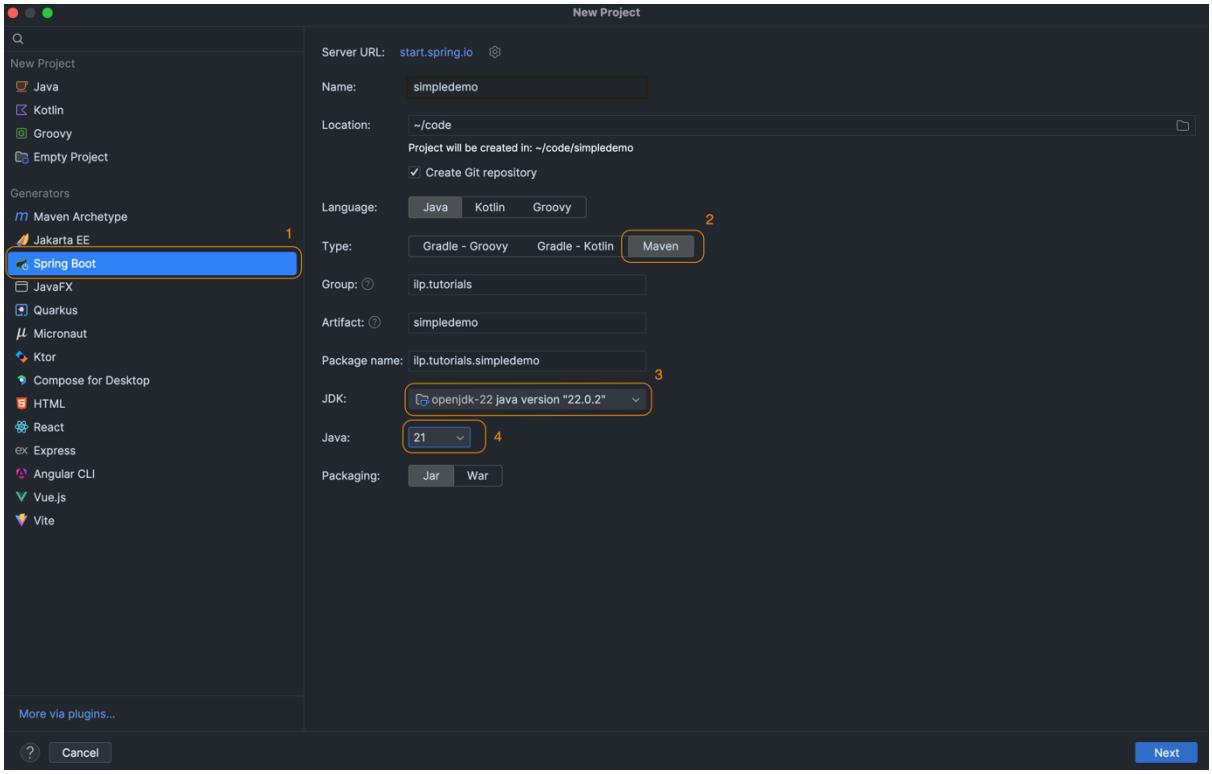
Tutorial 1 / ILP

Here we are going to walk through the very basics of setting up a REST service using Spring Boot within IntelliJ. We will then be packaging it into a Docker image and running the container. Once that is done, we will send HTTP requests to the server using CURL and Postman. If you have never used either Spring Boot or Docker before, this walk-through should give you an idea of the workflow! If there are any questions about this document, please ask on Piazza, we will try to answer as quick as we can!

