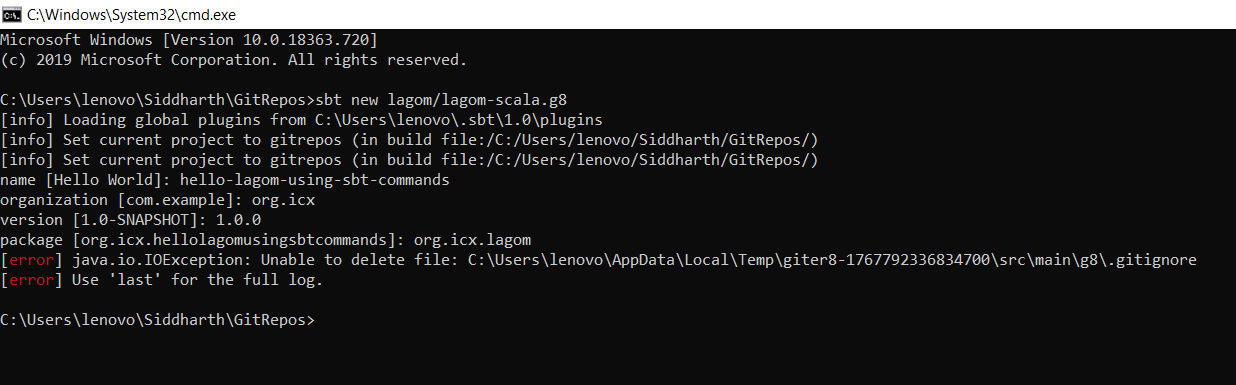
# **Creating a Lagom project and a most basic microservice**

This document captures an effort to create a microservice in the Lagom Framework with bare minimum things required. It doesn’t go all the way to explain things in detail. This is being done in parallel with reading through the Lagom documentation. So, there’s not a lot of explaining. It is a quick go through and will make the reader understand what are the bare minimum things required to create a microservice in lagom and make it show up on service locator and accessible via service gateway.

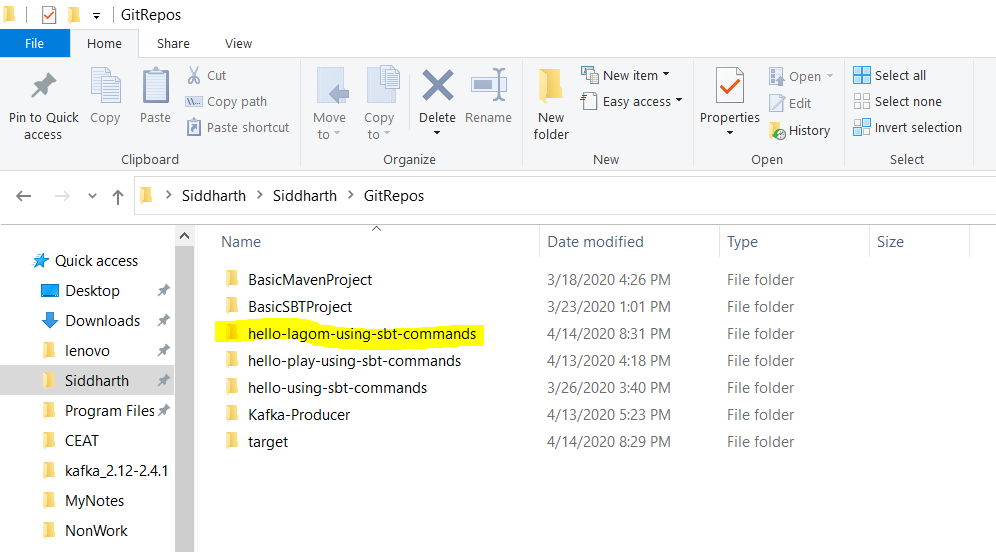
Start with a giter8 template for Lagom.

Run “sbt new lagom/lagom-scala.g8” in the directory you want to create project in.

You will be prompted for some basic things like name, organization, version, package name etc.

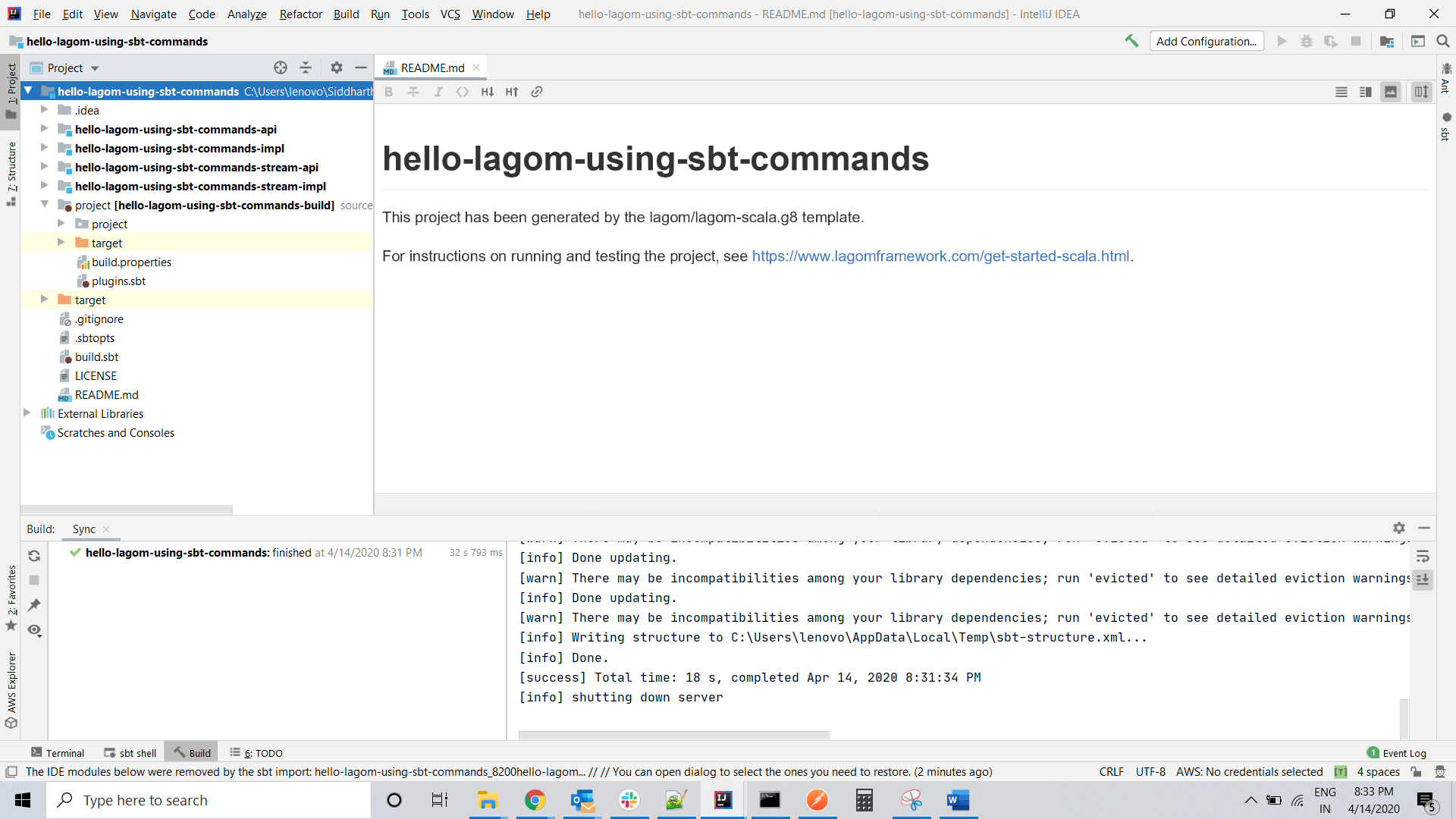


This will create a sample project for you.

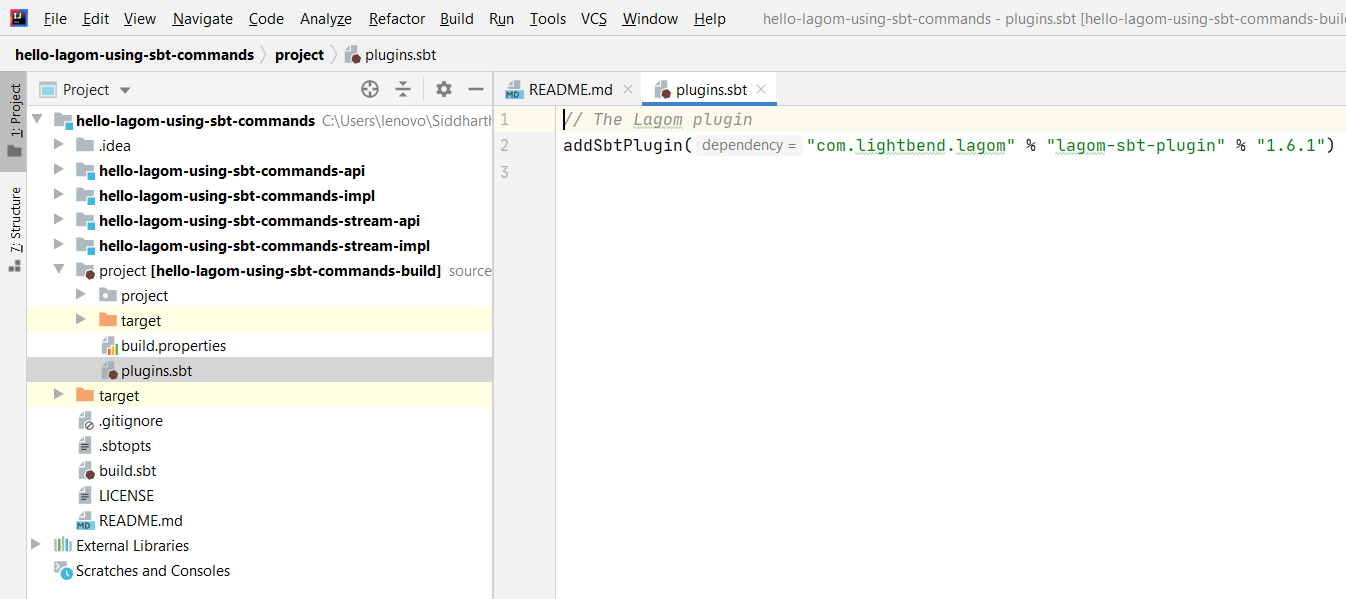


Open the project in your IDE. Snapshots are for IntelliJ. It assumes that IntelliJ is setup to use Scala SDK and Scala plugin is installed too.

