earn to create **asynchronous controller** methods in Spring framework with the help of [@Async](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/scheduling/annotation/Async.html) and [@EnableAsync](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/scheduling/annotation/EnableAsync.html) annotations, [async thread pool](https://howtodoinjava.com/java/multi-threading/java-thread-pool-executor-example/" \t "_blank) on top of Java [ExecutorService](https://howtodoinjava.com/java/multi-threading/executor-framework-tutorial/" \t "_blank) framework.

**1. Spring @Async rest controller**

Spring comes with @EnableAsync annotation and can be applied on application classes for asynchronous behavior. This annotation will look for methods marked with @Async annotation and run in background thread pools. The @Async annotated methods can return CompletableFuture to hold the result of an asynchronous computation.

To **enable async configuration in spring**, follow these steps:

1. **Create async thread pool**

|  |
| --- |
| AsyncConfiguration.java |
| @Configuration  @EnableAsync  public class AsynchConfiguration  {      @Bean(name = "asyncExecutor")      public Executor asyncExecutor()      {          ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();          executor.setCorePoolSize(3);          executor.setMaxPoolSize(3);          executor.setQueueCapacity(100);          executor.setThreadNamePrefix("AsynchThread-");          executor.initialize();          return executor;      }  } |

1. **@Async controller methods**

Methods which shall run asynchronously, annotate them with @Async annotation and method return type should return

|  |
| --- |
| @Async("asyncExecutor")  public CompletableFuture<EmployeeNames> methodOne() throws InterruptedException {      //code  } |

1. **Combine async method results**

|  |
| --- |
| Inside REST Controller |
| CompletableFuture.allOf(methodOne, methodTwo, methodThree).join(); |

**2. Spring @Async rest controller example**

In this demo, we will create an REST API which will fetch data from three (3) remote services asynchronously and when responses from all 3 services is available then aggregate the responses. e.g.

1. Invoke EmployeeName API
2. Invoke EmployeeAddress API
3. Invoke EmployeePhone API
4. Wait for responses from above services
5. Aggregate all three API responses and build final response to send back to client

**2.1. EmployeeName, EmployeeAddress and EmployeePhone APIs to be accessed async way**

|  |
| --- |
| EmployeeDataController.java |
| package com.howtodoinjava.example.sampleservice.controller;    import java.util.ArrayList;  import java.util.List;    import org.slf4j.Logger;  import org.slf4j.LoggerFactory;  import org.springframework.web.bind.annotation.RequestMapping;  import org.springframework.web.bind.annotation.RequestMethod;  import org.springframework.web.bind.annotation.RestController;    import com.howtodoinjava.example.sampleservice.model.EmployeeAddress;  import com.howtodoinjava.example.sampleservice.model.EmployeeAddresses;  import com.howtodoinjava.example.sampleservice.model.EmployeeName;  import com.howtodoinjava.example.sampleservice.model.EmployeeNames;  import com.howtodoinjava.example.sampleservice.model.EmployeePhone;    @RestController  public class EmployeeDataController  {      private static Logger log = LoggerFactory.getLogger(EmployeeDataController.class);        @RequestMapping(value = "/addresses", method = RequestMethod.GET)      public EmployeeAddresses getAddresses()      {          log.info("get addresses Start");            EmployeeAddresses employeeAddressesList = new EmployeeAddresses();            EmployeeAddress employeeAddress1 = new EmployeeAddress();          EmployeeAddress employeeAddress2 = new EmployeeAddress();            List<EmployeeAddress> addressList = new ArrayList<EmployeeAddress>();            {              employeeAddress1.setHouseNo("1111");              employeeAddress1.setStreetNo("111");              employeeAddress1.setZipCode("111111");                employeeAddress2.setHouseNo("222");              employeeAddress2.setStreetNo("222");              employeeAddress2.setZipCode("222222");                addressList.add(employeeAddress1);              addressList.add(employeeAddress2);                employeeAddressesList.setEmployeeAddressList(addressList);          }            return employeeAddressesList;      }        @RequestMapping(value = "/phones", method = RequestMethod.GET)      public EmployeePhone getPhoneNumbers()      {          log.info("get phones Start");            EmployeePhone employeePhone = new EmployeePhone();          {              ArrayList<String> phoneNumberList = new ArrayList<String>();                phoneNumberList.add("100000");              phoneNumberList.add("200000");                employeePhone.setPhoneNumbers(phoneNumberList);          }            return employeePhone;      }        @RequestMapping(value = "/names", method = RequestMethod.GET)      public EmployeeNames getEmployeeName()      {          log.info("get names Start");            EmployeeNames employeeNamesList = new EmployeeNames();            EmployeeName employeeName1 = new EmployeeName();          EmployeeName employeeName2 = new EmployeeName();            List<EmployeeName> employeeList = new ArrayList<EmployeeName>();          {              employeeName1.setFirstName("Santa");              employeeName1.setLastName("Singh");          }          {              employeeName2.setFirstName("Banta");              employeeName2.setLastName("Singh");          }            employeeList.add(employeeName1);          employeeList.add(employeeName2);            employeeNamesList.setEmployeeNameList(employeeList);            return employeeNamesList;      }  } |