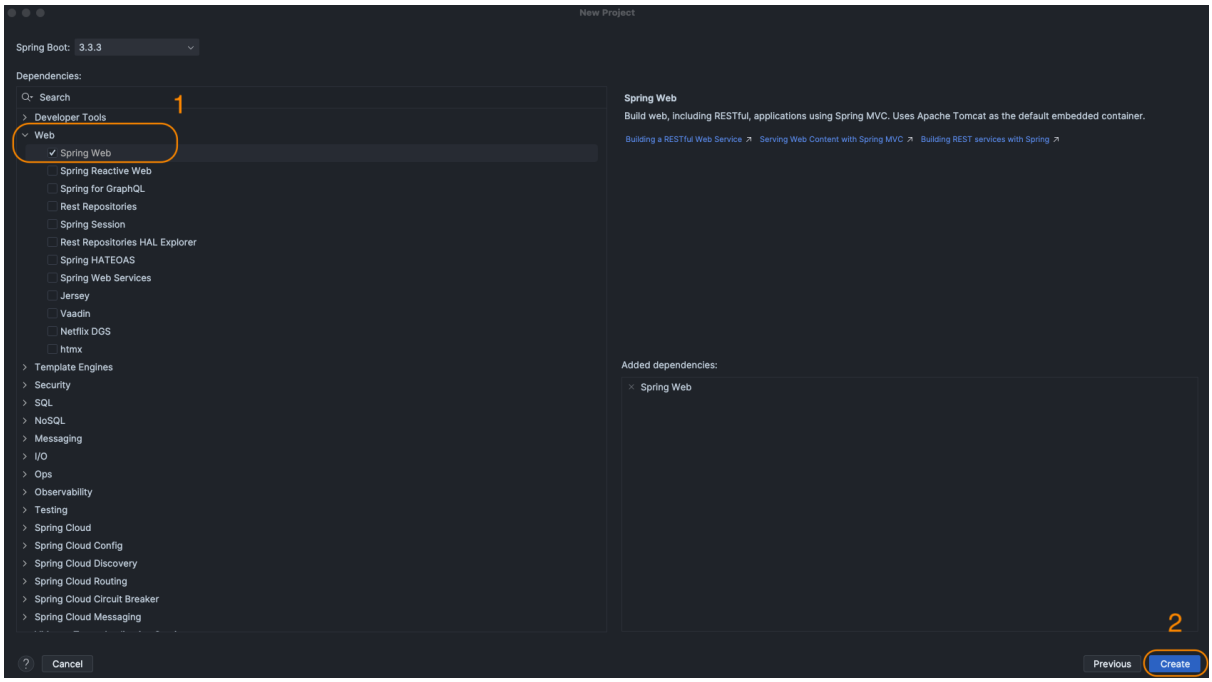
IntelliJ is recommended for the course. If you have not installed IntelliJ yet use the links below to take you to the right place, then follow the download instructions from there! As you are a student, you can also get JetBrains Ultimate for free.

- https://www.jetbrains.com/idea/download/?section=mac for Mac
- https://www.jetbrains.com/idea/download/?section=windows for Windows
- https://www.jetbrains.com/idea/download/?section=linux for Linux
- https://www.jetbrains.com/community/education/#students JetBrains Ultimate

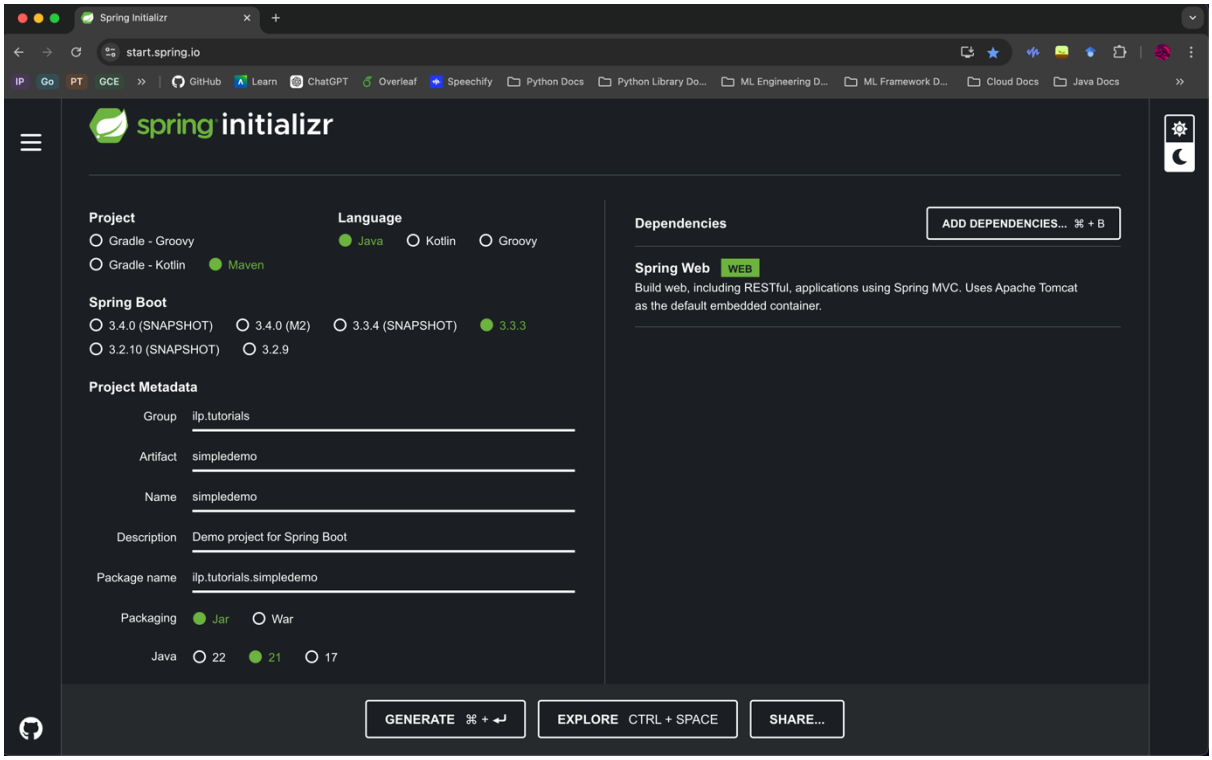
Once downloaded and through the introductions, we can start building our simple Representational State Transfer (REST) service. The framework we are going to use is Spring Boot (https://spring.io/projects/spring-boot). We can directly set up our project within IntelliJ with JetBrains Ultimate. However, other websites like Spring Initializr (https://start.spring.io/) also work! Figure 1 and 2 show the steps for IntelliJ. Whereas Figure 3 shows how to do it with Spring Initializr in case you do not want to get JetBrains Ultimate. Either way, its a simple setup!