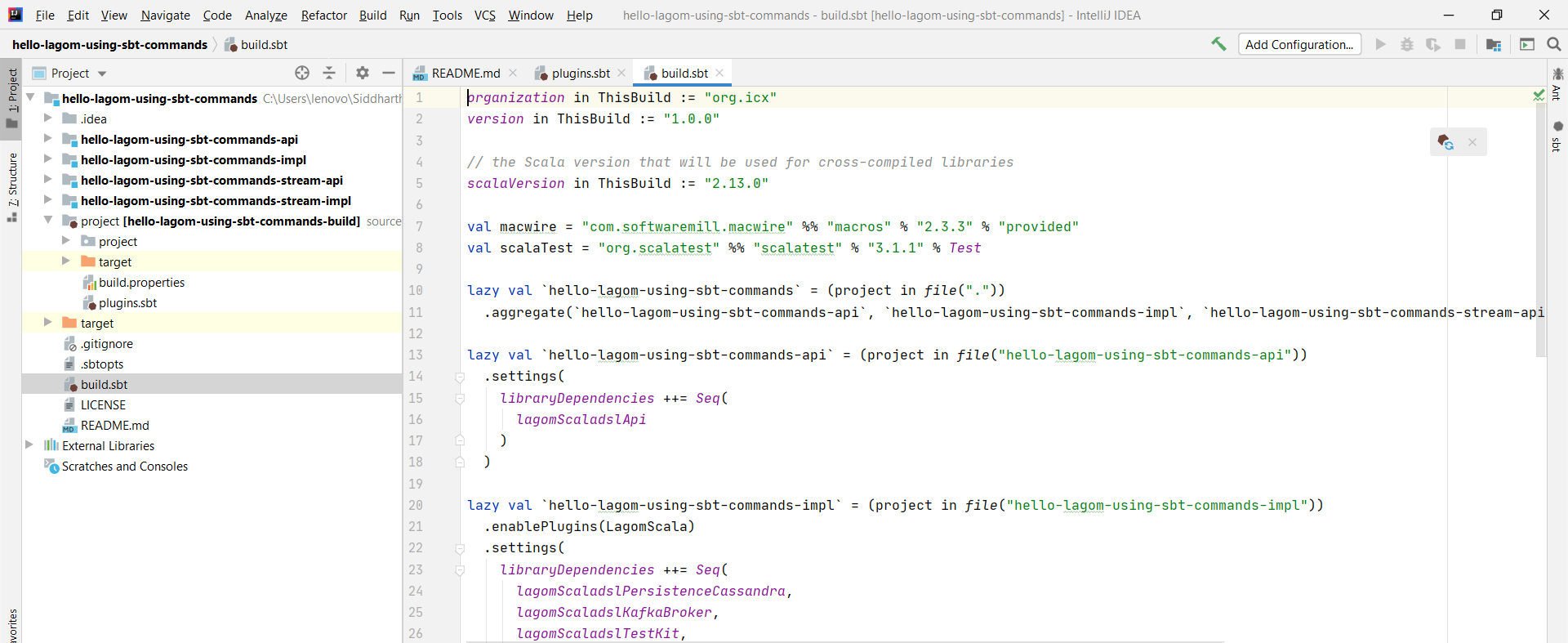
Notice there are two services and their corresponding implementation is already created. It sadly creates the service names using the project name that you provide while creating the project. So, the one used while creating this seems unnecessarily long now. Anyways, just explore the project to check what all things are already there.



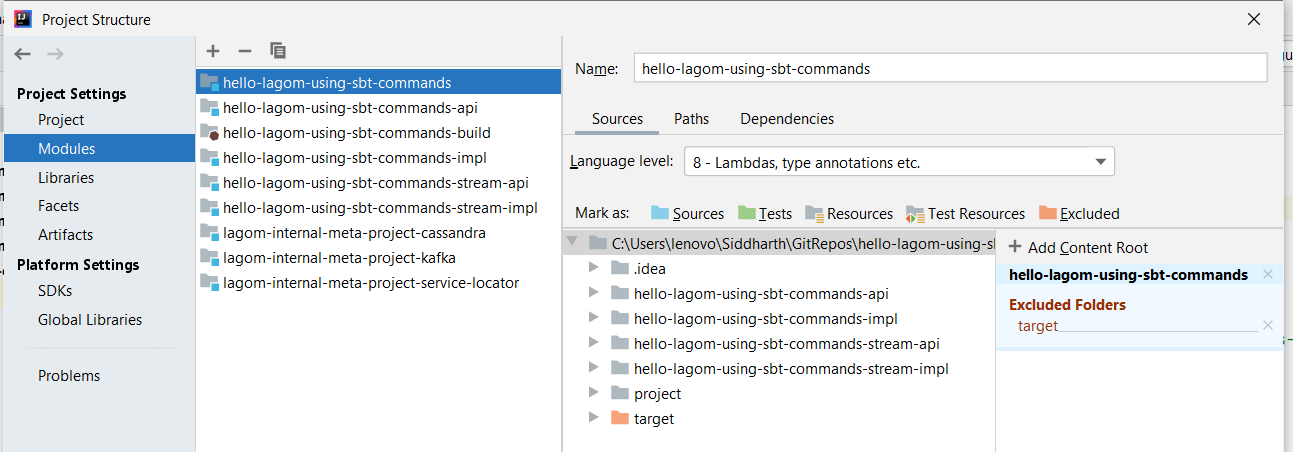
Check the project directory and notice that there’s a .sbt file that adds the lagom plugin. Plugins are the things that extend your base build file build.sbt.



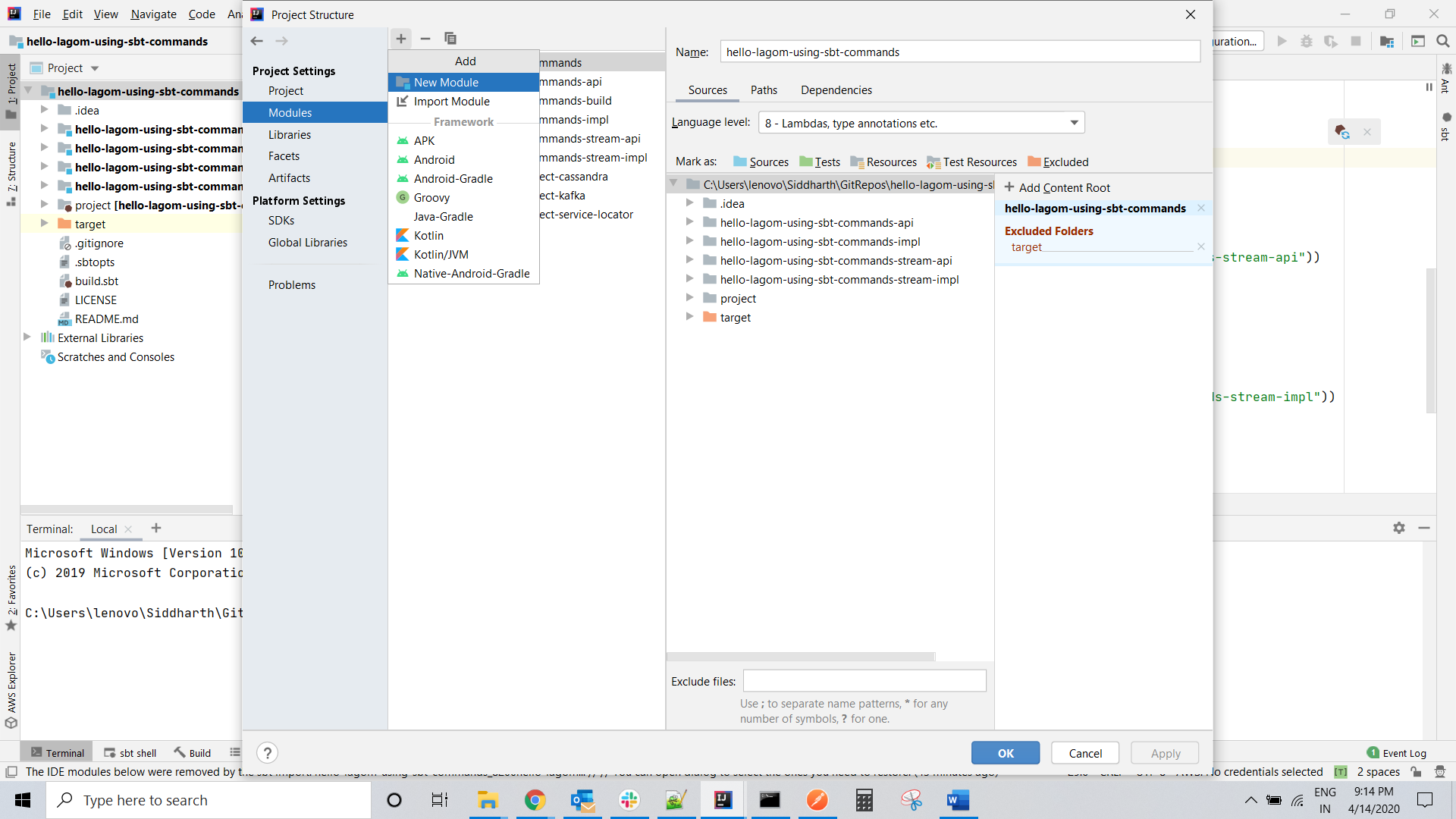
Open and read through the build.sbt file of the project and how it has registered both services. Eventually these will be removed but just to understand, go through them.