**2.2. Async thread pool configuration**

|  |
| --- |
| AsyncConfiguration.java |
| import java.util.concurrent.Executor;    import org.springframework.context.annotation.Bean;  import org.springframework.context.annotation.Configuration;  import org.springframework.scheduling.annotation.EnableAsync;  import org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor;    @Configuration  @EnableAsync  public class AsyncConfiguration  {      @Bean(name = "asyncExecutor")      public Executor asyncExecutor() {          ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();          executor.setCorePoolSize(3);          executor.setMaxPoolSize(3);          executor.setQueueCapacity(100);          executor.setThreadNamePrefix("AsynchThread-");          executor.initialize();          return executor;      }  } |

**2.3. Spring @Async controller methods**

|  |
| --- |
| AsyncService.java |
| package com.howtodoinjava.example.async.service;    import java.util.concurrent.CompletableFuture;    import org.slf4j.Logger;  import org.slf4j.LoggerFactory;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.context.annotation.Bean;  import org.springframework.scheduling.annotation.Async;  import org.springframework.stereotype.Service;  import org.springframework.web.client.RestTemplate;    import com.howtodoinjava.example.async.model.EmployeeAddresses;  import com.howtodoinjava.example.async.model.EmployeeNames;  import com.howtodoinjava.example.async.model.EmployeePhone;    @Service  public class AsyncService {        private static Logger log = LoggerFactory.getLogger(AsyncService.class);        @Autowired      private RestTemplate restTemplate;        @Bean      public RestTemplate restTemplate() {          return new RestTemplate();      }        @Async("asyncExecutor")      public CompletableFuture<EmployeeNames> getEmployeeName() throws InterruptedException      {          log.info("getEmployeeName starts");            EmployeeNames employeeNameData = restTemplate.getForObject("<http://localhost:8080/name>", EmployeeNames.class);            log.info("employeeNameData, {}", employeeNameData);          Thread.sleep(1000L);    //Intentional delay          log.info("employeeNameData completed");          return CompletableFuture.completedFuture(employeeNameData);      }        @Async("asyncExecutor")      public CompletableFuture<EmployeeAddresses> getEmployeeAddress() throws InterruptedException      {          log.info("getEmployeeAddress starts");            EmployeeAddresses employeeAddressData = restTemplate.getForObject("<http://localhost:8080/address>", EmployeeAddresses.class);            log.info("employeeAddressData, {}", employeeAddressData);          Thread.sleep(1000L);    //Intentional delay          log.info("employeeAddressData completed");          return CompletableFuture.completedFuture(employeeAddressData);      }        @Async("asyncExecutor")      public CompletableFuture<EmployeePhone> getEmployeePhone() throws InterruptedException      {          log.info("getEmployeePhone starts");            EmployeePhone employeePhoneData = restTemplate.getForObject("<http://localhost:8080/phone>", EmployeePhone.class);            log.info("employeePhoneData, {}", employeePhoneData);          Thread.sleep(1000L);    //Intentional delay          log.info("employeePhoneData completed");          return CompletableFuture.completedFuture(employeePhoneData);      }  } |

**2.4. Call async methods and aggregate results**

|  |
| --- |
| package com.howtodoinjava.example.async.controller;    import java.util.concurrent.CompletableFuture;  import java.util.concurrent.ExecutionException;    import org.slf4j.Logger;  import org.slf4j.LoggerFactory;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.web.bind.annotation.RequestMapping;  import org.springframework.web.bind.annotation.RequestMethod;  import org.springframework.web.bind.annotation.RestController;    import com.howtodoinjava.example.async.model.EmployeeAddresses;  import com.howtodoinjava.example.async.model.EmployeeNames;  import com.howtodoinjava.example.async.model.EmployeePhone;  import com.howtodoinjava.example.async.service.AsyncService;    @RestController  public class AsyncController {        private static Logger log = LoggerFactory.getLogger(AsyncController.class);        @Autowired      private AsyncService service;        @RequestMapping(value = "/testAsynch", method = RequestMethod.GET)      public void testAsynch() throws InterruptedException, ExecutionException      {          log.info("testAsynch Start");            CompletableFuture<EmployeeAddresses> employeeAddress = service.getEmployeeAddress();          CompletableFuture<EmployeeNames> employeeName = service.getEmployeeName();          CompletableFuture<EmployeePhone> employeePhone = service.getEmployeePhone();            // Wait until they are all done          CompletableFuture.allOf(employeeAddress, employeeName, employeePhone).join();            log.info("EmployeeAddress--> " + employeeAddress.get());          log.info("EmployeeName--> " + employeeName.get());          log.info("EmployeePhone--> " + employeePhone.get());      }  } |

