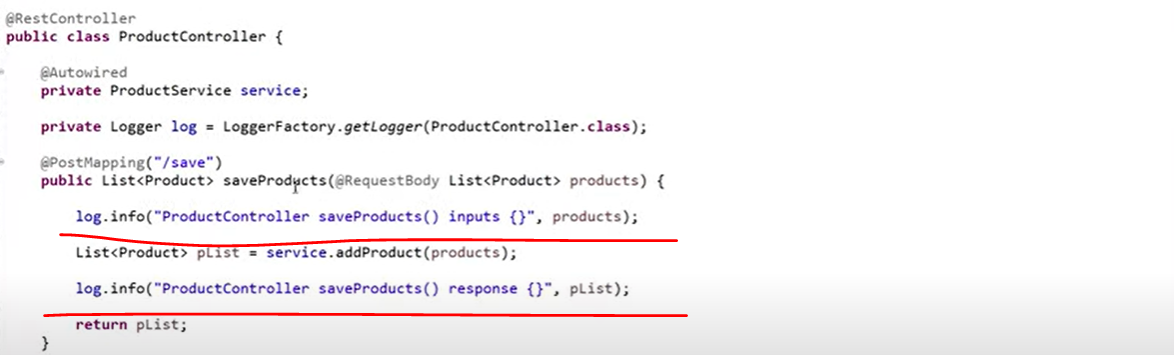
* **Logging - Spring AOP @Around Advice Centralized logging**

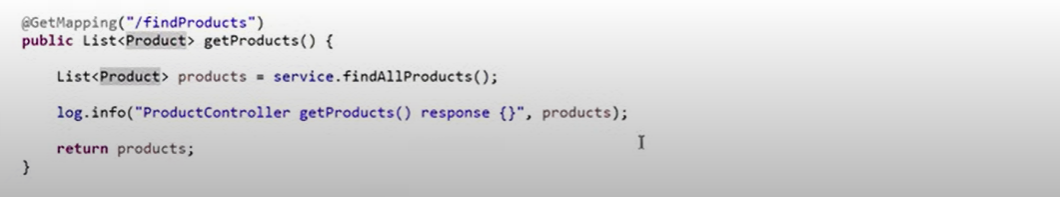
As we all known Logging features is very crucial feature in real time development. Because using a log file a developer can easily track a root cause of any failure.

Here I developed a CRUD application where I have 2 rest endpoints. One will save list of products and another one will fetch list of products from database.

If you observed here, I just want to capture the input request what I am getting from UI and here I added the log statement to capture the log statement returned back from service.



Similarly wrote the log statement to display the record from database.



The way we wrote log statement in our controller same way you need to write a log statement in your service, repository and in whatever the layer you are using. In real time you need to write log statement in each and every layer. So that easily you can track the flow of your application using this log statement. But sometimes you will find more log statement in your code compare to your actual business code.

**Now we need to think How we can decouple these log statements from my application for that we can go for SPRING AOP.**

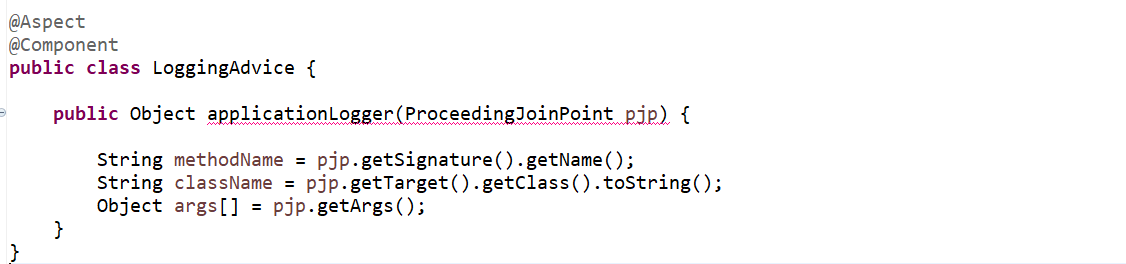
So, AOP stands for Aspect Oriented Programming language, which will help us to Segregate with your business logic to secondary logic. As I said this log statement is my secondary logic because without this log statement my actual business code can be execute. only we are adding this log statement to track the application flow. So how can we segregate this primary logic from the secondary logic using Spring AOP.

**So, we are going to Centralized our logging mechanism. Instead of writing log statement in our code we can write it in different layer so, that the logging feature can be applicable to my application.**

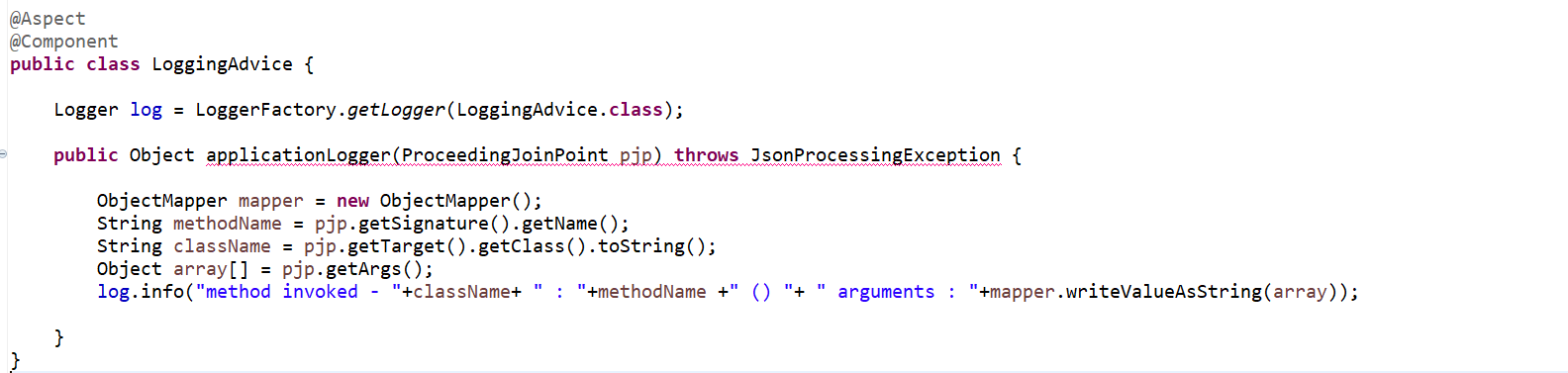
Let’s create a new package called “**com.java.selfdeveloped.advice**”

Let’s create a class called **LoggingAdvice** and I want to annotate this class with @**Aspect**.

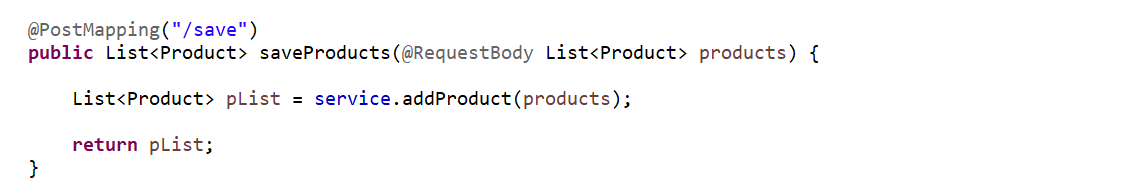
Let’s write a method and specify return type as an **Object** and in method argument we need to pass **ProceedingJoinPoint**.so, **ProceedingJoinPoint** came from the AOP. Internally this **ProceedingJoinPoint** internallyusing the Reflection API so that u can get the method details and what are the input parameters we can track at Runtime.



now we have configured all the required fields. Let’s add the log statement using this plugin Advice. So, instead of writing our logs in code we can managed in our **LoggingAdvice**. So, here in this log we can capture all these fields method Name, class Name and methods parameters. Let’s convert our array in proper json format.



So, we have added the log statement. This log statement will help let me show you. A person invoked this saveProduct() so whatever the input you are getting and what class it is invoking, and the method name it will display with the help of this syntax.