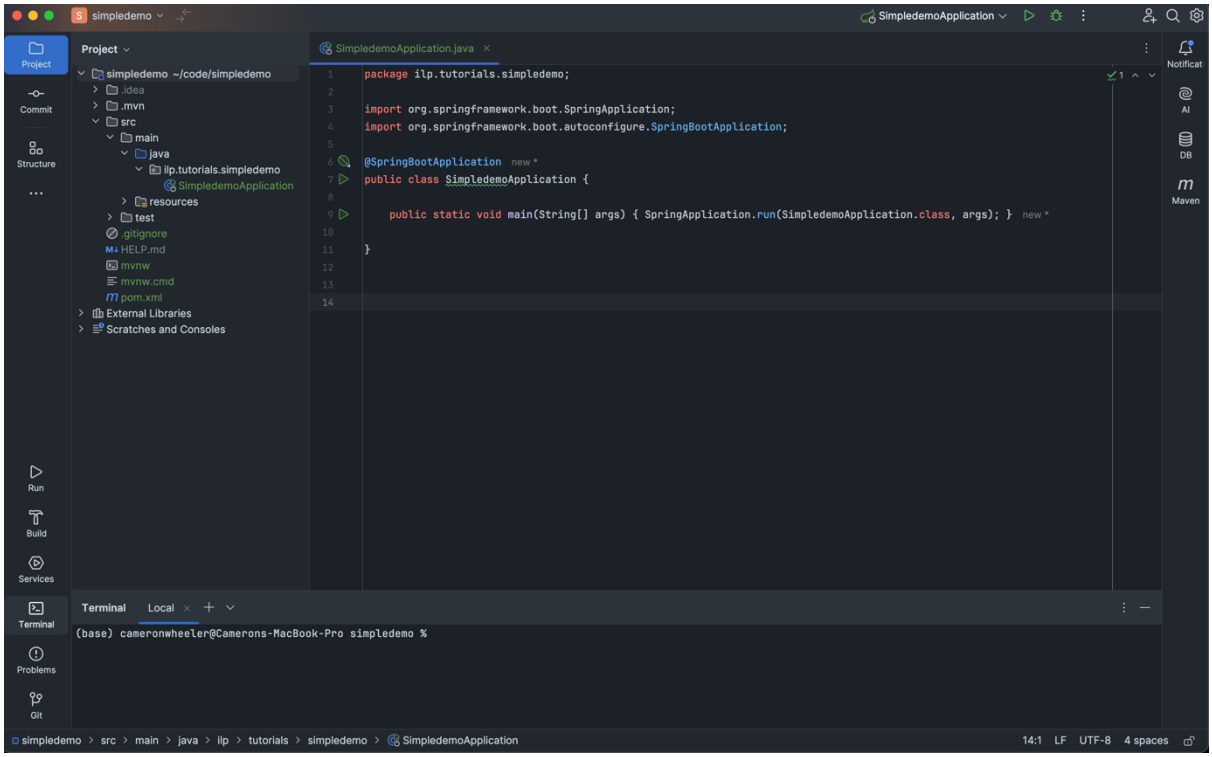
Spring Boot Initialisation: When opening a new project on IntelliJ, we can use Spring Boot to set up a lot of the boilerplate code within the our basic project. Step 1: Select Spring Boot. Step 2: Choose Maven as our build and dependency manager. Step 3: Select JDK (Download one if you do not have one installed, you can also point IntelliJ to a JDK you already have installed). Step 4: Select Java version. Note: If you do not have JetBrains Ultimate, Figure 3 shows you how to use Spring Initializr.