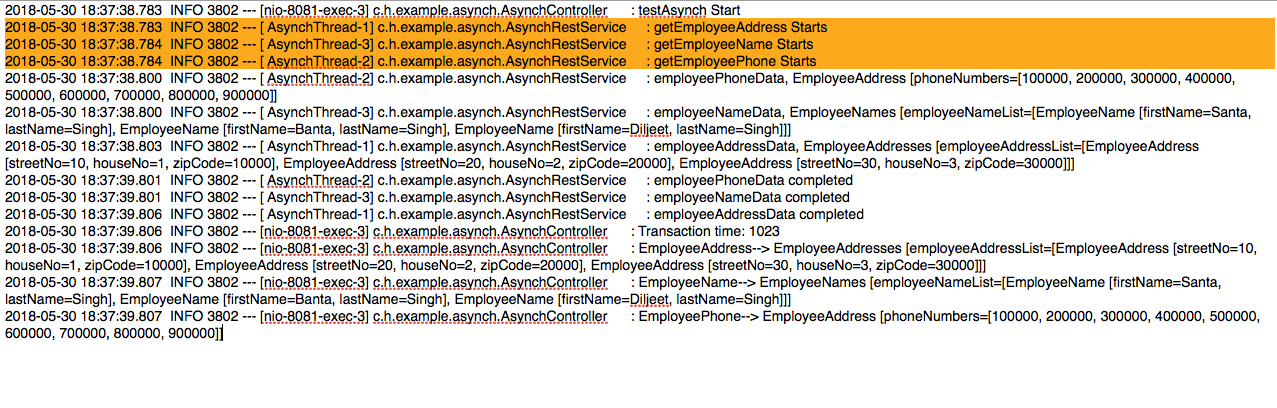
**2.5. How to run the demo**

Download and start both the applications.

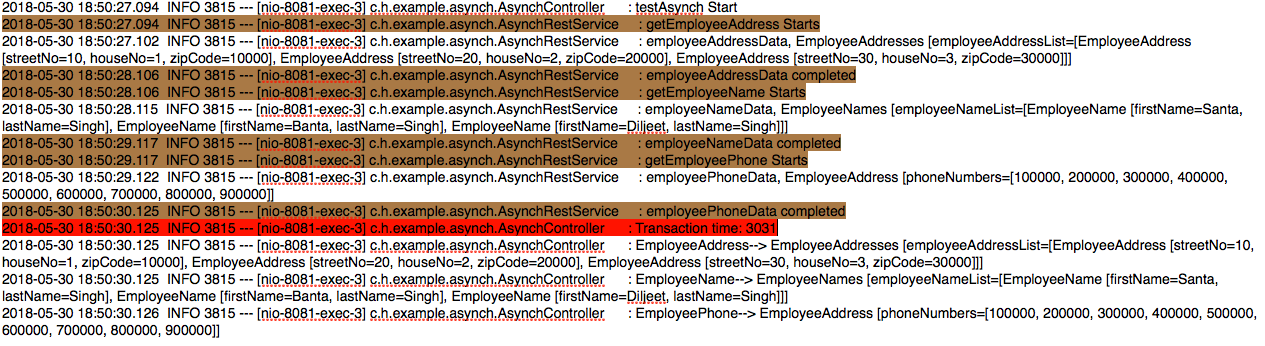
Hit the API: http://localhost:8081/testAsynch.

Observe the output in console.

**2.5.1. With @Aync Enabled**

With Aync Methods Enabled

**2.5.2. Without Aync Enabled**

Without Aync Methods Enabled

**From Different Article:** <https://medium.com/trendyol-tech/spring-boot-async-executor-management-with-threadpooltaskexecutor-f493903617d>

**Challenge :**In Trendyol, we are developing a micro service project for integrate some third party systems. In this integration, we should consume some pageable api’s in every two hours. In this progress micro service will consume api over than 200.000 times. Every api call using resources and managing these progress going to complex. We can not consume api in single thread, because every 2 hour starting new scheduler and progress should finish immediately. In this way we are choosing to use Asynchronous programming for this progress.  
In fact looks like simple to call every api request in asynchronous, but there is a threshold. If we start to consume over than 200.000 request in same time, every thread consume resources and if resources not enough system will block. In dockerized application workflow, your micro service container will down and start again when system block and not responding. This mean you will lose the state of your application and you should start progress again. Because of these you should make your asynchronous threads scaleable and non blocking.