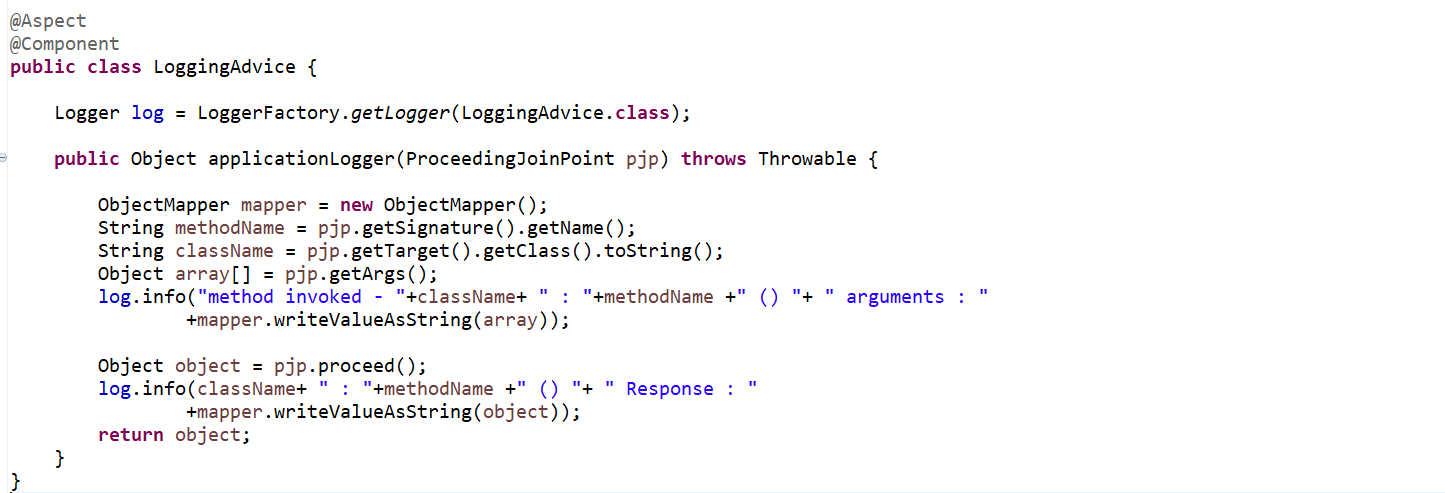


**I want to capture the Response as well. Once this method executes what exactly the response its giving that also I want to capture in my log. This is what we are going to implement the** @**Around Advice**. Around Advice is the combination of Before Advice and After Advice. So before executing this method I want to track the inputs and after executing this method I want to track the response. So that’s what we are going to implement Around advice for our log statement.

So, when you use below one it will give the return type object.

**Object object = pjp.proceed();**

This will give you a response which you will get from a service call



So, this is what my entire method which will take care my entire statement.

So, we centralized our logging mechanism.

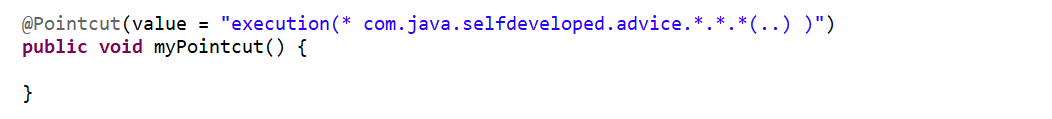
Now how we will inform Spring AOP that this is my Controller class, and this is what my methods for which I want to add the log statements. There is an annotation provided by Spring AOP that is called @**Pointcut**.

The role of Pointcut will inform to Spring AOP who will be my actual Target where I am going to implement this piece of code means logging mechanism.

Let’s specify the Pointcut here…...

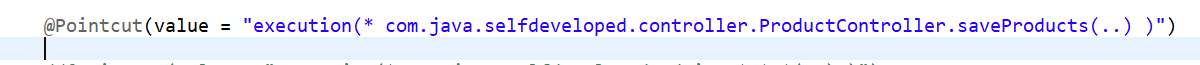
So here I need to provide a Pointcut as an expression so that Spring AOP can understand ok this is the path where I need to pass the customize logging statement. In Pointcut value either we need to provide the path of directory or the package or class or methods where exactly we want to implement this AOP.

So, we have a root package and then we have a controller class, service class and repository class. But here I want to implement this AOP concept across the application. So that’s the reason let me take this root directory path. So, where you want to exactly implement this AOP.



Here in execution, I want to implement for any methods I does not worry about the returned type that’s why I keep it as a \*, then root directory path then inside that any package any class any method with n number of arguments. That’s how I implement the logging mechanism.

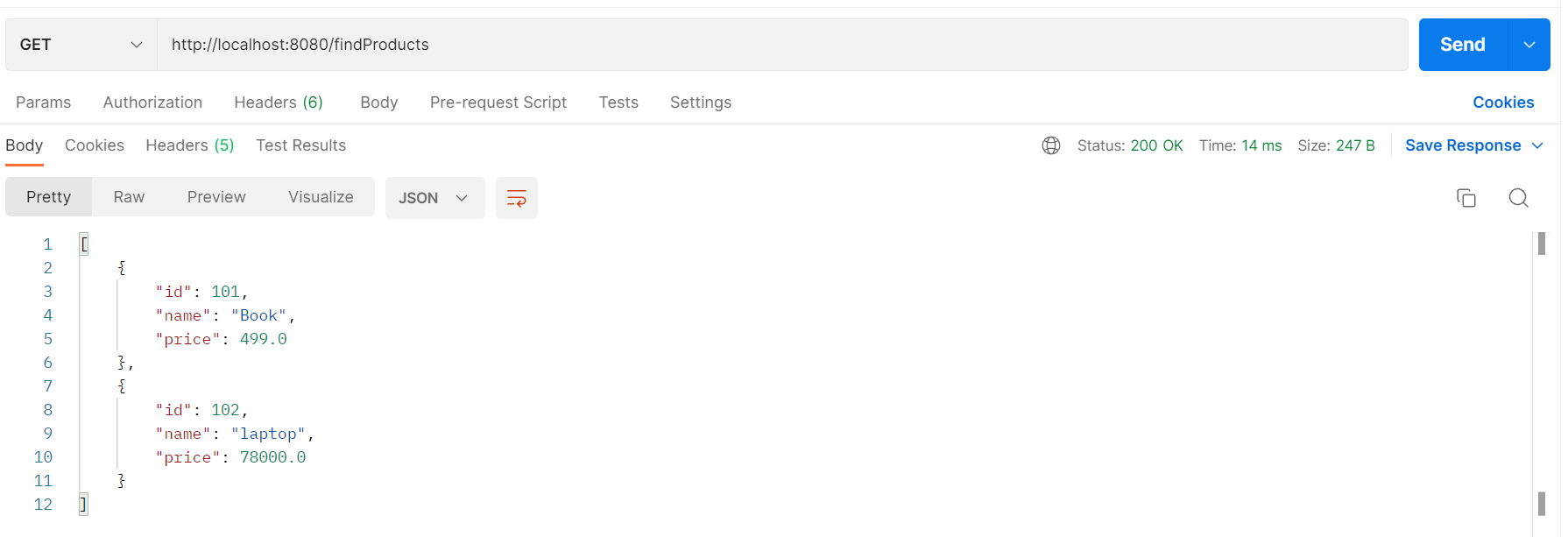
Let’s say he want to implement only for **ProductController**.

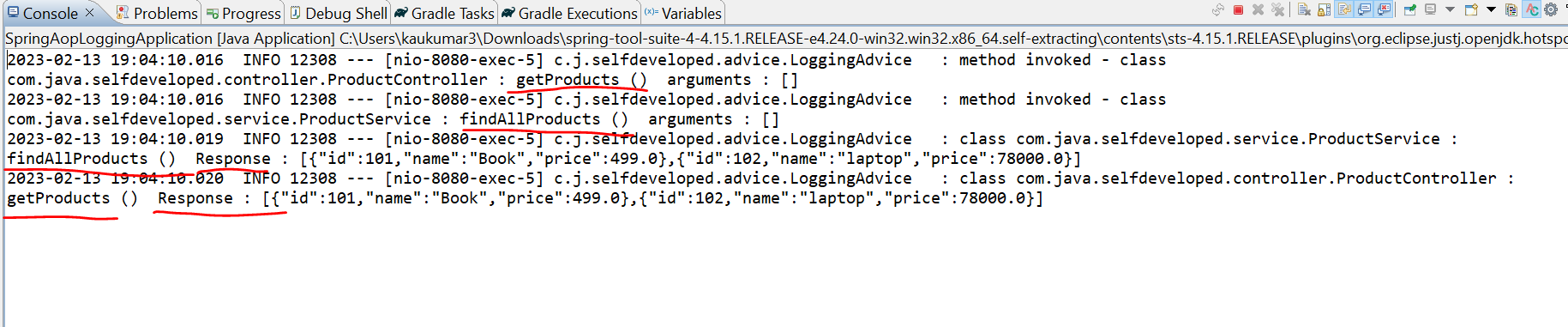


But here I want to apply for all the layers. For controller for service and for repository. that’s why I don’t want package, class, and method to specify here.

Now we need to specify @**Around** advice and in Around advice we need to specify the Pointcut.