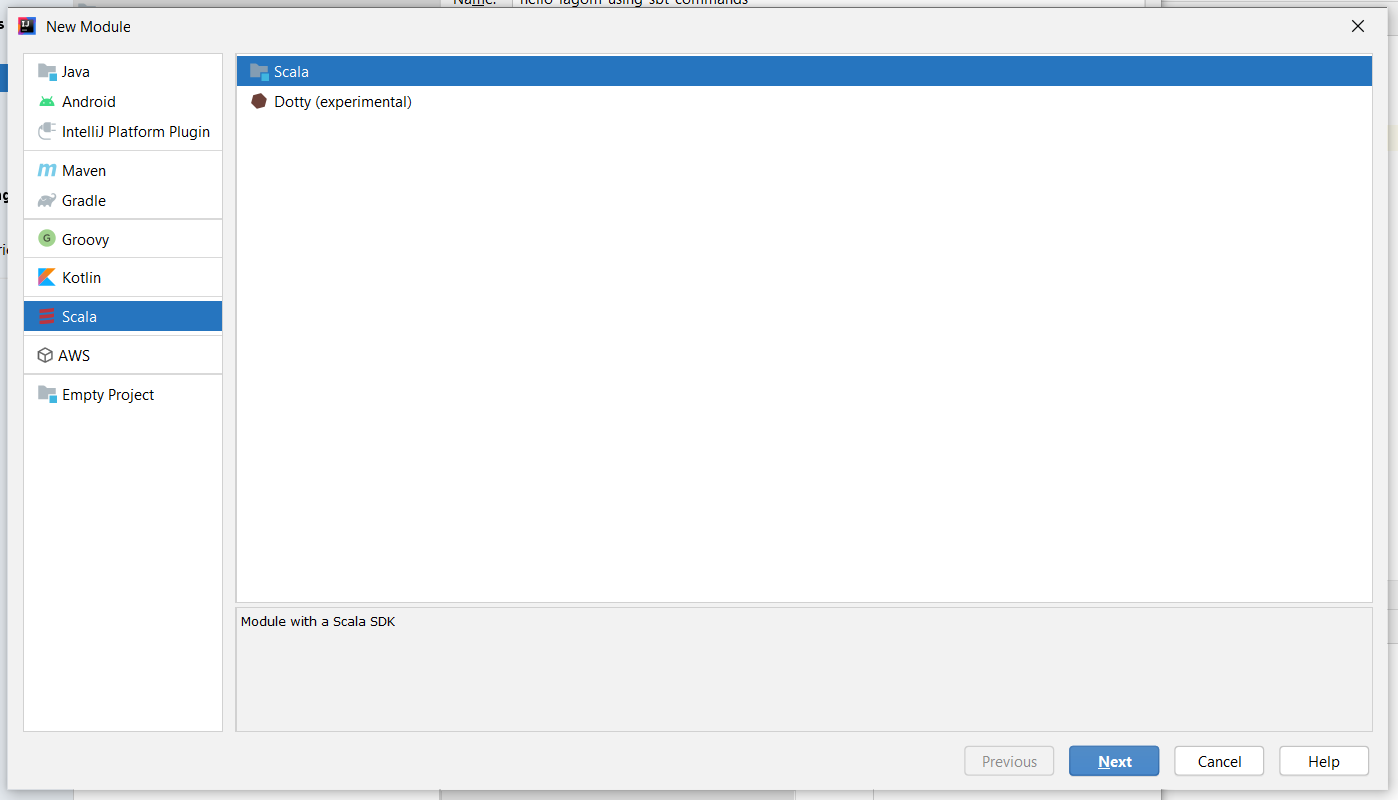


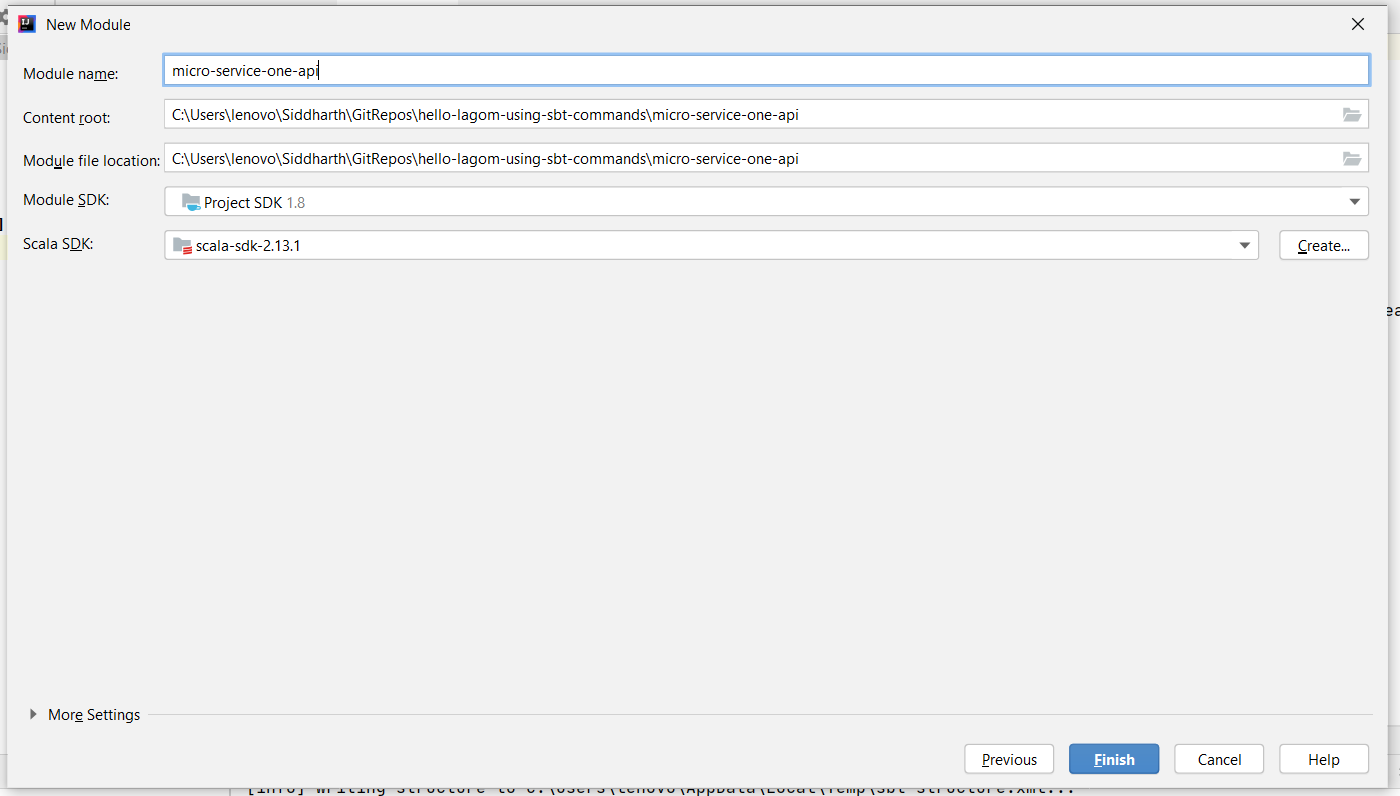
Let’s create a new module for the service we want to create. Right click on the project root in the project explorer and go to module settings.



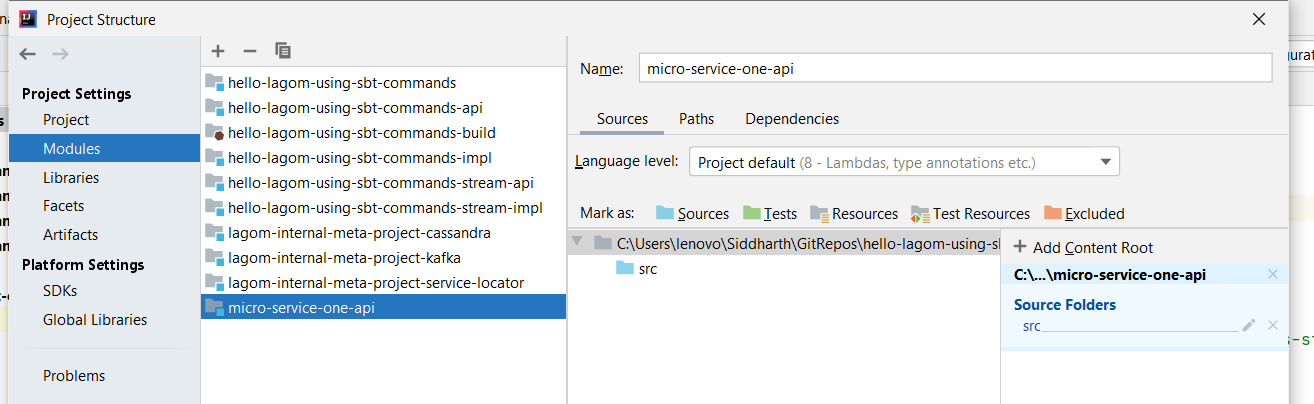
Create a new module of type Scala.



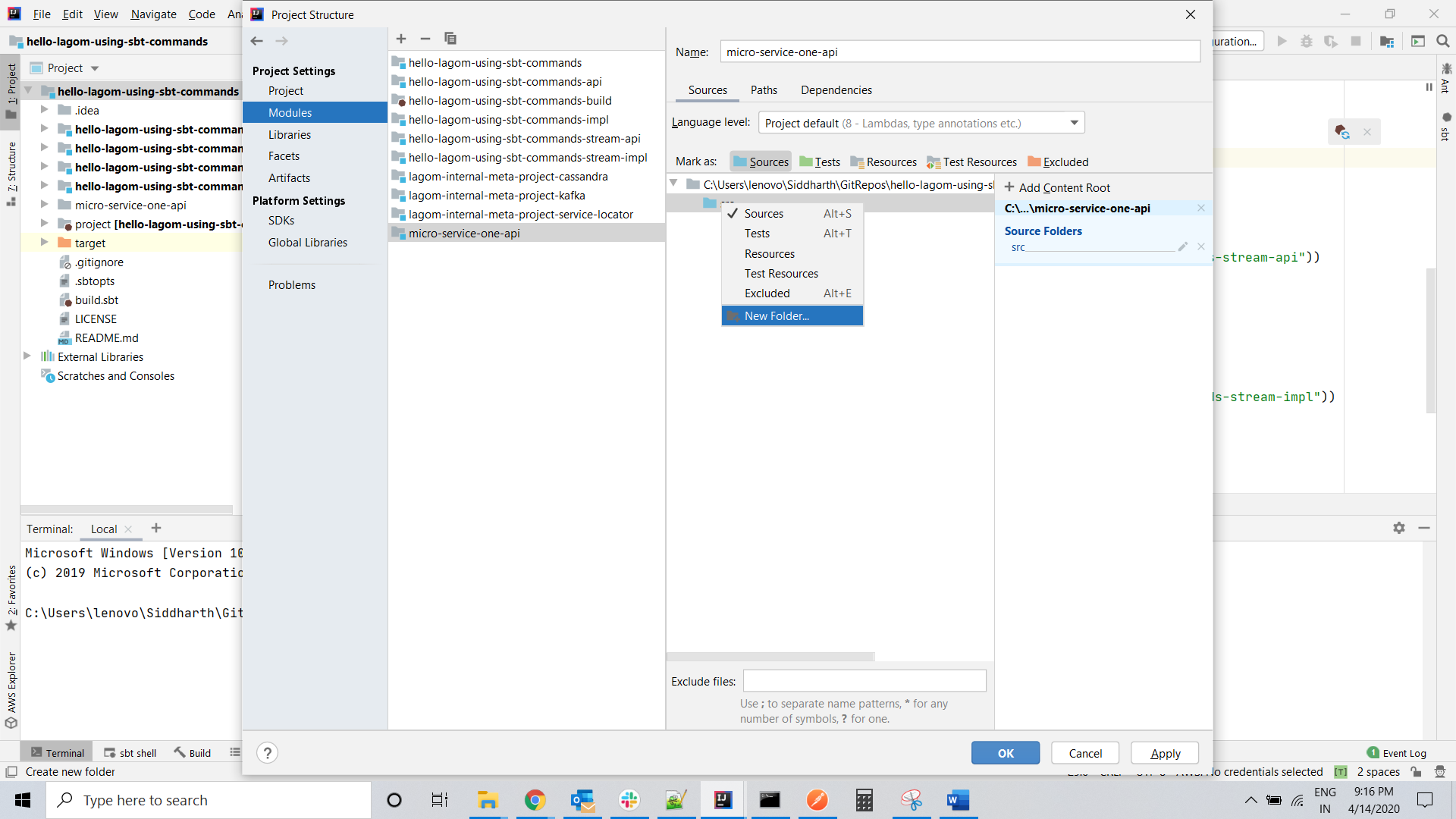


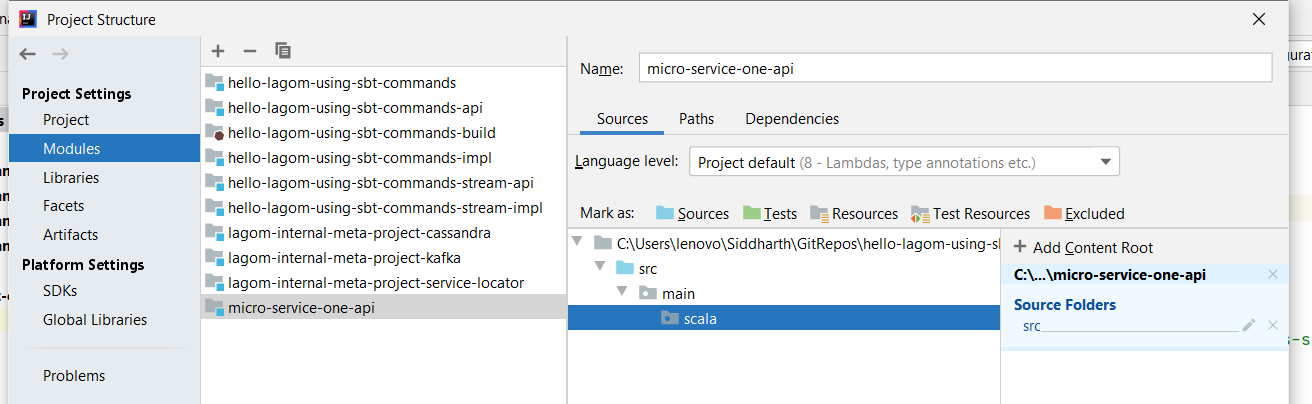
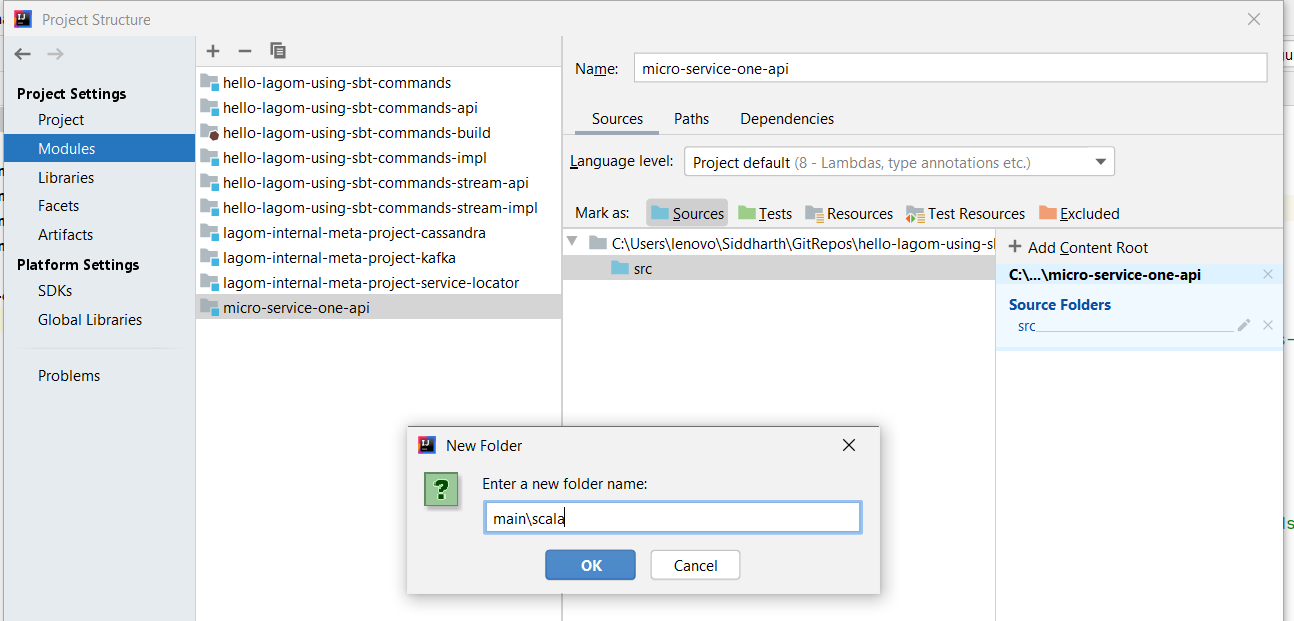


By default Intellij will create a module with a directory called “src” marked as a source folder.