**2. Summary :** In this article, i will try to explain important points of managing Spring Boot **@Async** feature with **ThreadPoolTaskExecutor**.

Enabling to asynchronous functionality in Spring Boot is very easy with using @EnableAsync annotation over any of Spring Boot configuration class.  
But managing is async not easy like this, you should analyze your application of how many threads will be running at same time and scalability. After that you should define an Executor (java.util.concurrent.Executor) implementation for manage your @Async functions.

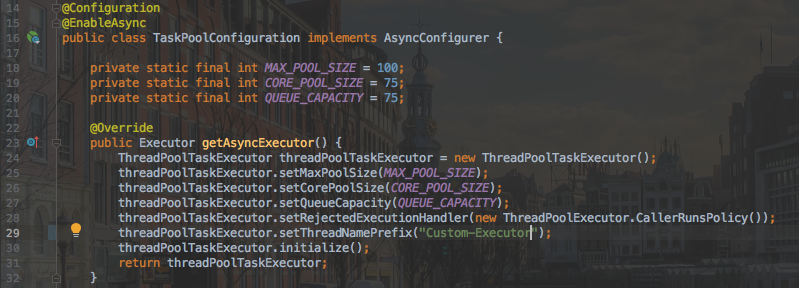
**3. Overview :**Spring’s TaskExecutor interface extends from java.util.concurrent.Executor interface. You can access TaskExecutor implementations from following link;  
<https://docs.spring.io/spring/docs/current/spring-framework-reference/integration.html#scheduling-task-executor-types>

In this article we will review **ThreadPoolTaskExecutor** implementation.

For activating your custom executor, you should create an **AsyncConfigurer** implementation and override getAsyncExecutor() function. Otherwise Spring will use it’s default Executor implementation and your configuration will not be active.

Sample of ThreadPoolTaskExecutor definition is like following picture. We will follow this definition in this article.

Image for post



ThreadPoolTaskExecutor contains following concepts;

**Core Pool Size :**Core pool size is defines minimum paralel threads can run at same time. In our sample, CORE\_POOL\_SIZE = 75 and this mean, our application can increase paralel running threads up to 75. If our application need more thread over than 75, new threads will be added into queue.

**Queue Capacity :**Queue is using when all core pool are filled. Threads will be scalable to maximum pool size when queue is full. In our application, let’s imagine 75 threads running at same time and also 75 threads more in queue. Totally we have 150 threads. Pool size will be increased until maximum pool size for each thread over 150.

**Maximum Pool Size :**Maximum pool size defines maximum parallel threads can run at same time. In our sample MAX\_POOL\_SIZE = 100 and this mean, our application can be increase to 100 parallel running threads when queue is full.

After all, we should decide what will happen to more thread when maximum pool and queue is full.  
Some approach is define queue is Integer.*MAX\_VALUE,*but i’m not suggested this approach. Maximum defined queue is means never will be filled and your application will never scale from core pool size to maximum pool size.  
Also some other approach is core pool and maximum pool size set same value and using queue maximum. This means fixed size thread pool without scalability. You can not decide your cpu and memory utilization.

Now let’s decide what will happen to more threads when maximum pool and queue is full. In Spring’s default , threads will be rejected with ThreadPoolExecutor.AbortPolicy and you lose new threads. For scalable application i’m suggested to use ThreadPoolExecutor.CallerRunsPolicy. This policy provides to us scalable queue when maximum pool is filled.

ThreadPoolTaskExecutor has following policies definitions;

**ThreadPoolExecutor.AboutPolicy** : Rejecting the thread with throwing RejectedExecutionException. You will lose the thread.  
**ThreadPoolExecutor.CallerRunsPolicy :**The thread invokes itself on rejected pool. You will not lose the thread. This policy like increasing queue capacity. You can manage your application with this policy.  
**ThreadPoolExecutor.DiscardPolicy :**The thread will be discard. You will lose the thread.  
**ThreadPoolExecutor.DiscardOldestPolicy :**This policy discards the oldest unhandled request. You will lose some threads inside queue.

**4. Conclusion :**This article will help to understand important concepts of Spring @Async feature. You can decide your ThreadPoolTaskExecutor configuration your with your application requirements. I’m suggesting to use monitoring tools for improve ThreadPoolTaskExecutor definitions. Please be careful about thread rejection policies otherwise you can use threads.