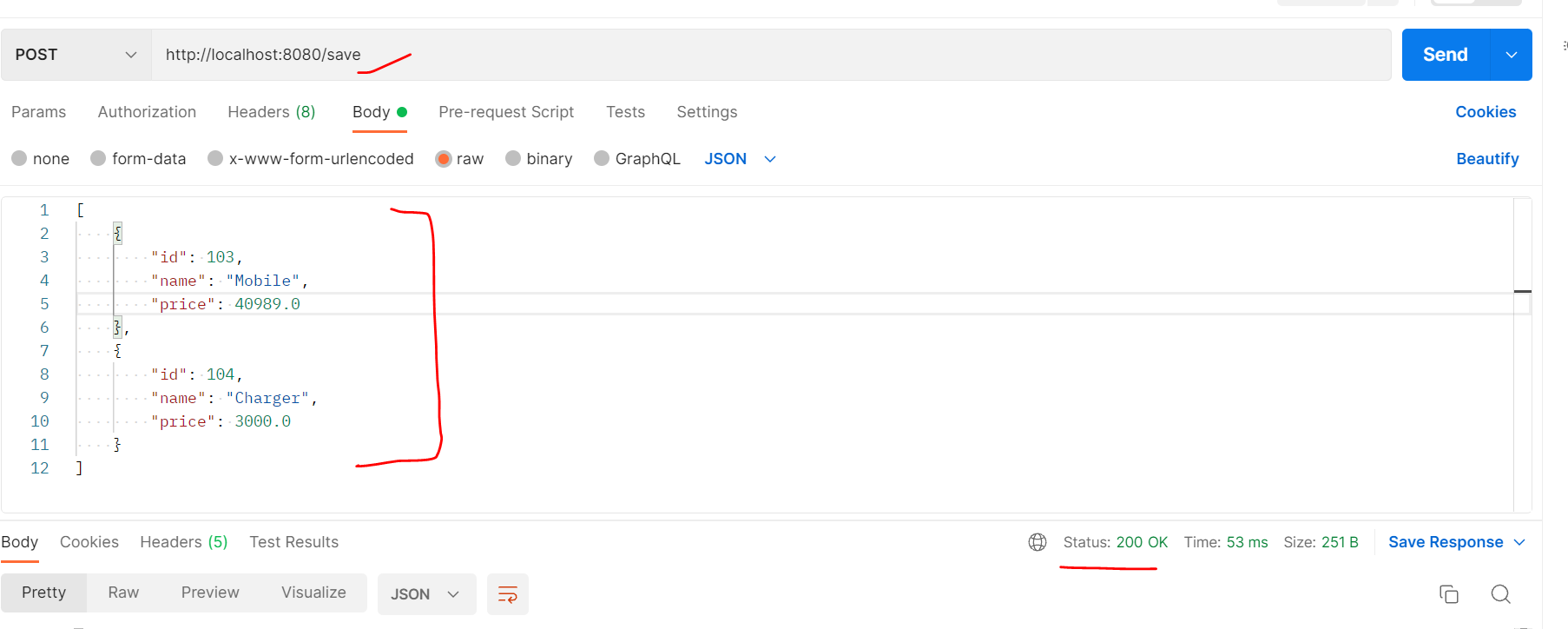


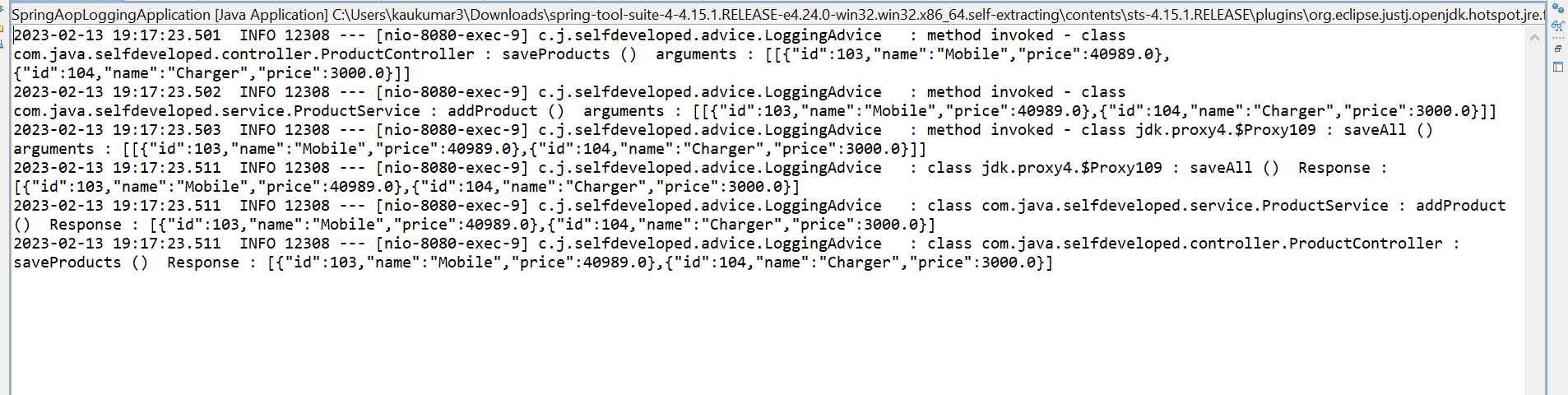




So if u observed here we are able to see our log statement method invoked then the class name is our **ProductController** and method is getProducts() as this is GET api so we don’t have any arguments here. similarly next control will goes to our **ProductService** and method call is **findAllProducts**() .

Then we got the Response one ProductService method is findAllProducts() and the response given as the list of 2 objects. Similarly, the response is going back to the controller, and we can see this 2 product objects. Now let’s hit our POST API.



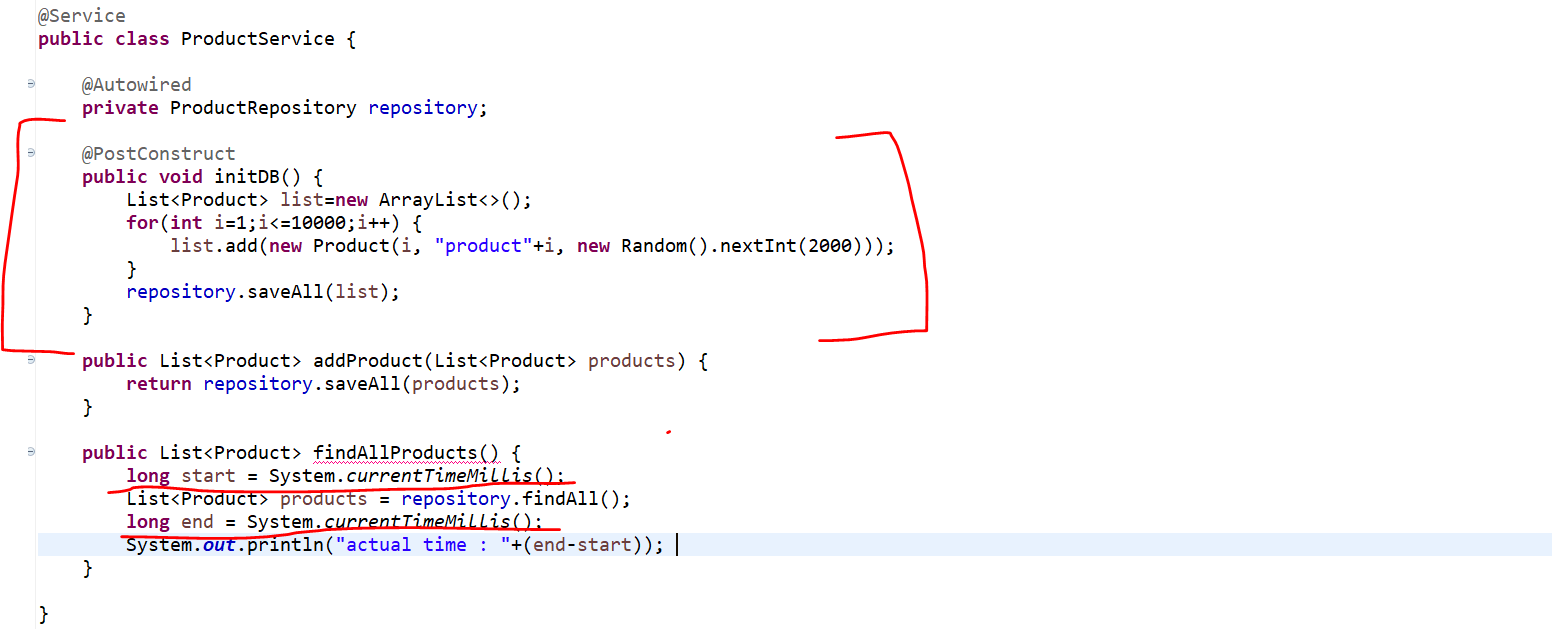


We can see the arguments what we are passing from Postman able to log them.

So, this is How we can centralize our Logging Mechanism instead of writing log statement in our code itself we can separate by creating our Advice Layer and we wrote our log statement.so that the secondary logic which is our logging mechanism we can segregate it from our Primary logic.

* **Spring AOP - Custom annotation to log method execution Time @TrackExecutionTime**

As a developer we need to focus on performance of our code means I should always expect the piece of code which I wrote should give faster response let's say I have a method who is doing some complex business so I want to evaluate how much time this method is taking to execute let's consider this example I have a product service so at the time of application load I am creating 10,000 product object and I am persisting them in database.



And I have a method called Find All products will page list of product object from database. So here I want to track execution time for this find all product method. I want to know how much time this method is taking to complete his execution I can do this, so this is very simple approach which people are following. what we can do we will use **system**.**currentTimeMillisecond**().

So, let's write long start time equal to **system**.**currentTimeMillisecond**() so I'll write **system**.**currentTimeMillisecond**() before and after method call.

And here instead of returning we can write list of product and at end we can return, or we can print the value for this end time minus start time. So, this particular where we are subtracting end time with our start time will give the actual result actual millisecond it will give so that we will get to know how much time this particular method is taking to execute. So, this will work no doubt.

