Controller and Service Logic using Multithreading in Java

To write controller and service logic using multithreading in a Java-based web application, we typically integrate multithreading into the service layer to handle long-running or asynchronous tasks efficiently. By using tools like CompletableFuture for asynchronous tasks or leveraging the executor service for more advanced multithreading management, we can make the service layer handle multiple tasks in parallel.

# Steps:

1. Controller: Handle HTTP requests and delegate tasks to the service layer.  
2. Service: Perform tasks asynchronously using CompletableFuture and multithreading techniques.  
3. Multithreading in Service Layer: You can use CompletableFuture or an ExecutorService for parallelism.

# Example of Multithreaded Service Logic Using CompletableFuture

In this example, the service method simulates fetching data from multiple sources and performs the tasks in parallel.

import org.springframework.stereotype.Service;  
import java.util.concurrent.CompletableFuture;  
import java.util.concurrent.TimeUnit;  
  
@Service  
public class MyService {  
  
 // Simulate an async task that takes time to complete  
 public CompletableFuture<String> asyncTask1() {  
 return CompletableFuture.supplyAsync(() -> {  
 try {  
 System.out.println("Executing Task 1...");  
 TimeUnit.SECONDS.sleep(2); // Simulate a delay of 2 seconds  
 } catch (InterruptedException e) {  
 throw new IllegalStateException(e);  
 }  
 return "Result from Task 1";  
 });  
 }  
  
 // Simulate another async task  
 public CompletableFuture<String> asyncTask2() {  
 return CompletableFuture.supplyAsync(() -> {  
 try {  
 System.out.println("Executing Task 2...");  
 TimeUnit.SECONDS.sleep(3); // Simulate a delay of 3 seconds  
 } catch (InterruptedException e) {  
 throw new IllegalStateException(e);  
 }  
 return "Result from Task 2";  
 });  
 }  
  
 // Combine the results of multiple tasks  
 public CompletableFuture<String> executeTasksInParallel() {  
 CompletableFuture<String> task1 = asyncTask1();  
 CompletableFuture<String> task2 = asyncTask2();  
  
 return task1.thenCombine(task2, (result1, result2) -> {  
 System.out.println("Combining results from Task 1 and Task 2...");  
 return result1 + " | " + result2;  
 });  
 }  
}

## Create the Controller to Handle HTTP Requests

import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.web.bind.annotation.GetMapping;  
import org.springframework.web.bind.annotation.RestController;  
  
import java.util.concurrent.CompletableFuture;  
  
@RestController  
public class MyController {  
  
 @Autowired  
 private MyService myService;  
  
 @GetMapping("/execute-tasks")  
 public CompletableFuture<String> executeTasks() {  
 // Call the service layer to execute tasks asynchronously  
 return myService.executeTasksInParallel();  
 }  
}

## Enable Async Processing in Spring Boot

import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.scheduling.annotation.EnableAsync;  
  
@SpringBootApplication  
@EnableAsync // Enable async processing in Spring Boot  
public class AsyncApplication {  
  
 public static void main(String[] args) {  
 SpringApplication.run(AsyncApplication.class, args);  
 }  
}

## Testing the Application

1. Run the Spring Boot application.  
2. Use a browser or a tool like Postman to send a GET request to http://localhost:8080/execute-tasks.  
3. The response will be sent once both tasks (asyncTask1() and asyncTask2()) have completed, and their results are combined in parallel.

## How it Works

1. asyncTask1() and asyncTask2() simulate tasks that take time to complete (2 and 3 seconds, respectively).  
2. executeTasksInParallel() triggers both tasks in parallel using CompletableFuture.  
3. thenCombine() waits for both tasks to complete and then combines their results.  
4. The controller returns the combined result of both tasks.

# Using a Custom ExecutorService for Better Thread Management

By default, CompletableFuture uses the ForkJoinPool, but you can provide a custom Executor to control the number of threads used for task execution. You can define a thread pool as follows:

import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
import java.util.concurrent.Executor;  
import java.util.concurrent.Executors;  
  
@Configuration  
public class AsyncConfig {  
  
 @Bean(name = "customExecutor")  
 public Executor customExecutor() {  
 return Executors.newFixedThreadPool(5); // Creates a thread pool with 5 threads  
 }  
}

Now, you can pass this custom executor to CompletableFuture:

import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Service;  
import java.util.concurrent.CompletableFuture;  
import java.util.concurrent.Executor;  
  