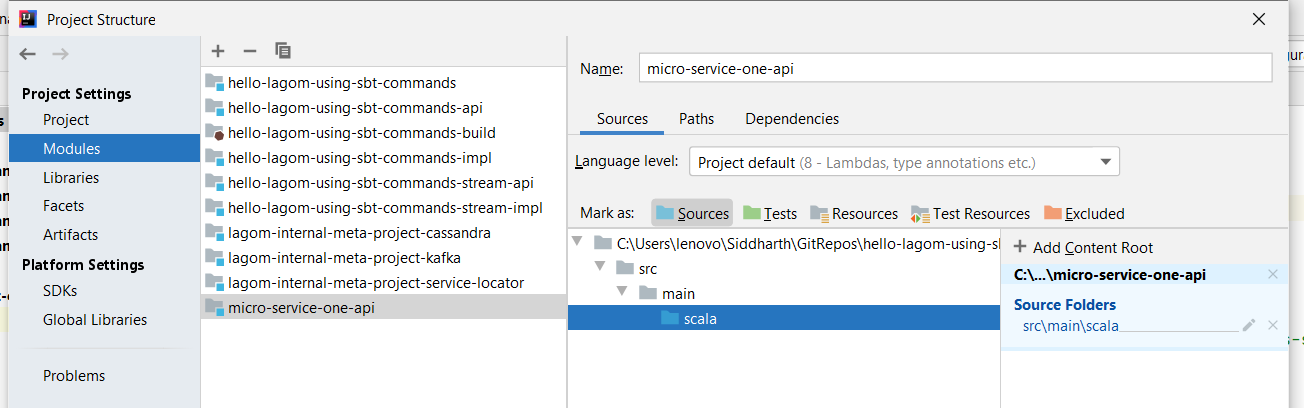


Right click on “src” directory and add two more directories “main/scala” to the hierarchy.

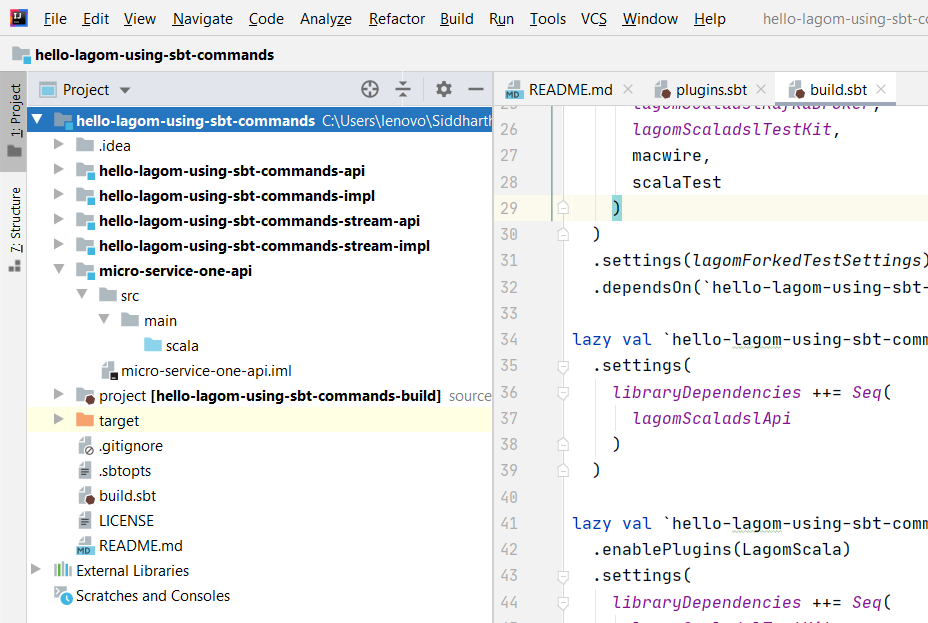




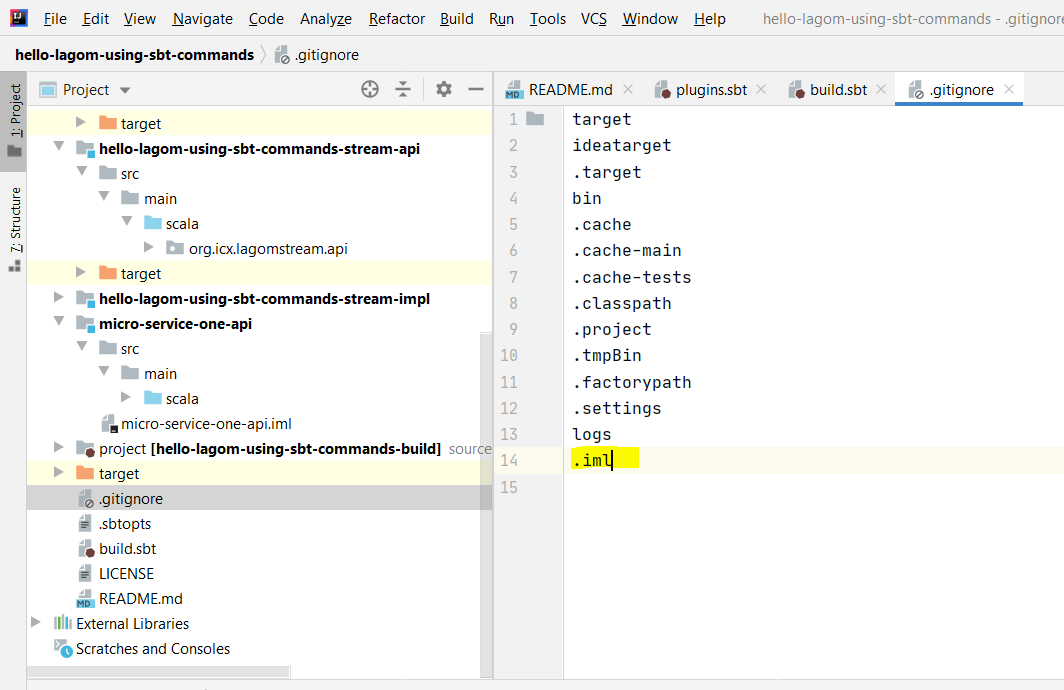
Right click on scala and mark it as source and from the right panel remove the parent “src” from the list of source folders.



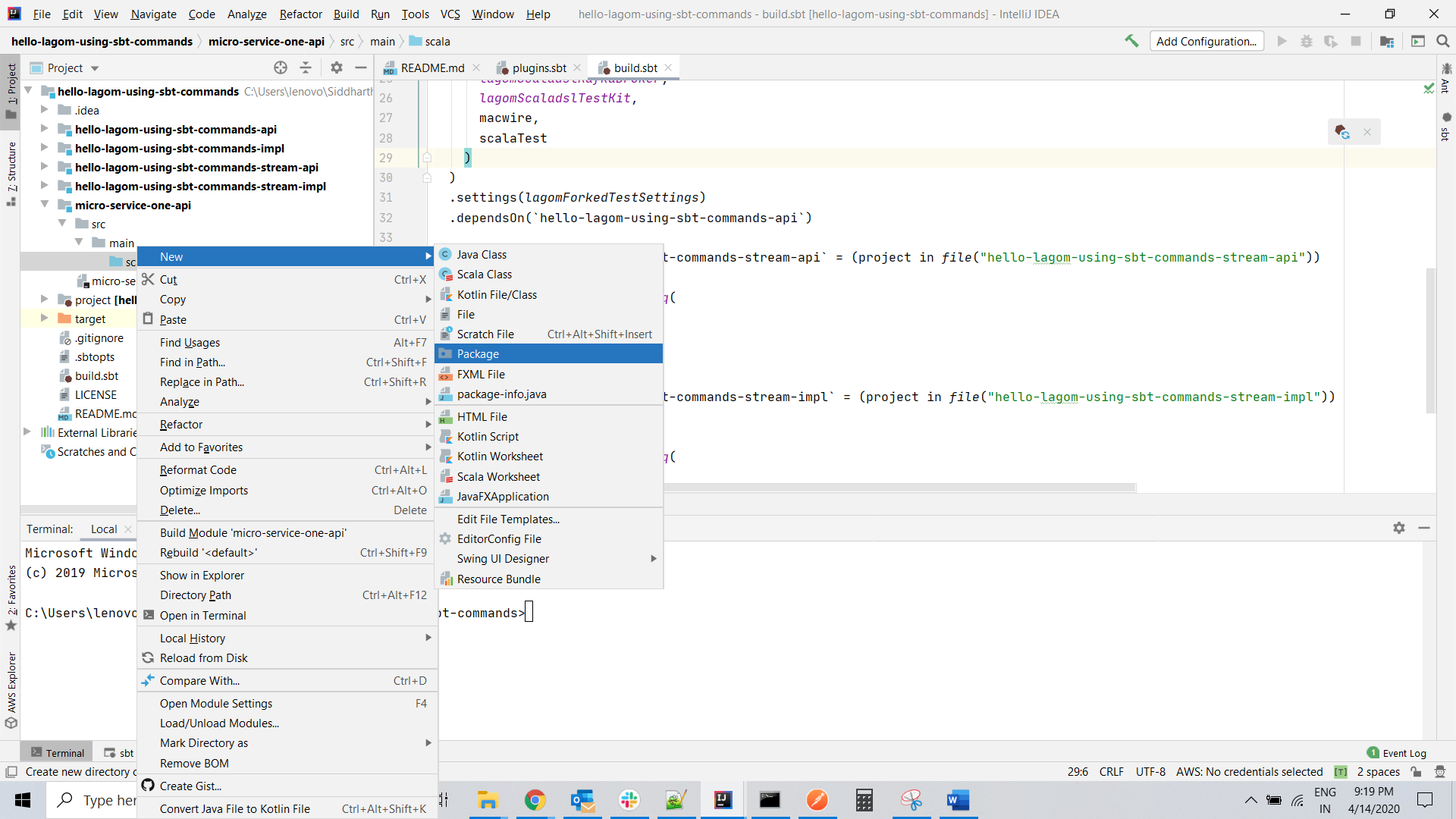
Notice the new module in the project structure.