But if same kind of track execution time I want in each and every method or let's say this is in service, I want same track execution time for controller. So how we can do this? Right? Again, we need to copy paste same kind of code across all the method in all the class which is not a feasible solution.

**Why not we will create our own custom annotation who will handle this execution time tracking.**

So, for that we can use Spring AOP around advice and we can create our own custom annotation. That's what we are going to learn in this tutorial. So, we'll create one custom annotation, and we can write on top of method so that that annotation will help us to evaluate execution time for a particular method.



So, let's quickly create a Spring boot project to demonstrate the **Spring AOP Around advice**. So, let me remove this piece of code which we just added which is cross cutting logic and that we want to segregate from our code.

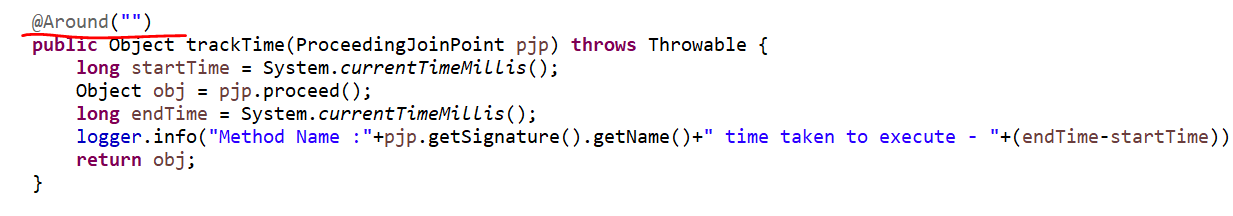
So, let's create a package called **advice**. Let’s create a class called **ExecutionTimeTrackerAdvice**.

As this is the Around advice we are going to use. We need to pass the proceeding join point as an argument. Then we need to get the start time and end time. Once we called the

**ProceedingJoinPoint pjp = new ProceedingJoinPoint();**

**pjp.proceed()**

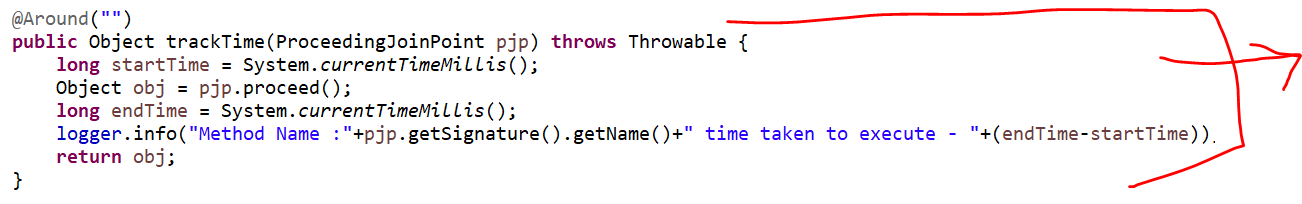
**it will behave like an Around advice so, before this method call it will behave like a Before Advice and once, we call this method and after that it will behave as a After Returning Advice that’s what the Around Advice we learnt in our earlier tutorial.** Now we get the start time before method call and end time after method call. Now we can easily log the actual execution time.



Here we created a method which will log execution time. What we need to do we need to annotate here @**Around**() advice, and we need to specify the Pointcut here.

**But I don’t want to apply this track Time logic in a specific method or in a specific class. I just want to create an annotation who will execute this business logic. So, for that instead of giving pointcut here we will create an annotation and we will give the fully qualified name of the annotation here.**

So, whenever you will use that annotation it will execute this piece of code.

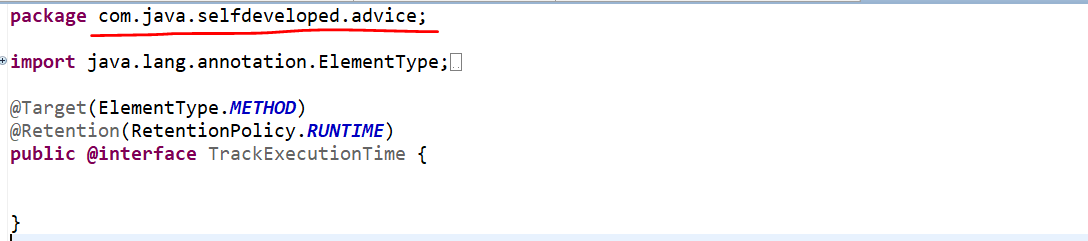


**So, this piece of code that annotation will execute. Let me create an annotation first…**

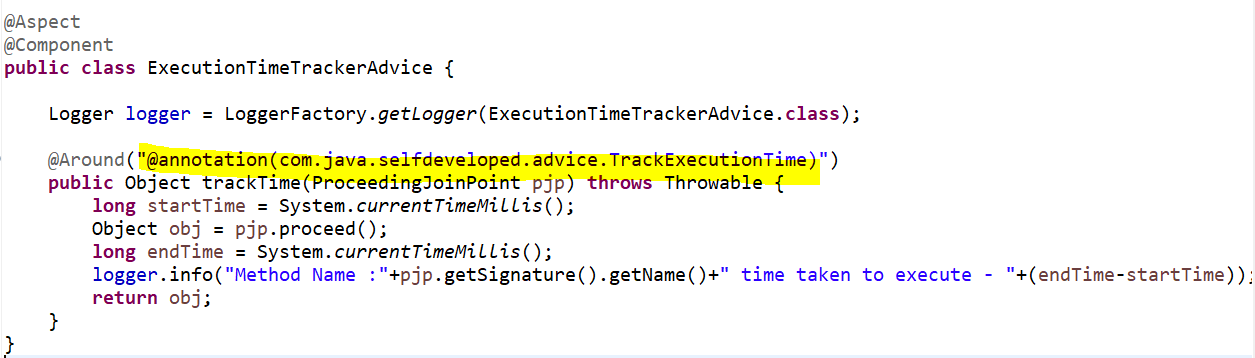
So, in advice package only let me create an annotation called **TrackExecutionTime** and this will be an Annotation.



So, this **TrackExecutionTime** is my custom annotation. U need to annotate with **@Target** to use onwhich level we want to specify this annotation, so we want to apply on method level. **@Retention** this only to use either we want to apply at compile time or run time. So, we want it to be apply on Run-Time.



We created this annotation, and we need to give fully qualified path of this annotation to our Around advice.



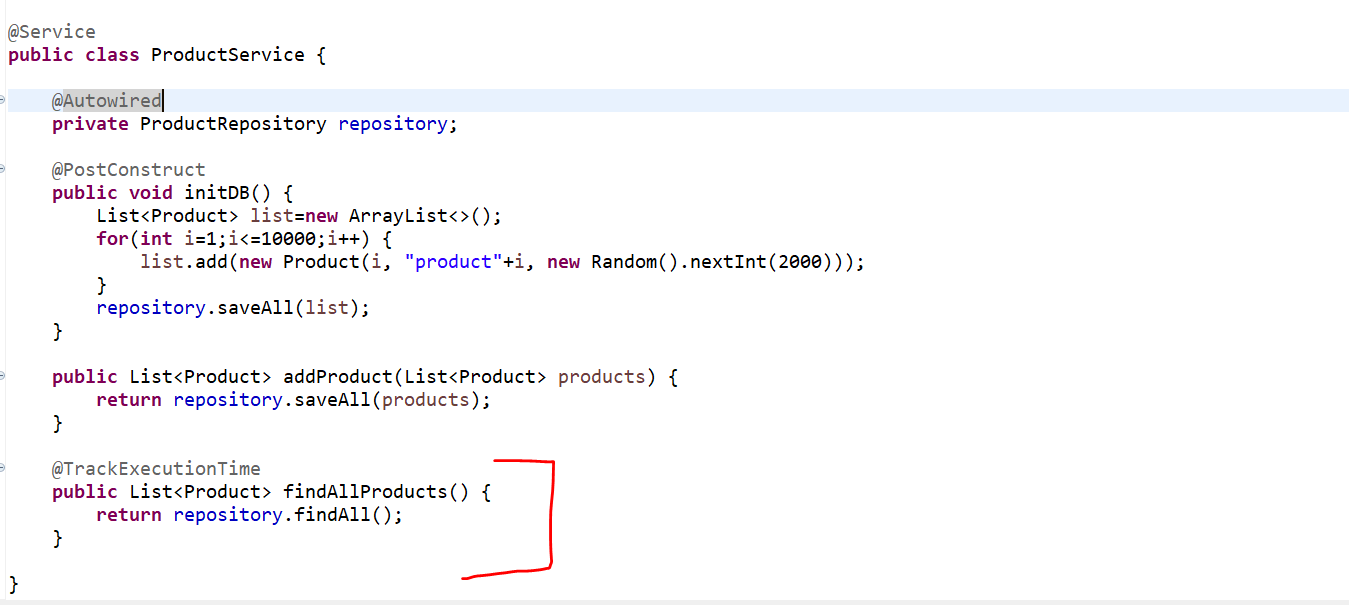
Now wherever I use this **TrackExecutionTime** annotation the request or the control will come to this advice which is our Around advice. And this piece of code will execute. That’s what I didn’t specified the pointcut here I specified the fully qualified annotation here.