@Service  
public class MyService {  
  
 @Autowired  
 private Executor customExecutor;  
  
 public CompletableFuture<String> asyncTask1() {  
 return CompletableFuture.supplyAsync(() -> {  
 try {  
 System.out.println("Task 1 is running on thread: " + Thread.currentThread().getName());  
 Thread.sleep(2000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 return "Task 1 result";  
 }, customExecutor);  
 }  
  
 public CompletableFuture<String> asyncTask2() {  
 return CompletableFuture.supplyAsync(() -> {  
 try {  
 System.out.println("Task 2 is running on thread: " + Thread.currentThread().getName());  
 Thread.sleep(3000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 return "Task 2 result";  
 }, customExecutor);  
 }  
}

# Summary

- The controller delegates task execution to the service layer, which handles business logic asynchronously using CompletableFuture.  
- You can run multiple tasks in parallel using CompletableFuture.supplyAsync() and combine the results using methods like thenCombine().  
- By using custom executors, you can control how many threads handle asynchronous tasks, improving performance and resource management.  
 ==============

CompletableFuture in Java 8 is a powerful tool for writing asynchronous code. With it, you can:

* Execute tasks asynchronously using supplyAsync() and runAsync().
* Chain tasks using thenApply(), thenAccept(), and thenRun().
* Combine tasks using thenCombine() or allOf().
* Handle errors gracefully using exceptionally() and handle().
*  **CompletableFuture.runAsync()**: Runs an asynchronous task that doesn't return any result.
*  **CompletableFuture.supplyAsync()**: Runs an asynchronous task that returns a result.
*  **thenApply()**: Applies a function to the result of the previous task.
*  **thenAccept()**: Consumes the result of the previous task without returning a new result.
*  **thenRun()**: Runs a task after the previous one finishes but doesn't take or return a result.
*  **thenCombine()**: Combines the results of two CompletableFutures.
*  **thenCompose()**: Flattens nested CompletableFutures.
*  **allOf()**: Runs multiple tasks in parallel and waits for all of them to complete.
* **exceptionally()**: Handles the exception and provides a fallback value.
* import java.util.concurrent.CompletableFuture;
* public class CompletableFutureExceptionHandling {
* public static void main(String[] args) {
* CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> {
* System.out.println("Running a task...");
* if (true) {
* throw new RuntimeException("Something went wrong!");
* }
* return "Success";
* });
* // Handle the exception and provide a fallback
* future = future.exceptionally(ex -> {
* System.out.println("Error occurred: " + ex.getMessage());
* return "Fallback result";
* });
* // Print the result
* future.thenAccept(result -> System.out.println("Result: " + result));
* // Sleep the main thread to allow async tasks to complete (for demo purposes)
* try { Thread.sleep(1000); } catch (InterruptedException e) { e.printStackTrace(); }
* }
* }

import java.util.concurrent.CompletableFuture;

public class CompletableFutureCombine {

public static void main(String[] args) {

CompletableFuture<String> future1 = CompletableFuture.supplyAsync(() -> {

System.out.println("Task 1: Fetching data...");

return "Data from Task 1";

});

CompletableFuture<String> future2 = CompletableFuture.supplyAsync(() -> {

System.out.println("Task 2: Fetching more data...");

return "Data from Task 2";

});

// Combine the results of future1 and future2

CompletableFuture<String> combinedFuture = future1.thenCombine(future2, (data1, data2) -> {

return data1 + " + " + data2;

});

// Block and get the combined result

try {

String result = combinedFuture.get(); // Blocking call

System.out.println("Combined result: " + result);

} catch (Exception e) {

e.printStackTrace();

}

}

}

import java.util.concurrent.CompletableFuture;

public class CompletableFutureChaining {

public static void main(String[] args) {

CompletableFuture.supplyAsync(() -> {

System.out.println("Task 1: Fetching data...");

return "Data from Task 1";

}).thenApply(data -> {

System.out.println("Task 2: Processing " + data);

return data + " + Processed by Task 2";

}).thenAccept(result -> {

System.out.println("Task 3: Result is " + result);

});

// Sleep the main thread to allow async tasks to complete (for demo purposes)

try { Thread.sleep(2000); } catch (InterruptedException e) { e.printStackTrace(); }

}

}Synchronization in Java

Synchronization in Java is the capability to control the access of multiple threads to any shared resource.

Java Synchronization is better option where we want to allow only one thread to access the shared resource.

1. Synchronized method.
2. Synchronized block.
3. Static synchronization.