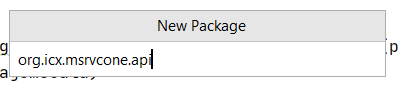


There is a .iml file created. This is created by IntelliJ for it to know that the directory it is in is a module. It contains some description about the module. Hence, it becomes important to add this extension to .gitignore file.

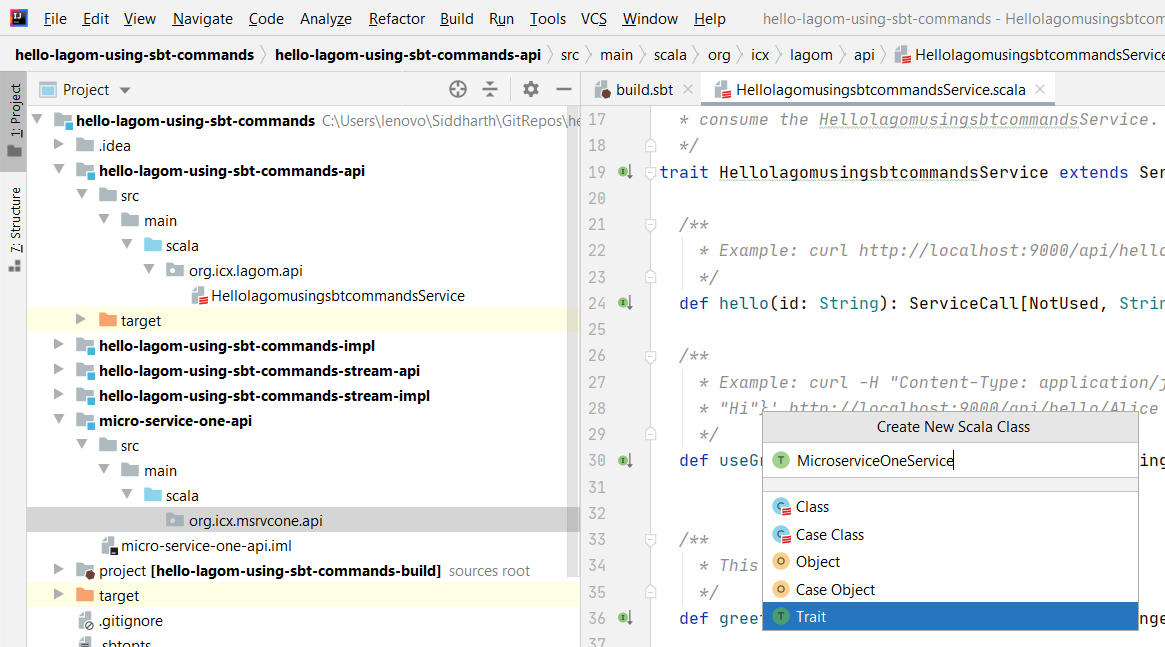


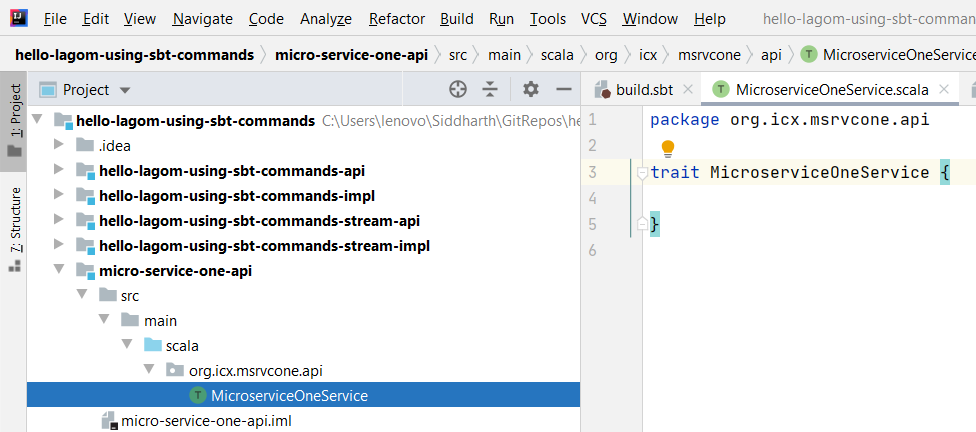
Now create a package to write API.





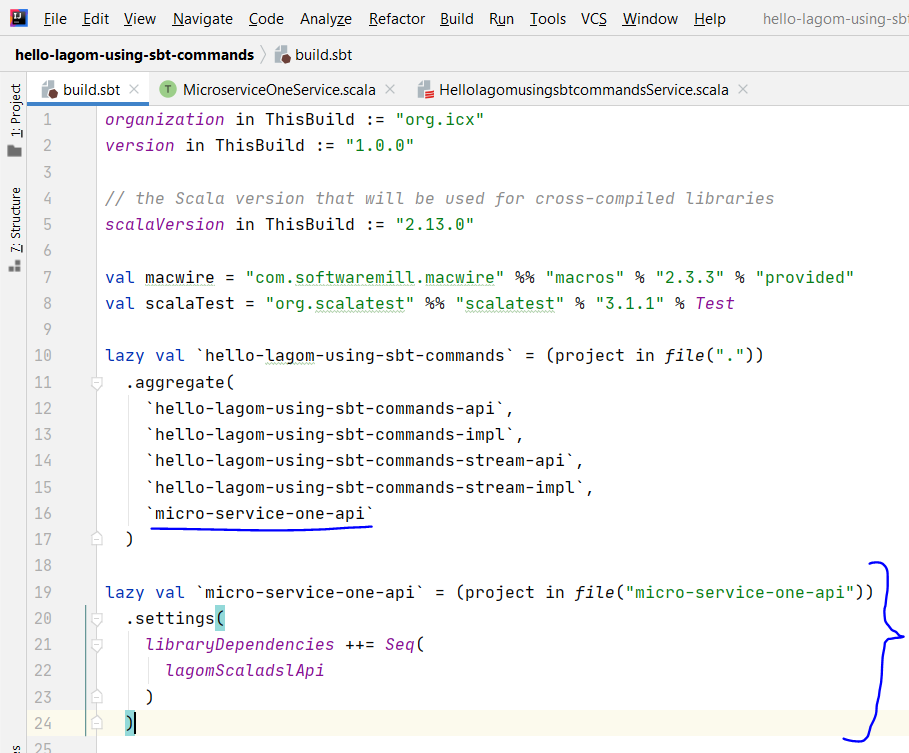
Create a trait for the API. Naming convention generally is that API name ends with …Service and implementation that we would create later ends with …Impl. So, the name **MicroserviceOneService** of trait looks a bit weird for this example as we have named the service as MicroserviceOne.





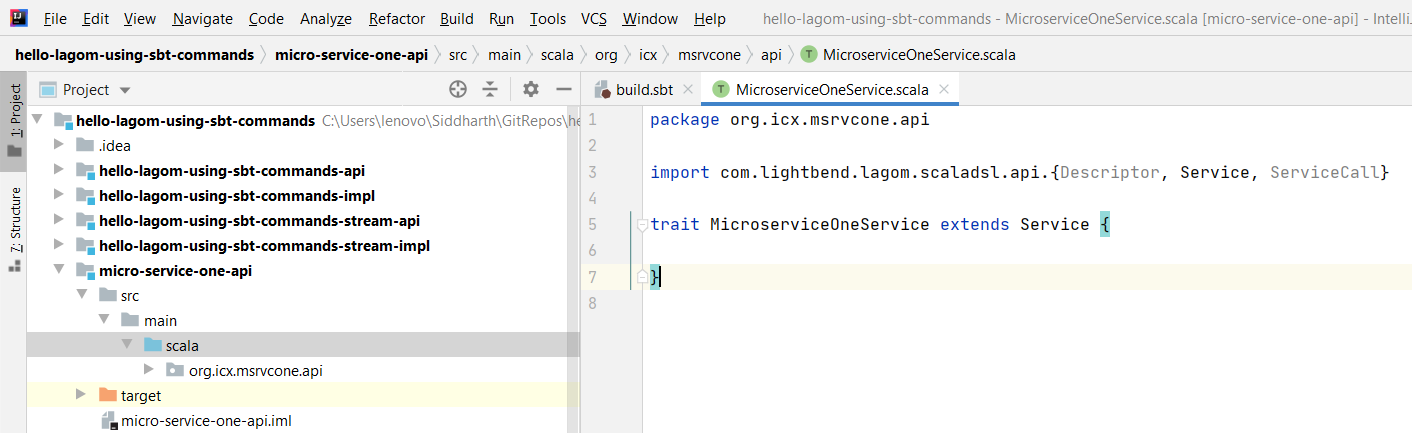
Next thing we want to do is to convert the trait that we have created to a lagom service and the way to do it is to make it extend lagom service. But then, this module has not got access to lagom library. Let’s go ahead and do it.

Adding a library dependency is just a part of this step. What we are doing is registering the service to project’s build. You can copy-paste-edit an existing service. So, one is defining the service including dependency and then add that definition as a module to the project root (the one under .aggregate()).



Refresh SBT once after adding above.

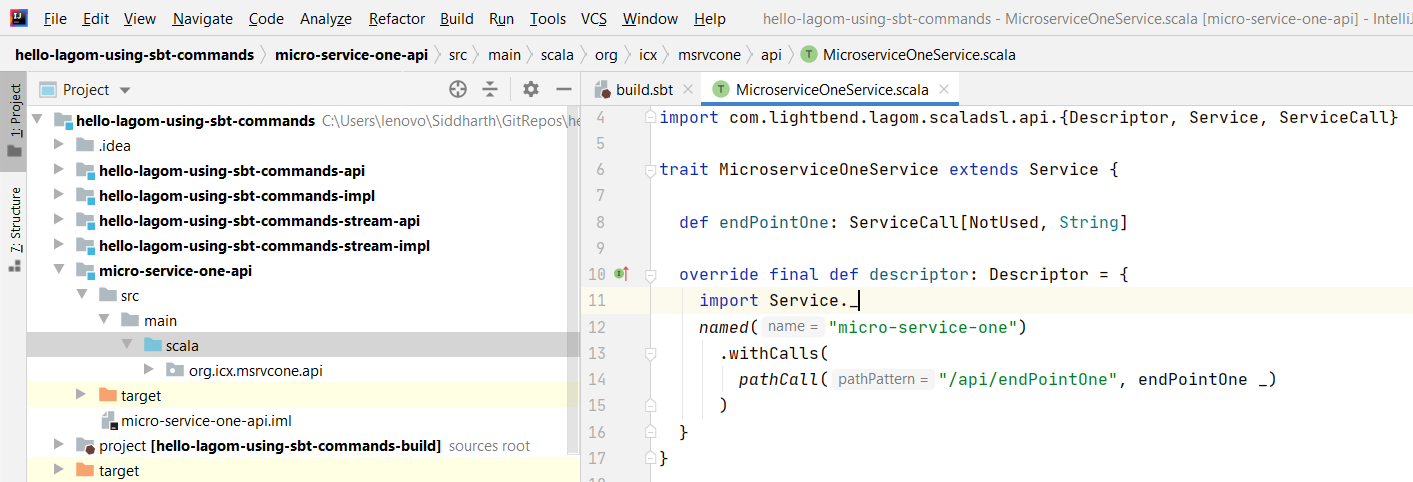
After adding above, you should be able to import scaladsl.api to the trait and extend the Service. You can copy-paste-edit most of the stuff from existing services provided with the template.