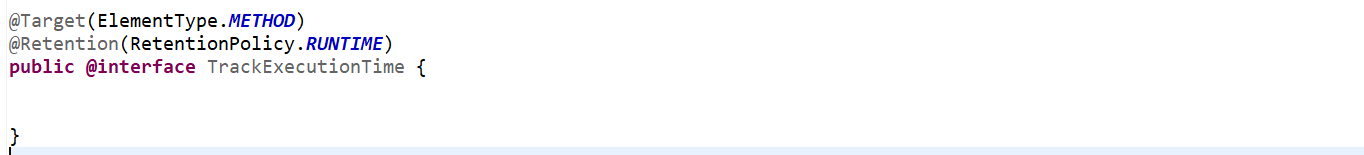
**So, let’s use this annotation in our method level where exactly we want to track our execution time.**

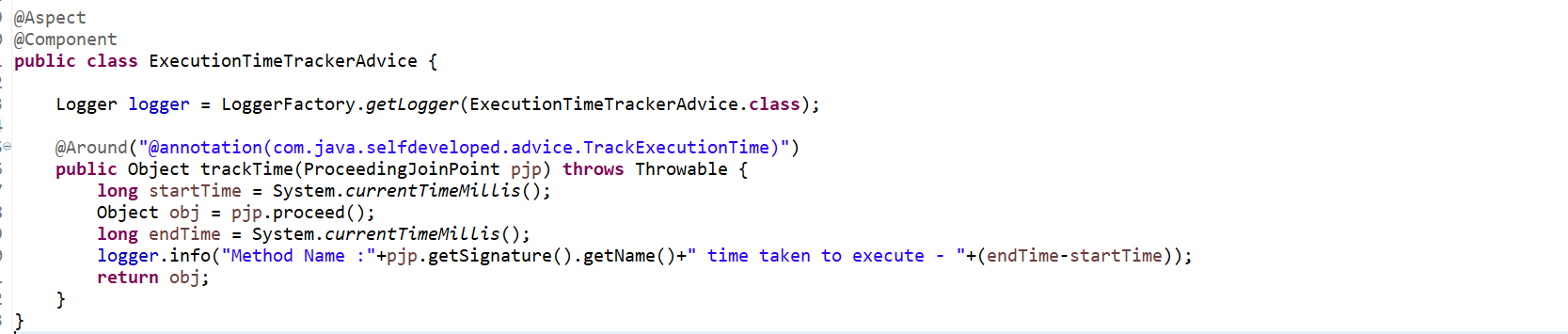


Let’s say I just want to track the execution time for this particular **findAllProducts**() method.

So what we can do we can use our own annotation **TrackExecutionTime** now once the Request will come to findAllProducts() it will go to based on annotation it will go to the advice this **TimeTrackerExecutionAdvice** and then it will execute that piece of code.



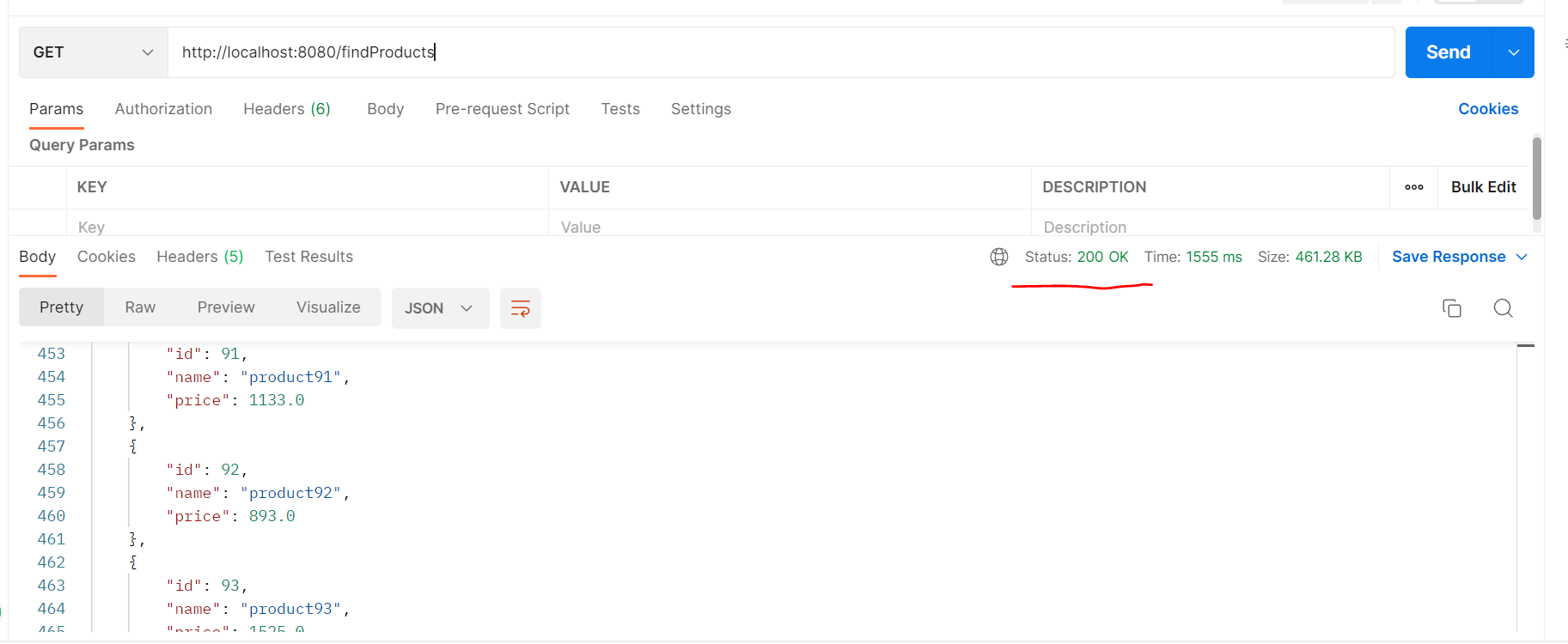




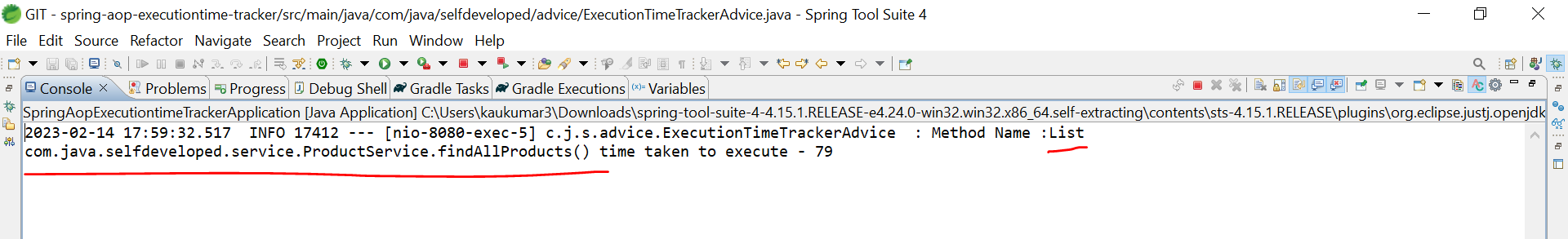
So, now let’s run and verify …

We are hitting one of the endpoints….and we are getting 10000 records.

<http://localhost:8080/findProducts>



Let’s go to console to verify our log statements.

