Creating Class Dynamically Using ByteBuddy and SpringBoot

2021-09-27 23:13:00 +0530

# Introduction

In this post, we are going to talk about how we can generate a class dynamically. Welcome to the brave new world of powerful possibilities. Generating classes dynamically as a concept, is as old as the Java language itself. RMI, J2EE, etc. used this powerful concept to a large extent. With the advent of Annotations in JDK 1.5, this became very popular. You had libraries pre-processing annotations and generating proxies to do away with lot of boiler-plate code. Especially worth mentioning are the Aspect Oriented Programming concepts (or AOP) around Transaction management. Suffice it to say that Spring and lot of other popular libraries like Hibernate, would not be able to function as they do, without this.

There are a number of very powerful libraries that allow you to generate classes dynamically. I had used [Apache BCEL](https://commons.apache.org/proper/commons-bcel/) a [bit longish back](https://palashray.com/your-cup-of-byte-code-re-engineering-bcel-style/). However, today, we will explore a simple, yet powerful library called [ByteBuddy](https://bytebuddy.net/#/).

# Let’s build…

I am building a trivial Web Application with Spring Boot. It uses versioned REST APIs. Nothing fancy, just that each of the end points starts with its version. For example, the V1 of my API would be <http://localhost:8080/v1/bank-item/>. Similarly, no prizes for guessing, that the V2 of my API would be <http://localhost:8080/v2/bank-item/>.

# The Challenge

I have already built the V1 of my REST end point. And of course, I have a RestController for that. Now, I am feeling too lazy to build the V2 of the same end point, as I know that would mean I have to write a DAO, a Service, the Model, the entire stack, before I even get to my RestController. And I love taking short cuts. So, can I extend my existing RestController for V1, and create another RestController such that it starts responding to my V2 requests? And can I do this dynamically too?

Assuming, we already have an existing RestController called **BankDetailController**, serving the V1 endpoint, basically what we would like to do is, create a class similar to below:

@RestController  
@RequestMapping("/v2/bank-item")  
public class BankDetailControllerV2 extends BankDetailController {  
  
 public BankDetailControllerV2(BankDetailService bankDetailService) {  
 super(bankDetailService);  
 }  
  
}

Let’s see: well, there are many ways we could achieve this, but for the sake of brevity, we will stick to just 2 approaches.

# First Approach

We create the class file dynamically using ByteBuddy and save it to the disk, somewhere in my classpath. I do this before Spring starts scanning the classpath and creating beans. When Spring starts scanning my classpath, the newly created classfile is already available to it. If I decorate that class with the correct annotations, Spring will pick it up and do the rest.

## The right life-cycle hook

The challenge really is to find the correct life-cycle hook, so that we can create this class in the right phase. I found that the [EnvironmentPostProcessor](https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/env/EnvironmentPostProcessor.html) to be the right thing. This is actually used to customize Environment variables, so this is a bit of a hack, but that’s OK. It is going to be invoked right after the Environment is created, but before any beans are initialized. Looks like just the thing for us.

So, we go ahead and create our own **DynamicControllerPostProcessor**, which implements **EnvironmentPostProcessor**. Remember to create a **META-INF/spring.factories** file and put the below entry there for this class to be invoked:

org.springframework.boot.env.EnvironmentPostProcessor=com.swayam.demo.springbootdemo.dynamicclassesbytebuddy.config.DynamicControllerPostProcessor

Since our Loggers are not yet initialized, remember to use plain *System.outs* instead of Loggers.

This is how **DynamicControllerPostProcessor** looks like:

@Order(Ordered.LOWEST\_PRECEDENCE)  
public class DynamicControllerPostProcessor implements EnvironmentPostProcessor {  
  
 @Override  
 public void postProcessEnvironment(ConfigurableEnvironment environment, SpringApplication application) {  
 // put the code to create dynamic class here  
 }  
}

## Creating the class dynamically

Here is the snippet for creating our cool new class:

private void createDynamicController() {  
  
String className = BankDetailController.class.getName() + "V2";  
  
System.out.println("Creating new class: " + className);  
  
Unloaded<?> generatedClass =  
new ByteBuddy().subclass(BankDetailController.class)  
 .annotateType(AnnotationDescription.Builder.ofType(RestController.class).build(),  
 AnnotationDescription.Builder.ofType(RequestMapping.class)  
 .defineArray("value", new String[] { "/v2/bank-item" }).build())  
 .name(className).make();  
  