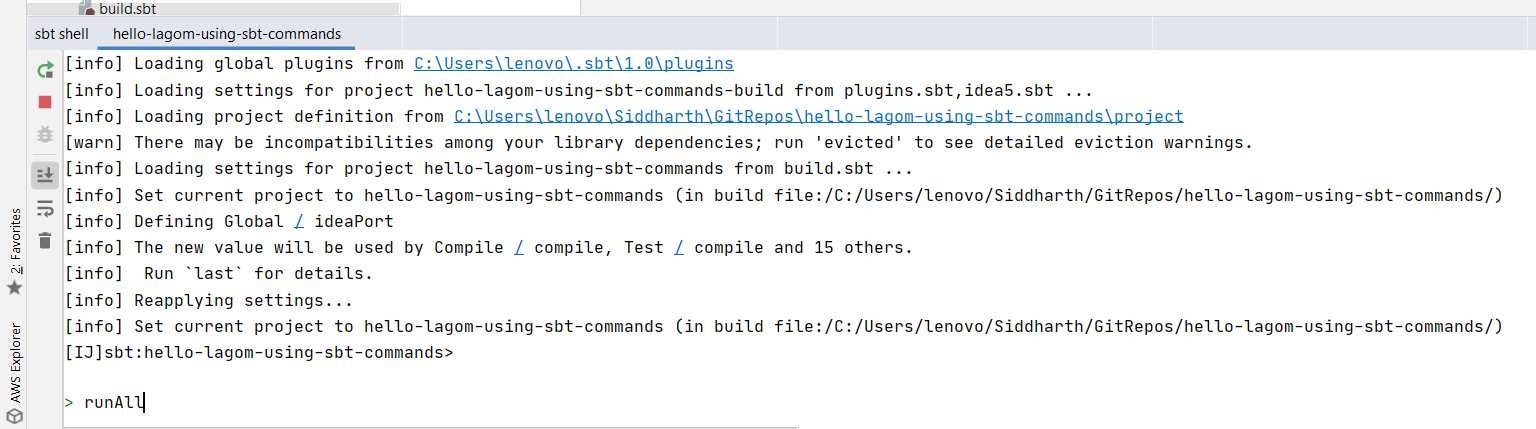
Next thing is to add a method that returns a handler to the service. Let’s name it **endPointOne.** Then add a service descriptor pointing to that method.

Notice that our method doesn’t take any input. So, an underscore towards the end of line 14 in the snapshot below is not required.

def endPointOne: ServiceCall[NotUsed, String]  
  
override final def descriptor: Descriptor = {  
 import Service.\_  
 *named*("micro-service-one")  
 .withCalls(  
 *pathCall*("/api/endPointOne", endPointOne)  
 )  
}



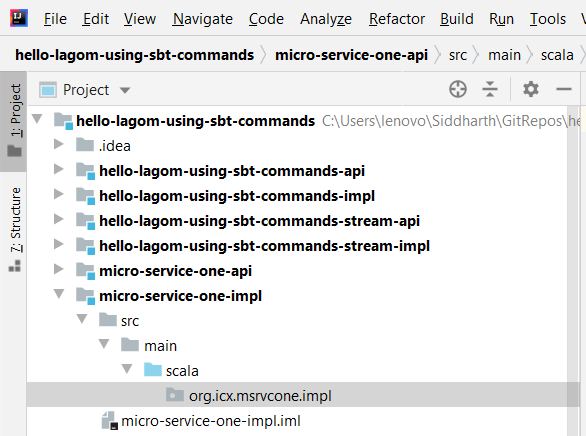
Now writing API is complete. Next is to add implementation. But let’s try to run the service without writing implementation.

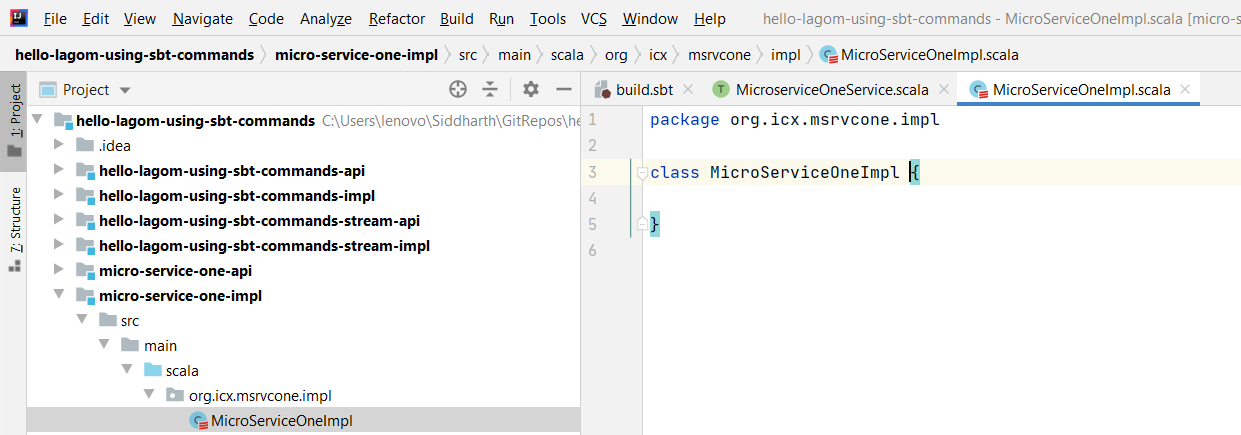


It doesn’t show up on the service descriptor.

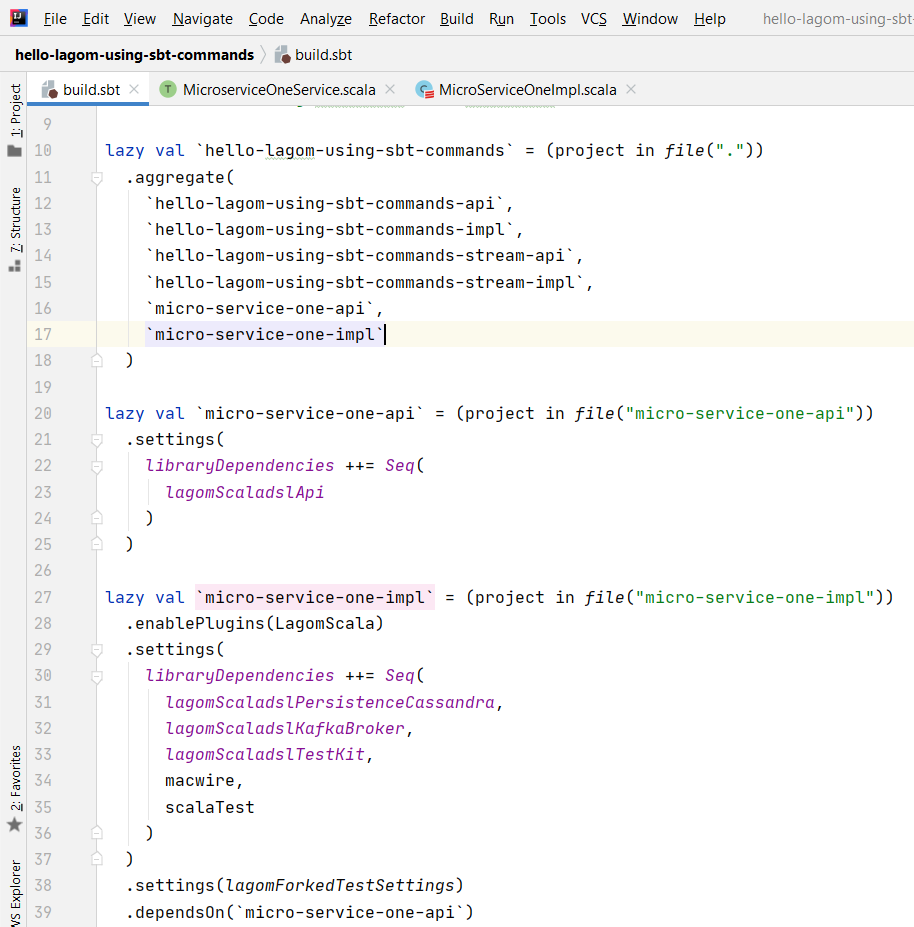


So, let’s create a module for the service implementation in the same way we did for service api. And then create a class for the service implementation.

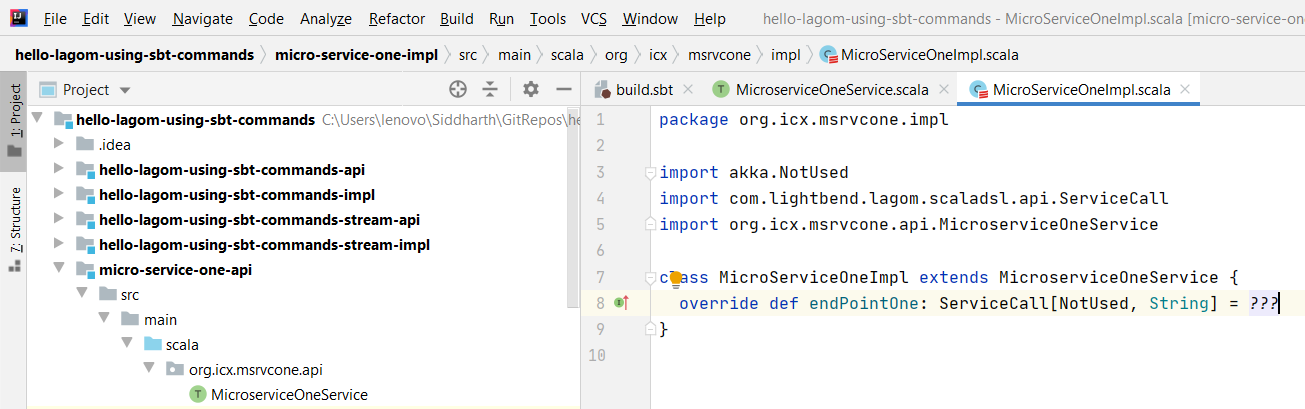




Add this to build definition and refresh sbt.



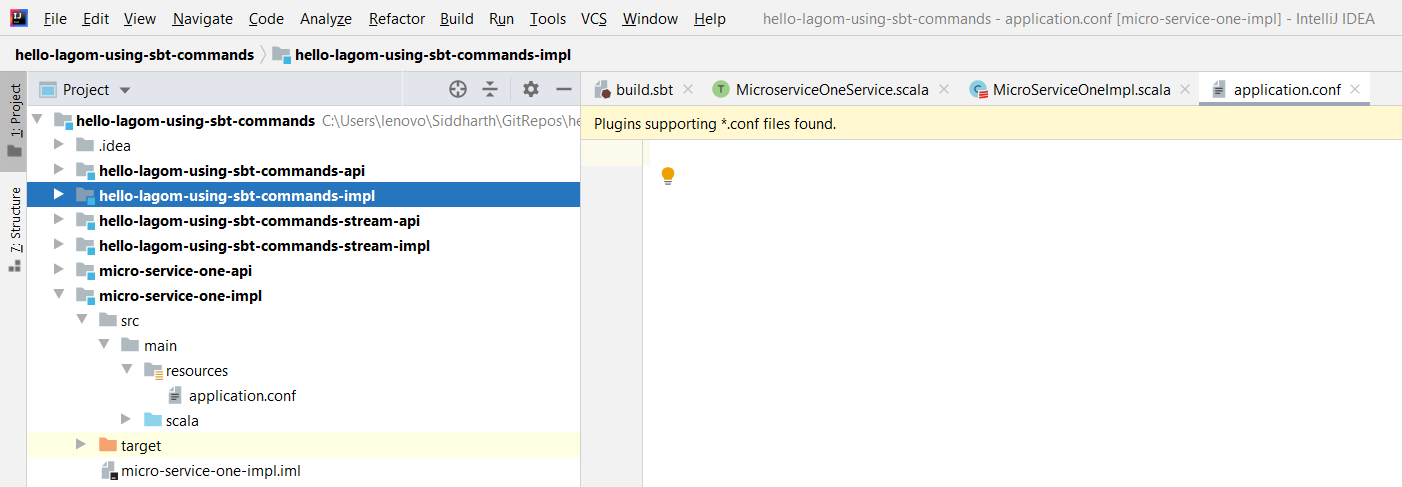
Now we can write the implementation. This class extends the API that was written earlier and hence must implement the method that was declared in the API.