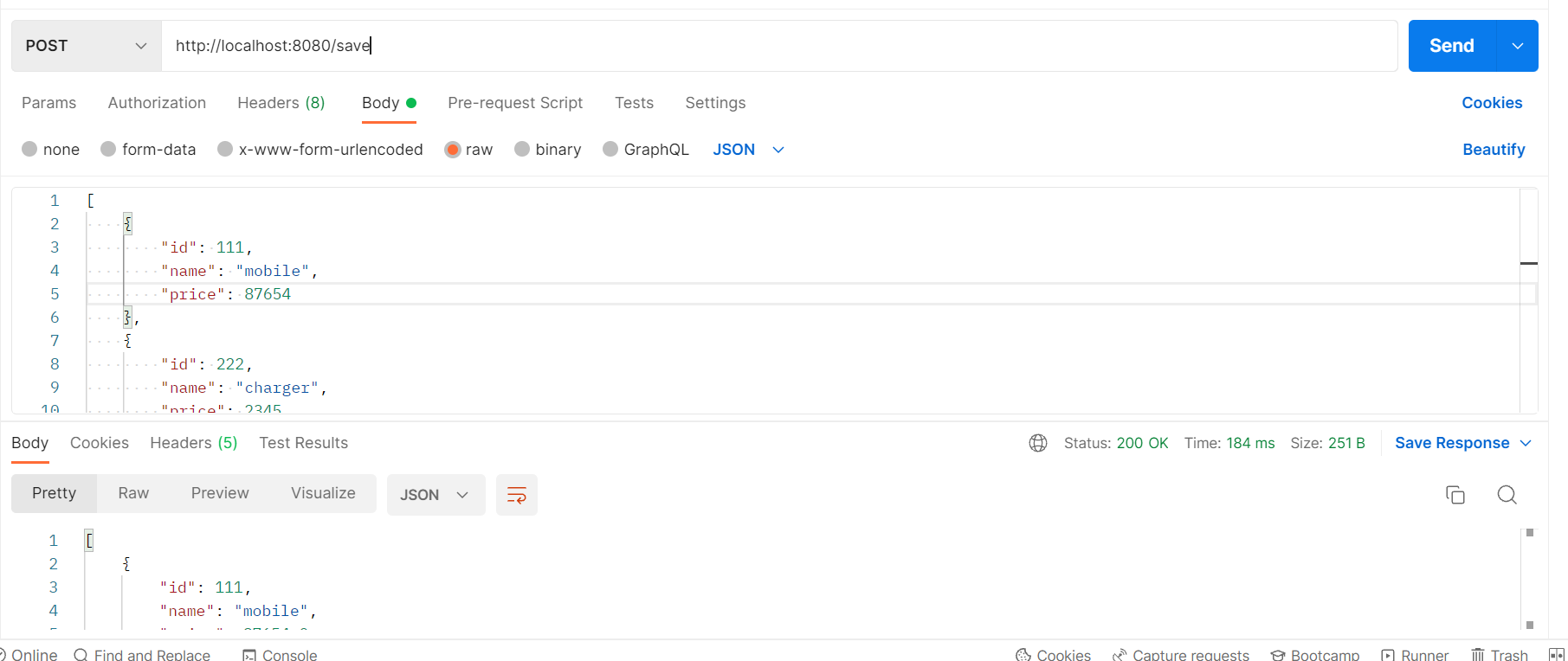


So, it took time 79 miliseconds to execute.

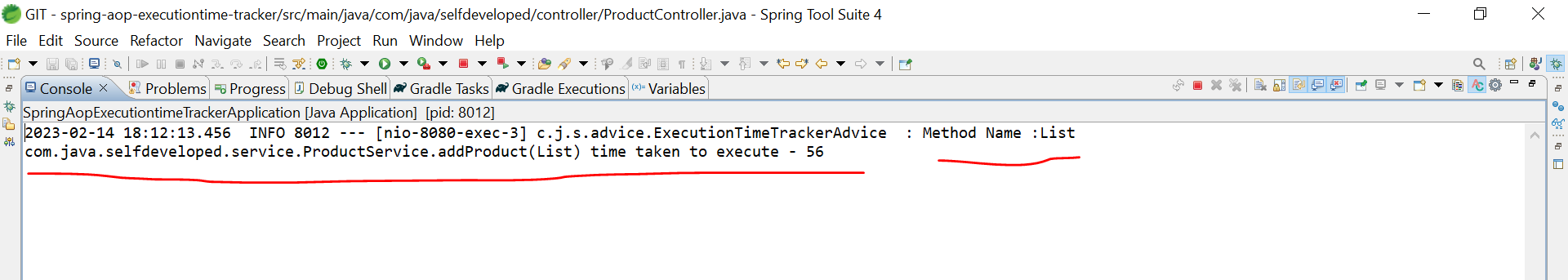
If u want to track for other method as well let’s go to ProductService and let’s say I want to track the execution time for **addProduct()** as well. So, u can do that using the annotation like…



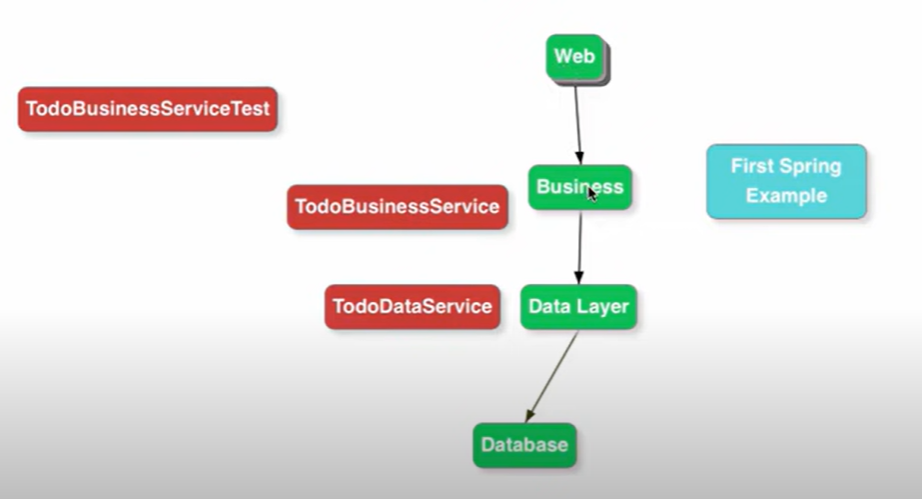
<http://localhost:8080/save>



So, wherever you want to get the execution time as a log you can simply use this annotation.



* **How do we implement Aspect Programming with Spring Framework**



**We have a typical web application. So, I have the Web Layer, Web Layer called the Service Layer and Service Layer called the Data Layer. What I want to do is every time when a business message is called, I would like to log those particular statements.**

For example, let’s say I want to log what is the method call, what are the input parameters that are there and what would be the return value that is written back there. I want to log all these information along with how much time it took to execute that particular method. If I want to do this kind of things if there are nothing like Aspect oriented programming the way I would need to do that I have to implement the logic in every method so every time we call the method in the business service in that particular method, we would need to do all the logic. Let’s see an example----

**What are the different options available to implement Aspect Orient Programming using Spring ?**

If you are using basic functionality of AOP then go with AOP which provided by Spring itself otw If u need complex AOP functionality for example if u want to intercept something intercept a method call which is not managed by spring so, something which is not managed by spring so spring can’t intercept it so in that case u should go for AOP framework like **AspectJ.**

What all terminology uses in Aspect Oriented Programming ?

* **Aspect**
* **Pointcut**
* **JoinPoint**
* **Advice**
* **Weaving**

**@Aspect** - Aspect is a kind of functionality which I would like to implement. For example, if I want to implement transaction management using Aspect Oriented Programming then Transaction Management is an Aspect which we are trying to implement. So, aspect is like Logging, Transaction Management, and performance metrics. like how much time methods took to execute then Performance Metrics is an Aspect. **So, Aspect is like what you want to achieve through Aspect Oriented Programming**. In this example we are looking is **Logging** what I want to do is? **At every method call in Business Layer I would want to intercept it and I would want to log the input parameters and what is the outputs.**

**@Pointcut** - A Pointcut you can think it’s like a Regular Expression. Some kind of expression which determines what calls are to be intercepted. So, the example of pointcut is-

**execution**(\* com.java.selfdeveloped.business. aop.HiByeService.\*(..))

so, any method calls on **HiByeService** we want to intercept. So, that is basically the Pointcut. **So Pointcut defines what would the things you want to intercept.** That’s the expression is called Pointcut. When do you want aspect orient programming to be applied that is the things which is define by Pointcut.

**Aspect is basically the functionality which I want to Implement that’s Logging and Pointcut is actually the expression which decides when the Aspect is Apply.**

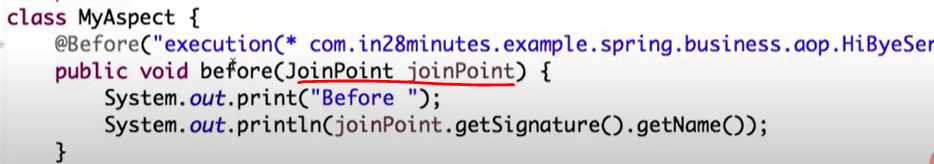
So, when is Logging done at every method call in **HiByeService** and that’s defines by the Pointcut.

**@Advice** – **Advice is basically what is to be done when the Pointcut is met. Like I just want to do a log when pointcut met.**

The things we discussed till now is Static or Compile Time Terminology. At Compile Time you can decide You Know what is the **Aspect**, You Know what is the **Pointcut**, You Know what is the **Advice**.