Loaded<?> loadedClass =  
generatedClass.load(getClass().getClassLoader(), ClassLoadingStrategy.Default.INJECTION);  
  
try {  
 Map<TypeDescription, File> map = loadedClass.saveIn(new File("target/classes"));  
 System.out.println("Successfully saved the newly created class in: " + map);  
} catch (IOException e) {  
 throw new RuntimeException(e);  
}  
  
}

Notice, how we begin by instructing **ByteBuddy** to *subclass* the **BankDetailController**. Next, we create the *@RestController* annotation using the **AnnotationDescription.Builder**. Then, we create the *@RequestMapping* annotation, and pass the *String[]* argument by using the *defineArray*. Finally, we add the *className*, which is a fully qualified className with the right package.

This class is then loaded into the ClassLoader and duly saved on the disk. The assumption, of course is that the *target/classes* is on my classpath.

## Sources

The source code can be found here: <https://github.com/paawak/spring-boot-demo/tree/master/dynamic-classes-bytebuddy/actual-classfile-example>.

# Second Approach

The first approach works reasonably well. There is one hitch though: we have to save the actual classfile on the disk. That might not be ideal in certain circumstances. For example, I am running this inside of a Docker container, and I don’t have write permissions.

What if, we could create the class dynamically, but, instead of actually saving it on the disk, just load it into the ClassLoader and use it? Say, create a Spring Bean dynamically?

## Dynamically defining a Spring Bean

Turns out that the [BeanFactoryPostProcessor](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/config/BeanFactoryPostProcessor.html) is just the thing for us. This is primarily used for customizing the beans defined in the BeanFactory. We can also leverage this is to register our custom Spring Bean dynamically.

This is how it is done in **DynamicBeanFactoryPostProcessor**:

@Configuration  
public class DynamicBeanFactoryPostProcessor implements BeanFactoryPostProcessor {  
  
 private static final Logger LOG = LoggerFactory.getLogger(DynamicBeanFactoryPostProcessor.class);  
  
 @Override  
 public void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) throws BeansException {  
  
 Class<?> dynamicController = ...//create the class here and load it into the classloader  
  
 BeanDefinitionBuilder beanDefinitionBuilder = BeanDefinitionBuilder.rootBeanDefinition(dynamicController)  
 .addConstructorArgReference("bankDetailServiceImpl");  
  
 ((DefaultListableBeanFactory) beanFactory).registerBeanDefinition(className,  
 beanDefinitionBuilder.getBeanDefinition());  
  
 }  
}

We use the **BeanDefinitionBuilder** to define our bean. Before that, the first step is to create the class dynamically and then load it into the ClassLoader. You do not need to save it on the disk. We pass in our freshly baked class into the **rootBeanDefinition()**. As our RestController also has a constructor dependency, we would need to define that as well. We use the **addConstructorArgReference()**, which takes in a String instead of the actual Object. This is really convenient, as you do not have to manually resolve all dependency.

Once we are good with our Bean Definition, we can register that in the **DefaultListableBeanFactory**.

## Creating a class in-memory

Almost forgot: how do we do the first step of creating the class and loading it without actually saving it disk? Very simple:

private Class<?> createDynamicController(String className) {  
  
LOG.info("Creating new class: {}", className);  
  
Unloaded<?> generatedClass =  
new ByteBuddy().subclass(BankDetailController.class)  
 .annotateType(AnnotationDescription.Builder.ofType(RestController.class).build(),  
 AnnotationDescription.Builder.ofType(RequestMapping.class)  
 .defineArray("value", new String[] { "/v2/bank-item" }).build())  
 .name(className).make();  
  
generatedClass.load(getClass().getClassLoader(), ClassLoadingStrategy.Default.INJECTION);  
  
LOG.info("Loaded the class {} successfully into the classloader {}", className, getClass().getClassLoader());  
  
try {  
 return Class.forName(className);  
} catch (ClassNotFoundException e) {  
 throw new RuntimeException(e);  
}  
  
}

After the call to **generatedClass.load()**, the class has been loaded into the ClassLoader, so we can do a **Class.forName()** and return the Class representation with which we can work.

## Sources

The source code can be found here: <https://github.com/paawak/spring-boot-demo/tree/master/dynamic-classes-bytebuddy/in-memory-example>.