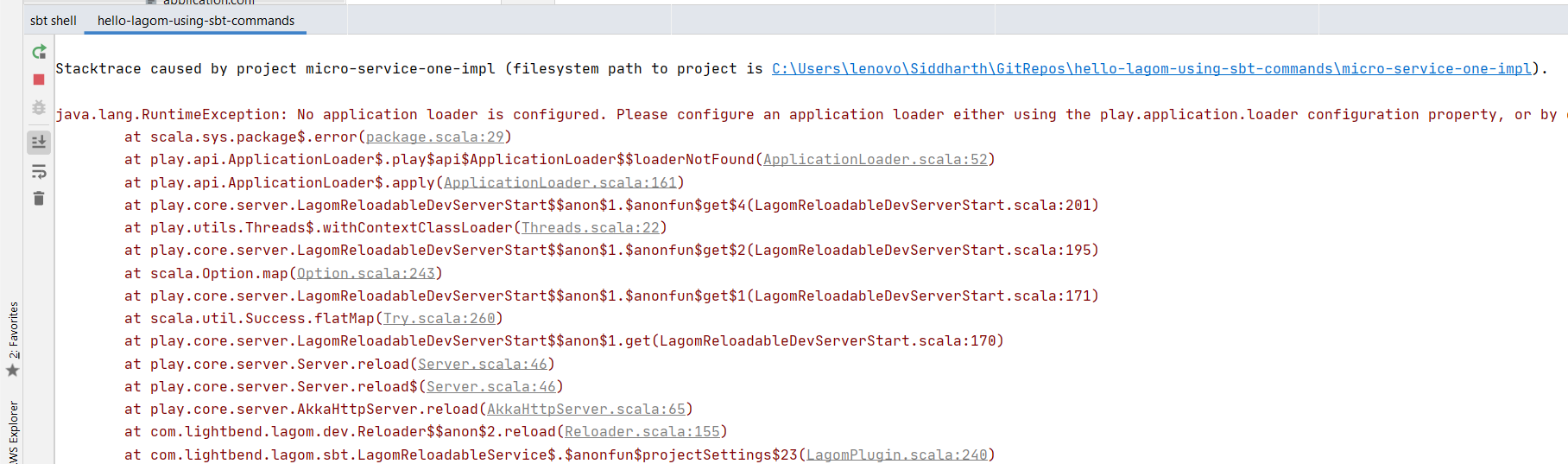
Let’s try run this with minimal code written above.



It requires an application.conf file. So, let’s add a resource directory to the implementation class and add an application.conf.



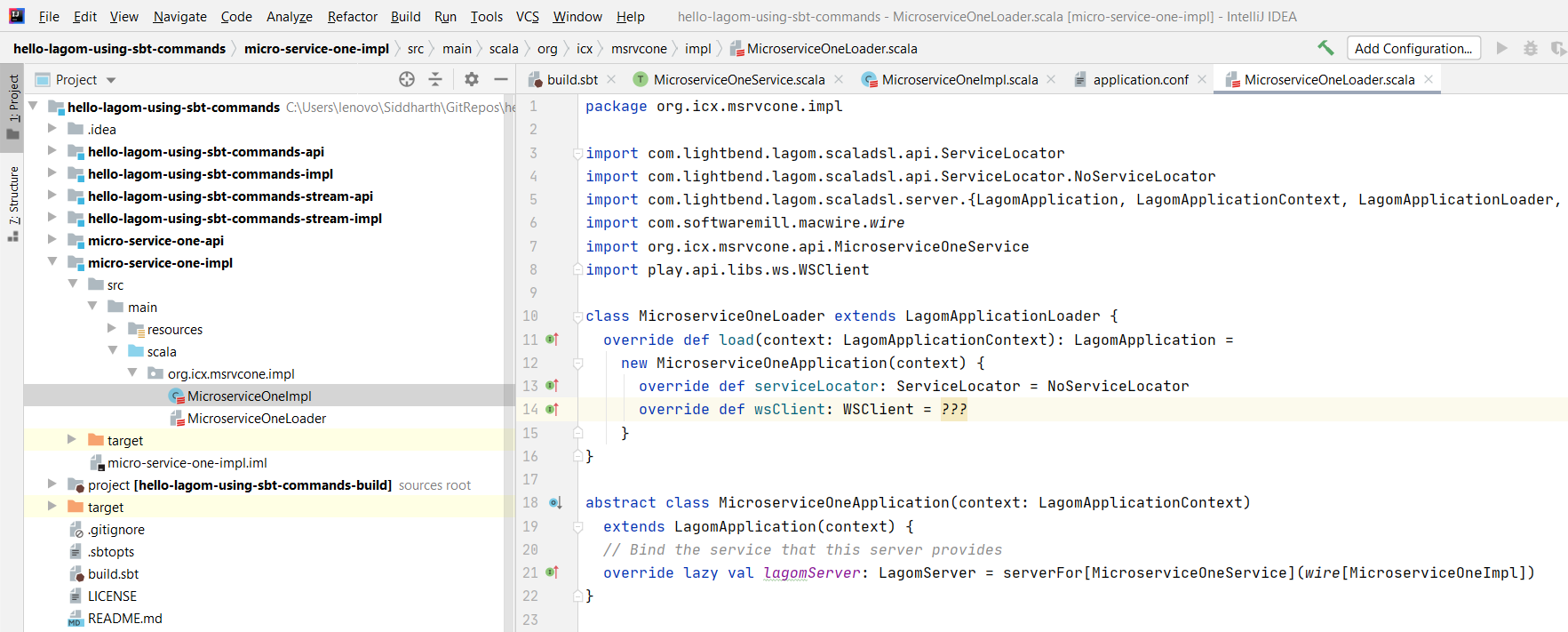
Run now.



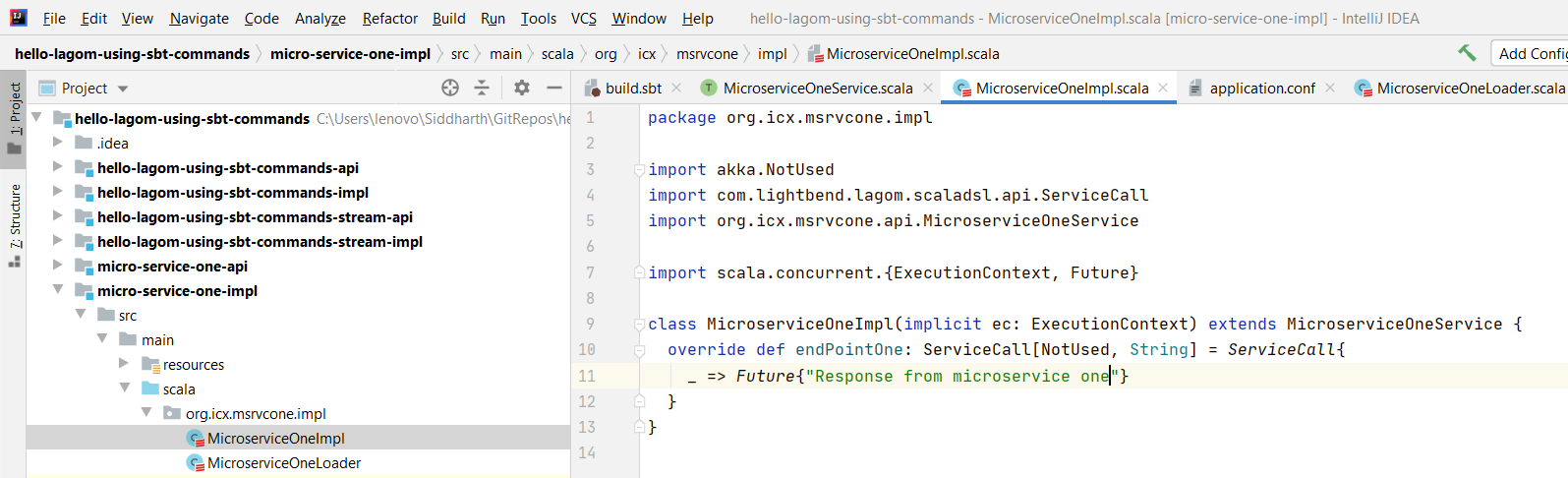
New error. No application loader found.

java.lang.RuntimeException: No application loader is configured. Please configure an application loader either using the play.application.loader configuration property, or by depending on a module that configures one. You can add the Guice support module by adding "libraryDependencies += guice" to your build.sbt.

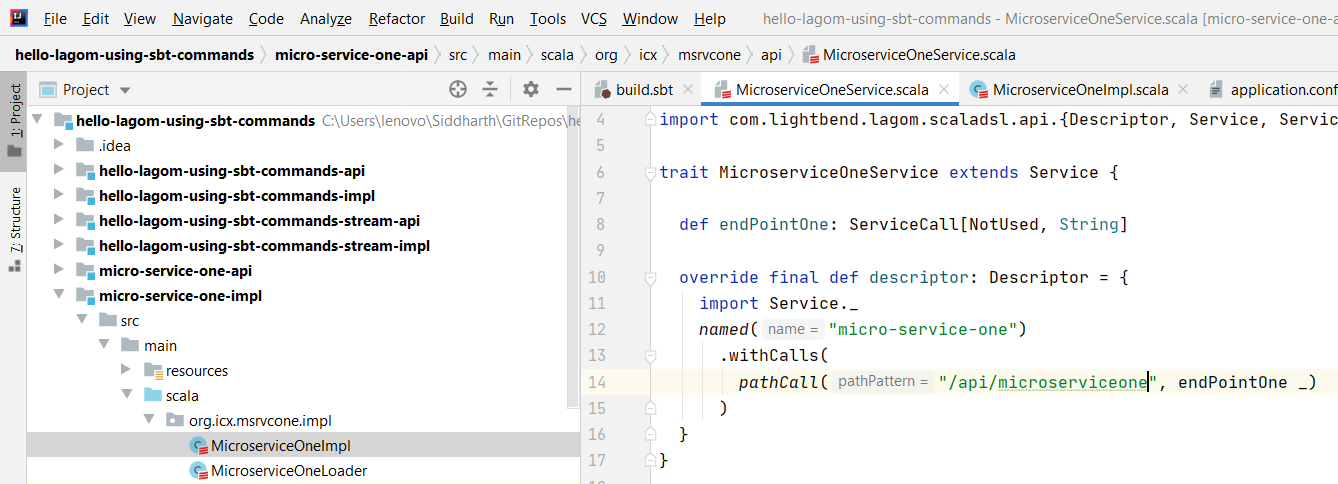
Let’s add one. Again copy-paste the minimum things from existing services. I picked below things. Not sure what is making it require statement at line 14. It is not there in existing service.



Also, let’s implement the service too. It returns a Service call which takes a request and returns a Future of response type. Future[String] in this case.