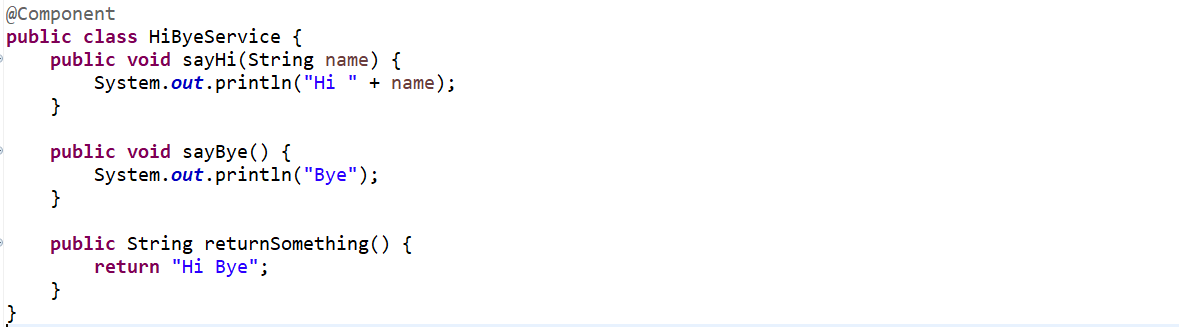
**The Next what we are looking at is a more Dynamic Stuff are a Runtime Stuff.**

**@Join Point – JoinPoint is more a Runtime thing. So, when I am executing and a Pointcut is matches the JoinPoint is comes into the picture. JoinPoint is basically the execution of this specific AOP methods that we have define. At runtime once every condition is made this method is executed and execution of this method is called JoinPoint.**

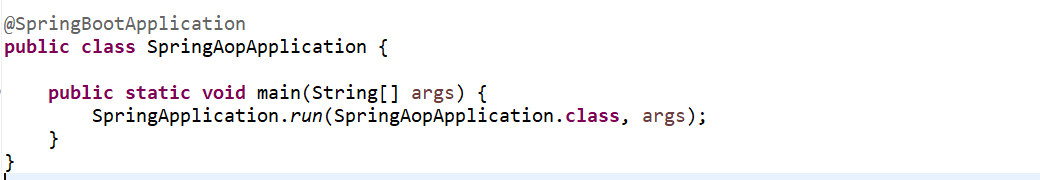


**Weaving – When this particular condition or Pointcut is met it should make sure that this methos is executed that process of wired the whole things together is called Weaving. So, whenever this Pointcut Mets these methods need to be executed the entire process around that is called weaving.**

Let’s see an example:-



**What I want to do Before every method call, I just want to print something and after the method is completed execution, I would want to print something out. I want to do this in all these 3 methods.**



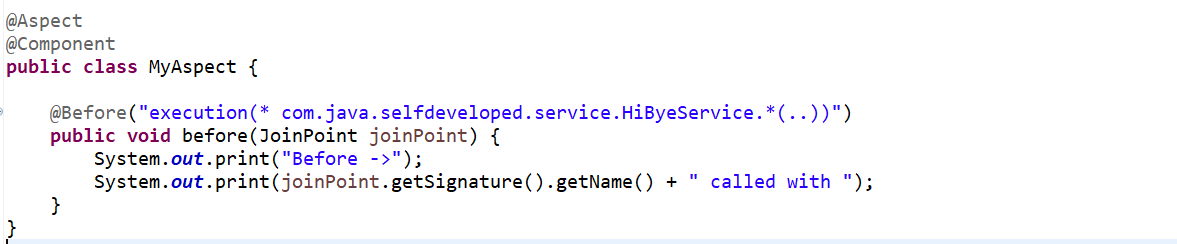
**So, whenever this sayHi is called I want to print something. So how we will do that.**

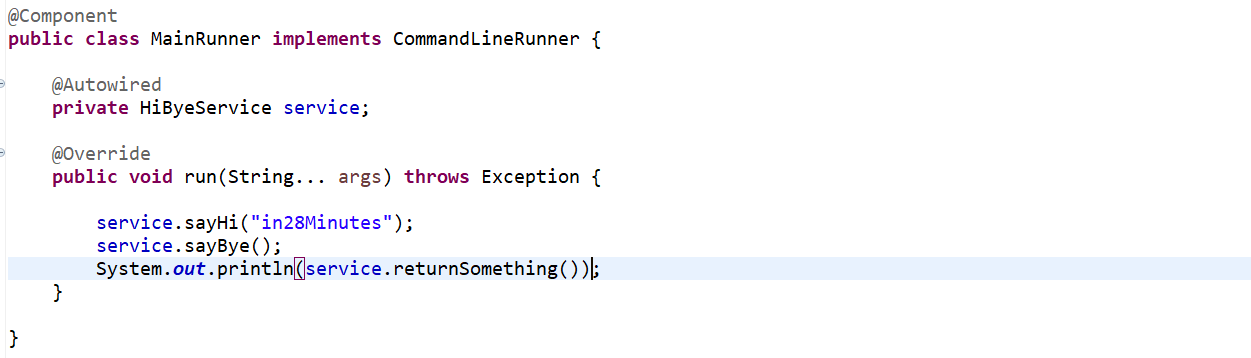
First thing I am creating a class **MyAspect** and this class is a critical thing and I am annotating this class with @**Aspect**. It also has to be managed by Spring so let’s annotate with @**Component**.

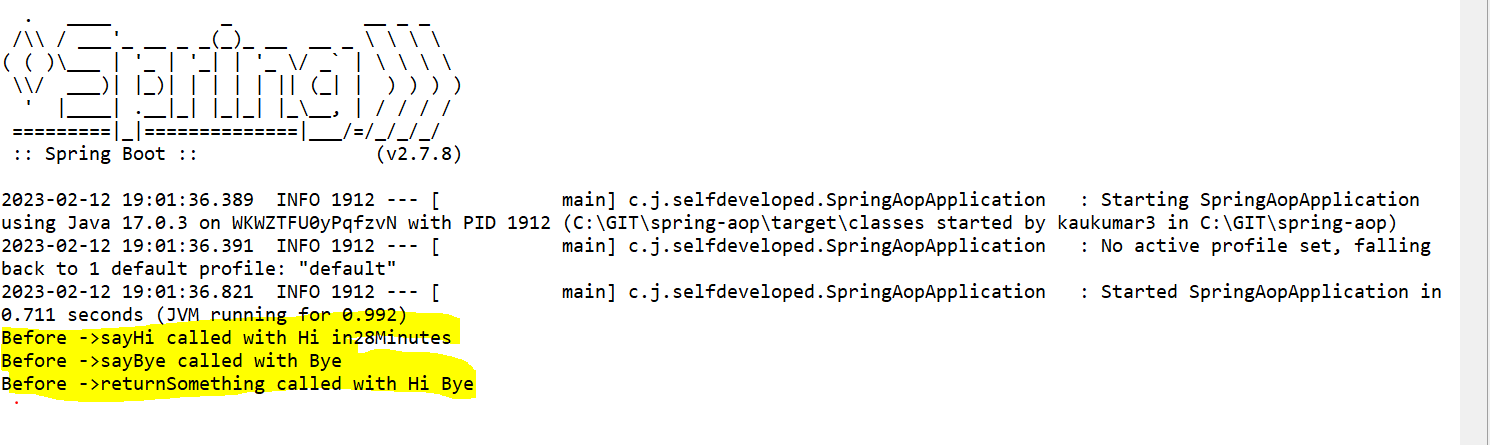
The first thing I am doing here is @**Before** annotation, and this is used to identified something I want to do before the method execution of a particular method, and I am defining the **Pointcut**. So, I am defining the Pointcut to say this the entire package where this **HiByeService** is there. I would want to intercept every methods call on this particular class **HiByeService.** This is the method which would what we want to do when the Pointcut is made.

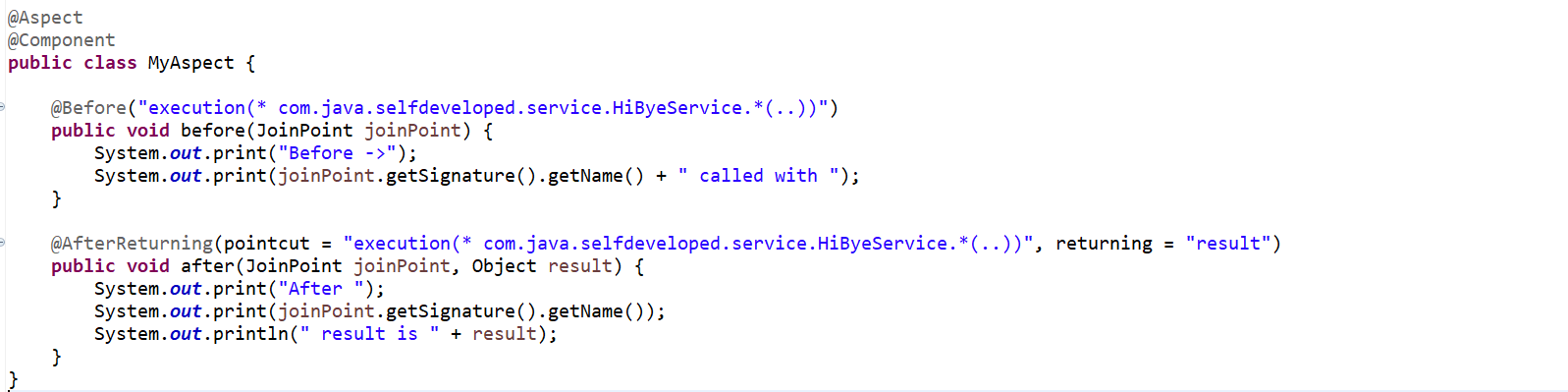
The **JoinPoint** basically contains all the information’s about the method call all the runtime information’s about the method call. So, at runtime what is the method that is being called that u can get using joinPoint.getSignature().getName().

