FAQ: What is a real-time use case for @Bean?

Here is a real-time use case of using @Bean: *You can use @Bean to make an existing third-party class available to your Spring framework application context.*

For example, I was recently working on a global real-time project using Amazon Web Services. The project made use of the Amazon Simple Storage Service (AWS S3). This is remote service that provides object storage in the cloud. You can think of AWS S3 at a high-level as a remote file server for storing files (pdfs, pngs etc).

Our Spring application needed to integrate with AWS S3 and store pdf documents. Amazon provides an AWS SDK for integrating with AWS S3. Their API provides a class, S3Client. This is a regular Java class that provides a client interface to the AWS S3 service. We needed to share the S3Client object in various services in our Spring application. However, the S3Client does not have the @Component annotation. The S3Client does not use Spring.

Since the S3Client is part of the AWS framework, we can't modify the source code for the S3Client directly. We can't simply add the @Component annotation to the S3Client source code. As a result, we need an alternative solution.

But no problem, by using the @Bean annotation, I can wrap this third-party class, S3Client, as a Spring bean. And then once it is wrapped using @Bean, it is as a singleton object and available in our Spring framework application context. I can now easily share this bean in my app using dependency injection and @Autowired. So think of the @Bean annotation was a wrapper / adapter for third-party classes. You want to make the third-party classes available to your Spring framework application context.

Here's a real-time example

Here is a snippet from our @Configuration class. We create an instance of the S3Client and wrap it as a Spring bean. The default scope is singleton. It is now available in our application context and we can inject it to other parts of our Spring application using @Autowired.

```
    @Bean
    public S3Client remoteClient() {

            // Create an S3 client to connect to AWS S3
            S3Client s3Client = S3Client.builder().region(Region.of(region))

.credentialsProvider(StaticCredentialsProvider.create(awsCreds)).build();

            return s3Client;
    }
```

In the code below, this is a Spring service that uses the S3Client. The service @Service annotation is a subclass of @Component. This code uses @Autowired to inject the bean named "remoteClient". This bean was created in the configuration code above using @Bean.

Once the bean is injected, then our method can use this to interact with the Amazon S3 service. In this real-time project, we were processing insurance claims. We store the PDF invoices in the cloud using the AWS S3 service.

```
    @Service
    public class InsuranceClaimsServiceImpl implements ClaimsService {

        @Autowired
        private S3Client remoteClient;

        ...

        public void processClaim(Claim theClaim) {


            // read claim data
            FileData fileData = theClaim.getFileData("payerInvoice");
            String fileName = theClaim.getSubmittedFileName();

            // get the input stream and file size
```

```
                InputStream fileInputStream = fileData.getInputStream();
                    long contentLength = fileData.getSize();

                //
            // store claim data in AWS S3
            //

            // Create a put request for the object
            PutObjectRequest putObjectRequest = PutObjectRequest.builder()
                                .bucket(bucketName)
                                .key(subDirectory + "/" + fileName)

.acl(ObjectCannedACL.BUCKET_OWNER_FULL_CONTROL).build();

            // perform the putObject operation to AWS S3 ... using our
autowired bean
            remoteClient.putObject(putObjectRequest,
RequestBody.fromInputStream(fileInputStream, contentLength))
        }
    }
```

As you can see, I was able to wrap a third-party class as a Spring bean. The AWS S3Client object was not originally annotated with @Component. The S3Client is not aware of Spring. But I could manually wrap it using @Bean. By doing this, the object is now available in our Spring application context. We can now share/reuse this bean in other areas of our Spring app by using dependency injection and @Autowired.

For other services in our application, if they need access to the S3client (singleton) then they can simply inject it using @Autowired. No need for each service to create a new instance of the S3Client every time. This keeps the application efficient in terms of memory and performance.

In summary: *You can use @Bean to make an existing third-party class available to your Spring framework application context.*