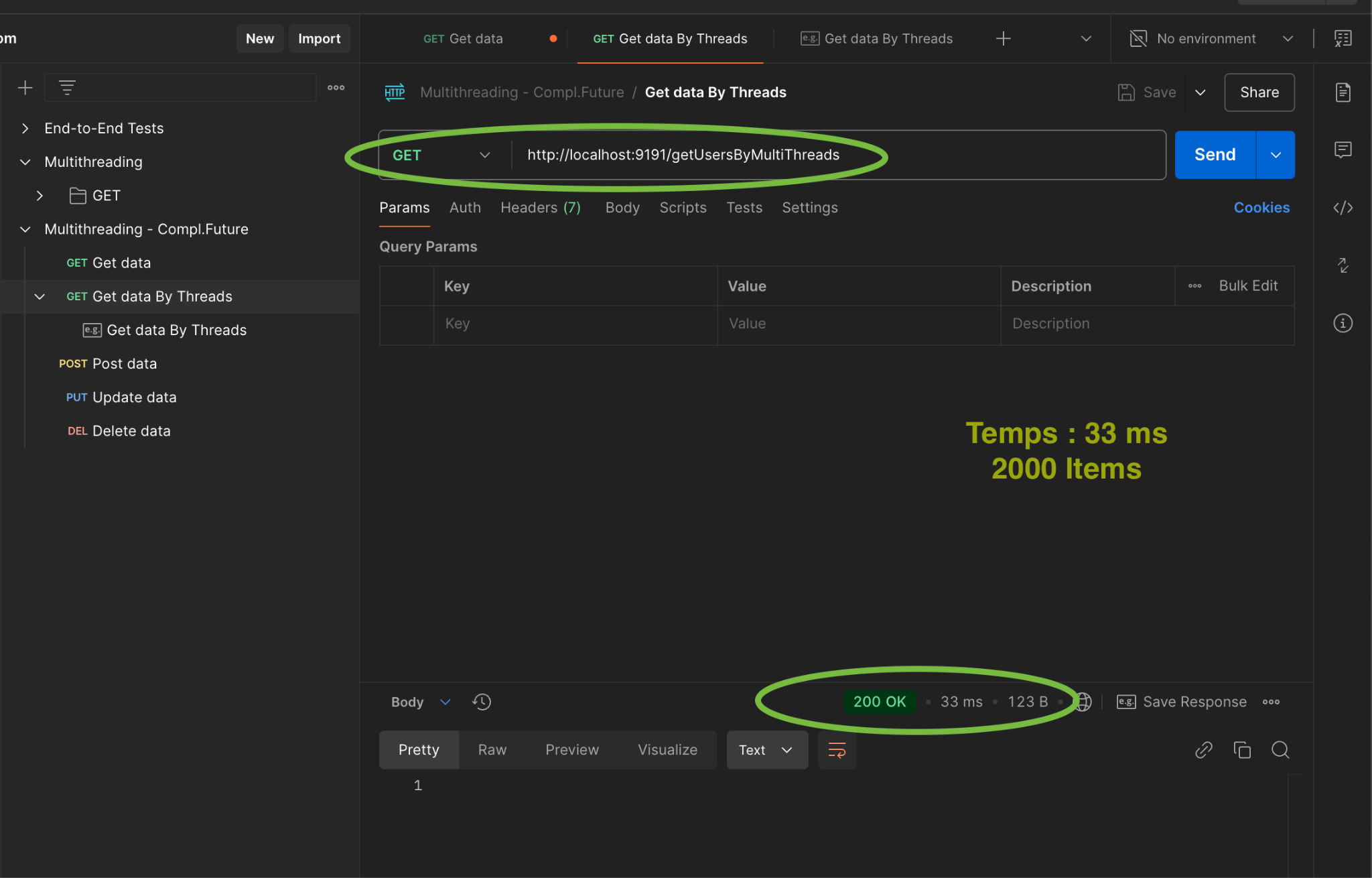
#### Temps d’exécution

* 

#### Exécution du processus : 1 Thread



#### Temps d’exécution



#### Exécution du processus : 3 Threads

### 

### USER

package com.example.multithreading\_completableFuture.entity;

import jakarta.persistence.Entity;

import jakarta.persistence.GeneratedValue;

import jakarta.persistence.Id;

import jakarta.persistence.Table;

import lombok.AllArgsConstructor;

import lombok.Data;

import lombok.NoArgsConstructor;

@Data

@Entity

@Table(name = "USER\_TBL")

@AllArgsConstructor

@NoArgsConstructor

public class User {

@Id

@GeneratedValue

private Integer id;

private String name;

private String email;

private String gender;

}

### AsyncConfig

package com.example.multithreading\_completableFuture.config;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.scheduling.annotation.EnableAsync;

import org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor;

import java.util.concurrent.Executor;