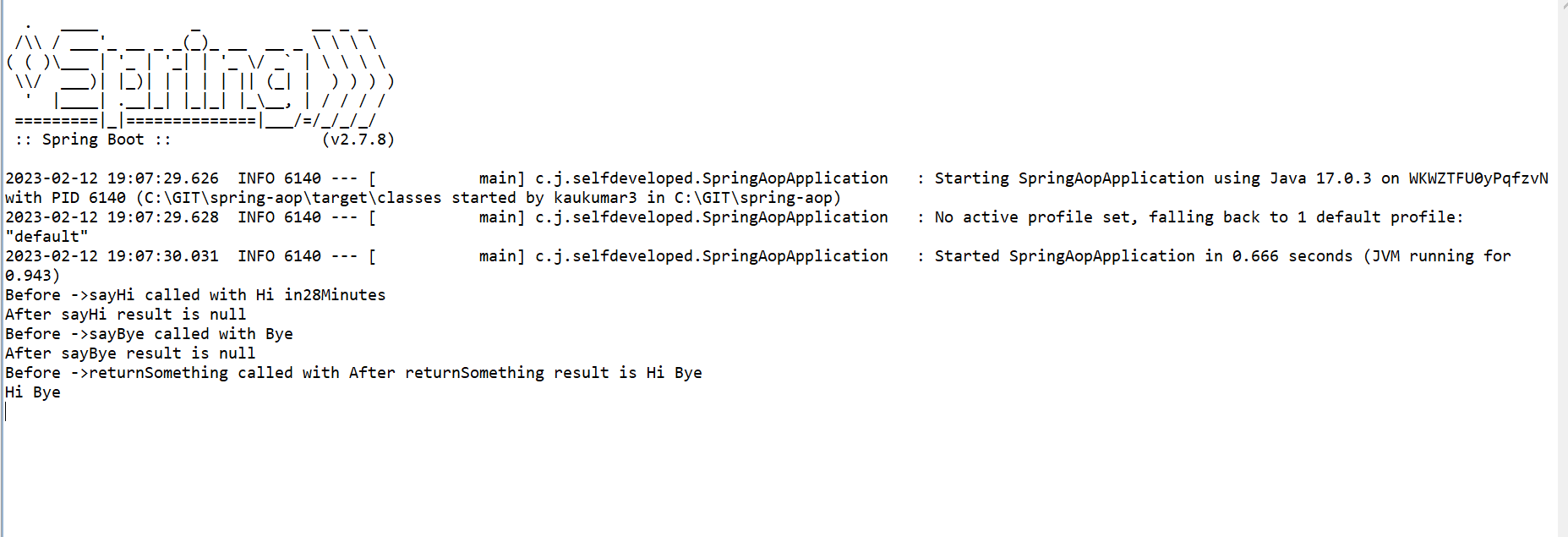
Let’s run and see…



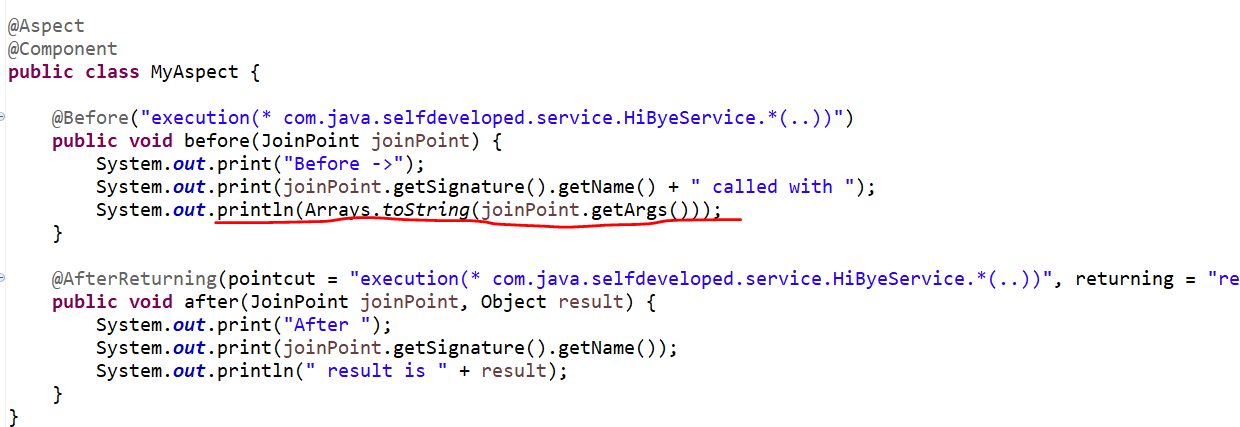


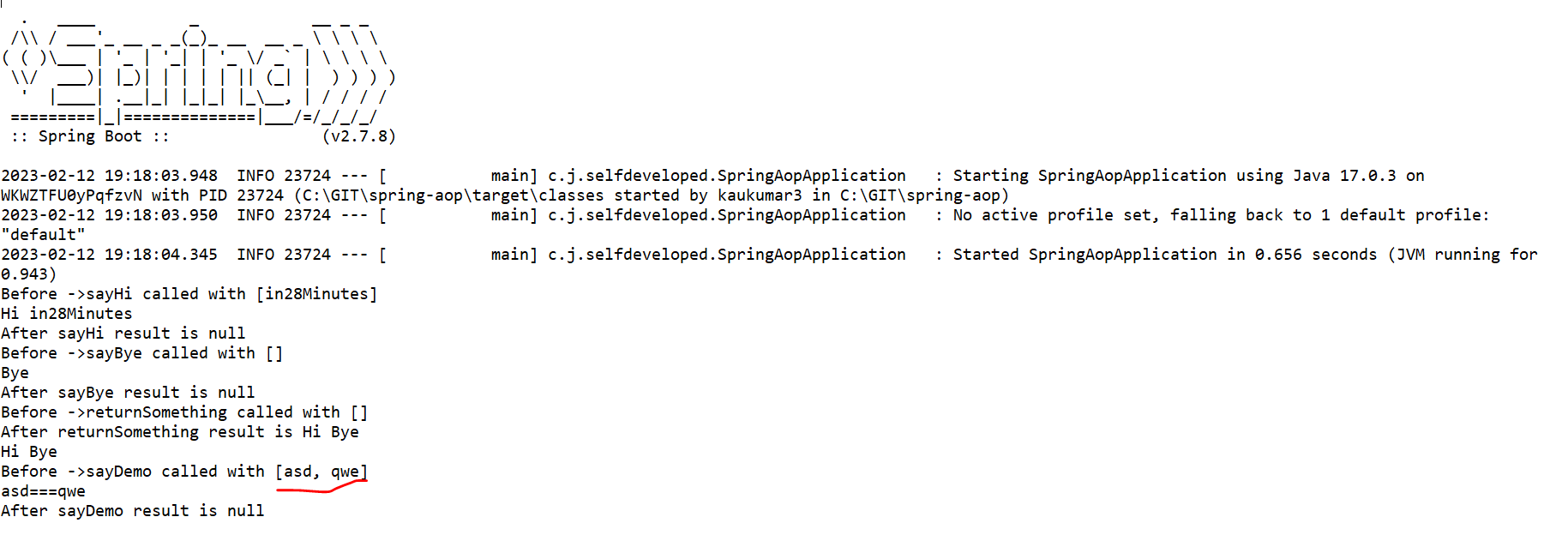






Let’s say this particular method had a particular method had a couple of arguments being passed. So, I want to do like.





All Code:-

@SpringBootApplication

**public** **class** SpringAopApplication {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(SpringAopApplication.**class**, args);

}

}

-----------------------------------------------------------------------------------------------------------------------------

@Component

**public** **class** MainRunner **implements** CommandLineRunner {

@Autowired

**private** HiByeService service;

@Override

**public** **void** run(String... args) **throws** Exception {

service.sayHi("in28Minutes");

service.sayBye();

System.***out***.println(service.returnSomething());

service.sayDemo("asd","qwe");

}

}

-------------------------------------------------------------------------------

@Component

**public** **class** HiByeService {

**public** **void** sayHi(String name) {

System.***out***.println("Hi " + name);

}

**public** **void** sayBye() {

System.***out***.println("Bye");

}

**public** String returnSomething() {

**return** "Hi Bye";

}

**public** **void** sayDemo(String a, String b) {

System.***out***.println(a+"==="+b);

}

}

---------------------------------------------------------------------------------------------------------------------------------------

@Aspect

@Component

**public** **class** MyAspect {

@Before("execution(\* com.java.selfdeveloped.service.HiByeService.\*(..))")

**public** **void** before(JoinPoint joinPoint) {

System.***out***.print("Before ->");

System.***out***.print(joinPoint.getSignature().getName() + " called with ");

System.***out***.println(Arrays.*toString*(joinPoint.getArgs()));

}

@After("execution(\* com.java.selfdeveloped.service.HiByeService.\*(..))")

**public** **void** after(JoinPoint joinPoint) {

System.***out***.print("After -> ");

System.***out***.println(joinPoint.getSignature().getName());

}

@AfterReturning(pointcut = "execution(\* com.java.selfdeveloped.service.HiByeService.\*(..))", returning = "result")

**public** **void** after(JoinPoint joinPoint, Object result) {

System.***out***.print("After Returning -> ");

System.***out***.print(joinPoint.getSignature().getName());

System.***out***.println(" result is " + result);

}

}