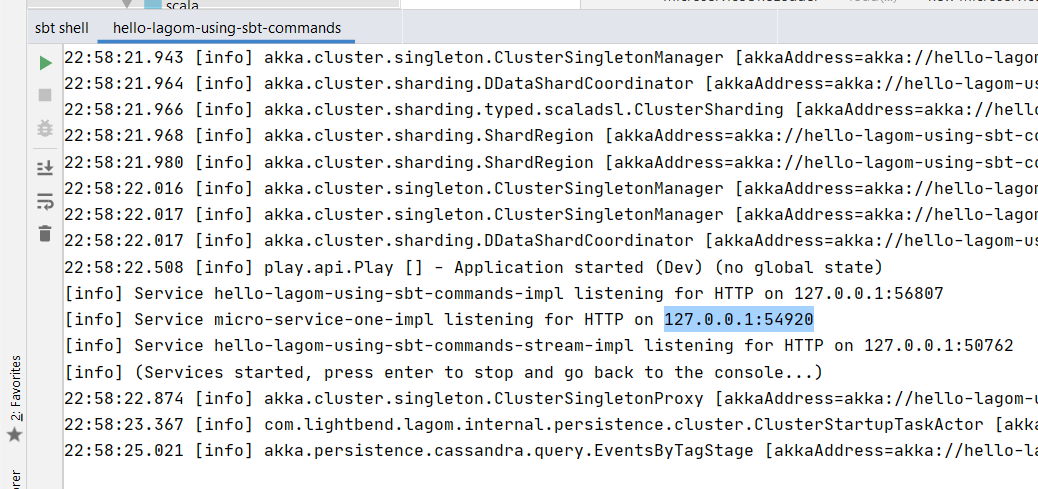


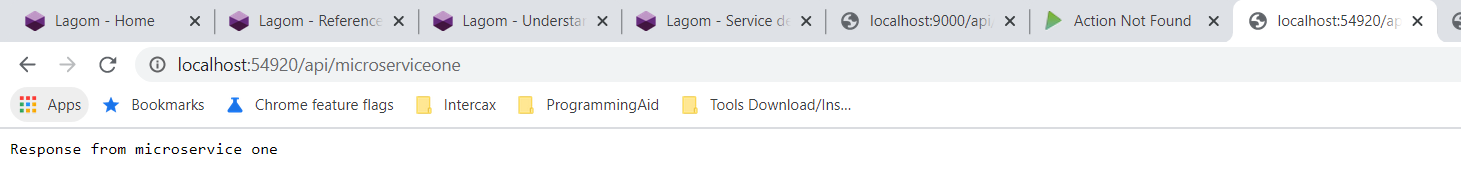
Also, let’s change the end point a bit.



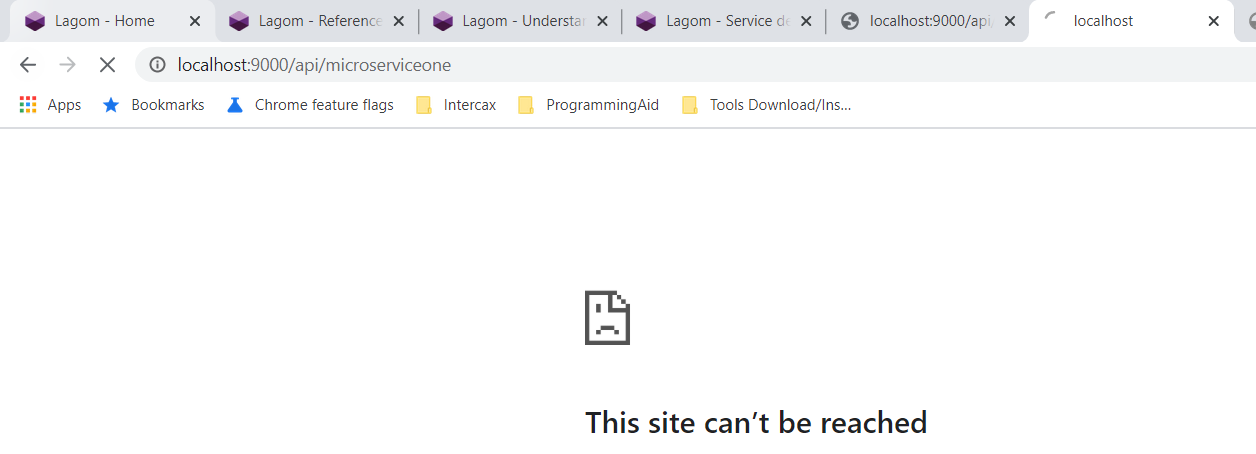
Let’s run it now.

So, service is up at port 54920.

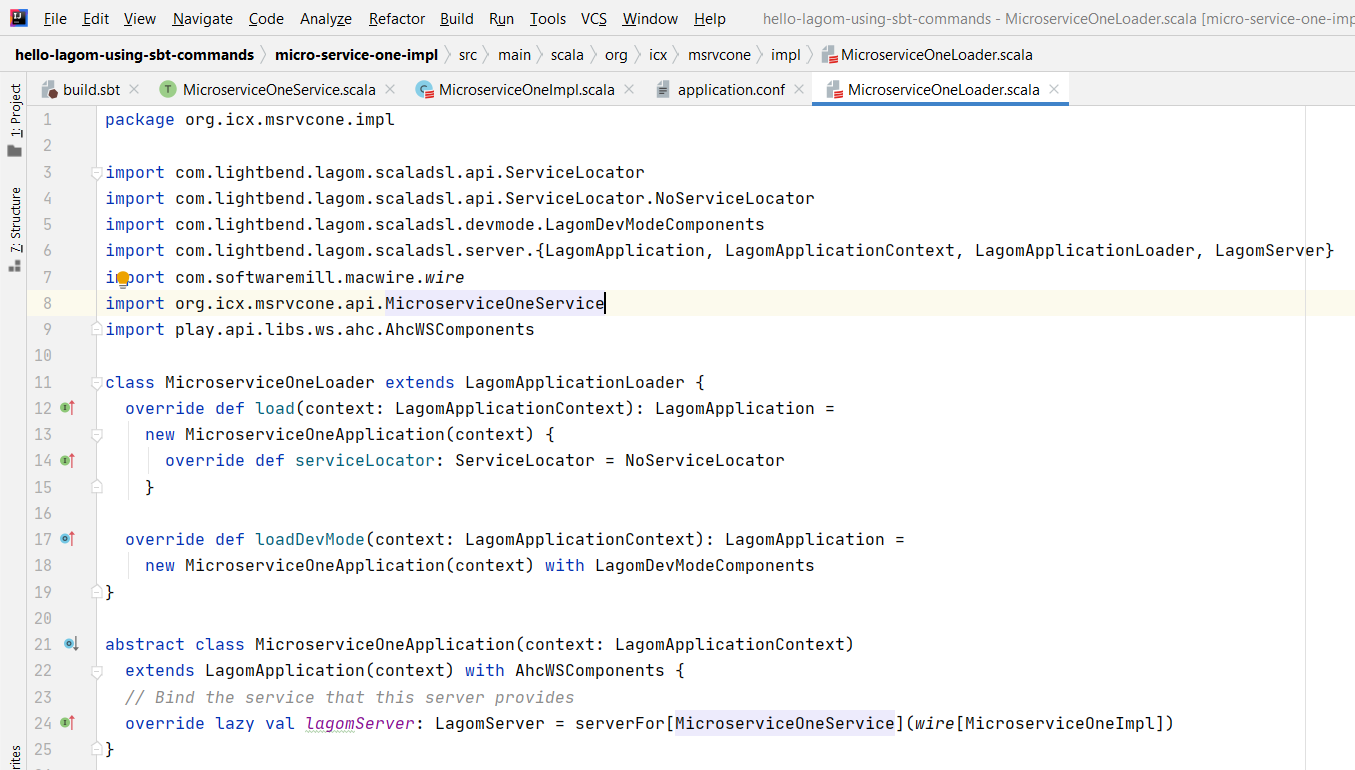




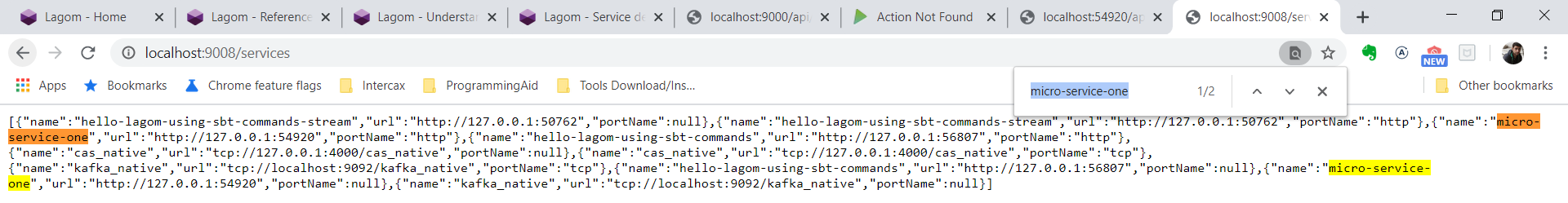
Let’s try it via the service gateway too. Nope, not showing there.



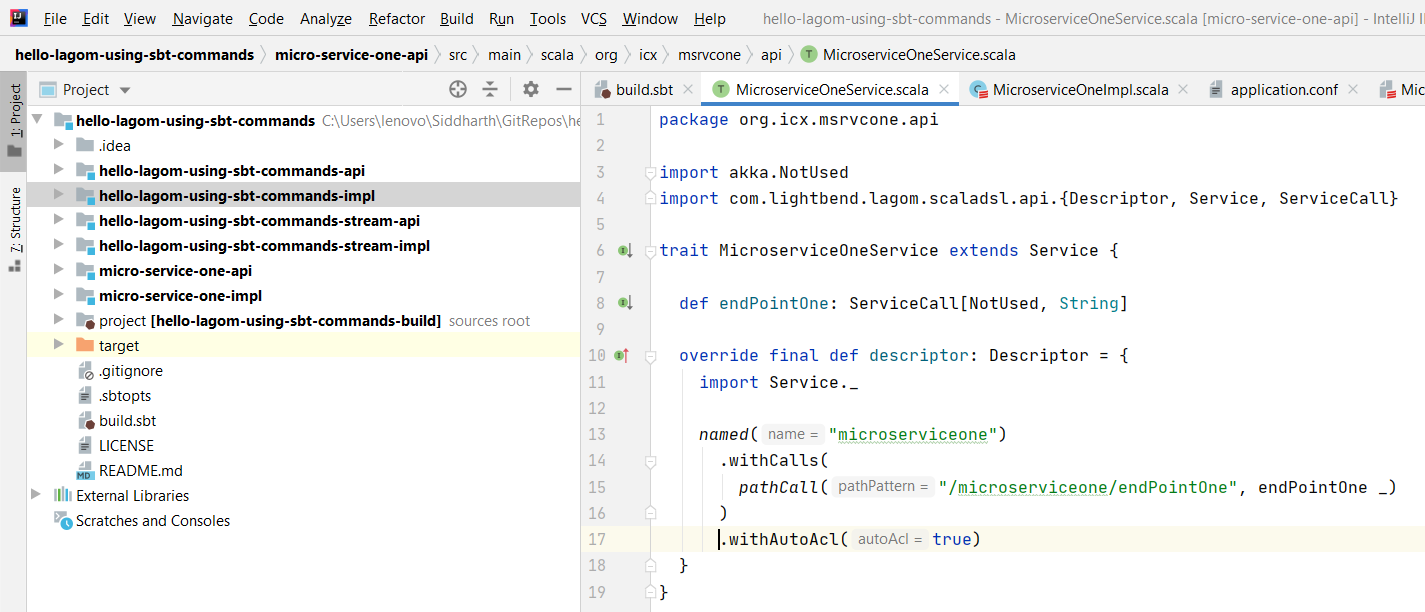
Let’s try adding the dev mode to the loader. And, adding AhcWSComponents at line 22 below made that wsClient thing go away.



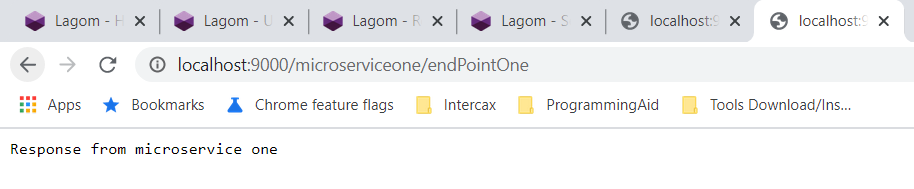
It now shows up on the service descriptor at least.



But service is still not accessible via service gateway. Let’s add one more thing that we avoided earlier. Line 17.



There you go. It is accessible via gateway too.



Lastly, delete the services provided by the template. Take the ownership of the project.