@Configuration

@EnableAsync

public class AsyncConfig {

@Bean(name = "taskExecutor")

public Executor taskExecutor() {

ThreadPoolTaskExecutor threadPoolTaskExecutor = new ThreadPoolTaskExecutor();

threadPoolTaskExecutor.setCorePoolSize(3);

threadPoolTaskExecutor.setMaxPoolSize(3);

threadPoolTaskExecutor.setQueueCapacity(100);

threadPoolTaskExecutor.setThreadNamePrefix("userthread-");

threadPoolTaskExecutor.initialize();

return threadPoolTaskExecutor;

}

}

### UserRepository

package com.example.multithreading\_completableFuture.repository;

import com.example.multithreading\_completableFuture.entity.User;

import org.springframework.data.jpa.repository.JpaRepository;

public interface UserRepository extends JpaRepository<User, Integer> {

}

### UserService

package com.example.multithreading\_completableFuture.service;

import com.example.multithreading\_completableFuture.entity.User;

import com.example.multithreading\_completableFuture.repository.UserRepository;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.scheduling.annotation.Async;

import org.springframework.stereotype.Service;

import org.springframework.web.multipart.MultipartFile;

import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStreamReader;

import java.util.ArrayList;

import java.util.List;

import java.util.concurrent.CompletableFuture;

@Service

public class UserService {

private UserRepository userRepository;

public UserService(UserRepository userRepository) {

this.userRepository = userRepository;

}

Object target;

Logger logger = LoggerFactory.*getLogger*(UserService.class);

@Async

public CompletableFuture<List<User>> saveUser(MultipartFile file) throws Exception {

long startTime = System.*currentTimeMillis*();

List<User> users = parceCSVFile(file);

//logger.info("Saving list of users with size = {}", users.size(), " "+ Thread.currentThread().getName());

users = userRepository.saveAll(users);

long endTime = System.*currentTimeMillis*();

//logger.info("Total time {}", (endTime - startTime));

logger.info("Saving list of users with size = {}", users.size() +" "+ Thread.*currentThread*().getName() + " -Total time " + (endTime - startTime));

return CompletableFuture.*completedFuture*(users);

}

@Async

public CompletableFuture findAllUsers() {

long startTime = System.*currentTimeMillis*();

List<User> users = userRepository.findAll();

long endTime = System.*currentTimeMillis*();

logger.info("Get List of users by " + Thread.*currentThread*().getName() + " -Total time: {} ms", (endTime - startTime));

return CompletableFuture.*completedFuture*(users);

}

public List<User> parceCSVFile(MultipartFile file) throws Exception {

final List<User> users = new ArrayList<>();

try (BufferedReader br = new BufferedReader(new InputStreamReader(file.getInputStream()))) {

String line;

while ((line = br.readLine()) != null) {

final String[] data = line.split(",");

final User user = new User();

user.setName(data[0]);

user.setEmail(data[1]);

user.setGender(data[2]);

users.add(user);

}

return users;

}catch (final IOException e) {

logger.error("Failled to parse CSV file {} ", file.getOriginalFilename());

throw new Exception("Failled to parse CSV file {} ", e);

}

}

}

### UserController

package com.example.multithreading\_completableFuture.controller;

import com.example.multithreading\_completableFuture.entity.User;

import com.example.multithreading\_completableFuture.service.UserService;

import org.springframework.http.HttpStatus;

import org.springframework.http.MediaType;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.PostMapping;

import org.springframework.web.bind.annotation.RequestParam;

import org.springframework.web.bind.annotation.RestController;

import org.springframework.web.multipart.MultipartFile;

import java.util.List;

import java.util.concurrent.CompletableFuture;

@RestController

public class UserController {

private UserService userService;

public UserController(UserService userService) {

this.userService = userService;

}

@PostMapping(value = "/users", consumes = {MediaType.*MULTIPART\_FORM\_DATA\_VALUE*}, produces = {MediaType.*APPLICATION\_JSON\_VALUE*})

public ResponseEntity saveUser(@RequestParam (value = "files") MultipartFile[] files) throws Exception {

for (MultipartFile file:files){

userService.saveUser(file);

}

return ResponseEntity.*status*(HttpStatus.*CREATED*).build();

}

@GetMapping(value = "/users", produces = {MediaType.*APPLICATION\_JSON\_VALUE*})

public CompletableFuture<ResponseEntity> findAllUsers () {

return userService.findAllUsers().thenApply(ResponseEntity::*ok*);

}

@GetMapping(value = "/getUsersByMultiThreads", produces = {MediaType.*APPLICATION\_JSON\_VALUE*})

public ResponseEntity findAllUsersMultiThreads () {

CompletableFuture<List<User>> usersFuture1 = userService.findAllUsers();

CompletableFuture<List<User>> usersFuture2 = userService.findAllUsers();

CompletableFuture<List<User>> usersFuture3 = userService.findAllUsers();

CompletableFuture.*allOf*(usersFuture1, usersFuture2, usersFuture3).join();

return ResponseEntity.*status*(HttpStatus.*OK*).build();

}

}