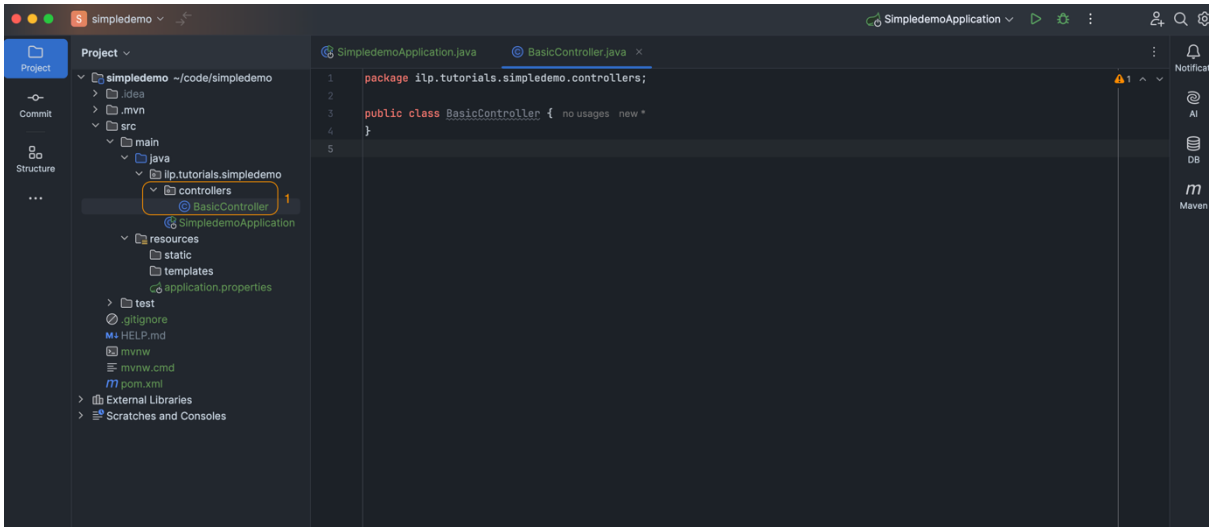
Spring Web Dependency: We need to add Spring Web to our dependencies to create RESTful APIs. Step 1: Tick Spring Web. Step 2: Press create, this will initialise the project and generate the boilerplate code and other files.



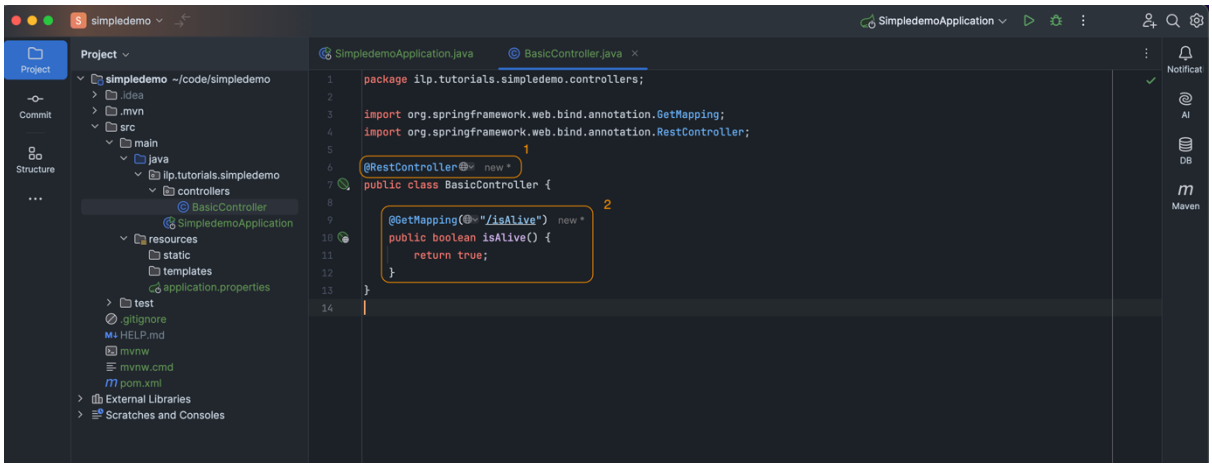
Spring Initializr Setup: If you do not want to use JetBrains Ultimate then you need to use Spring Initializr. Use the configuration on shown above then open up the unzipped file in IntelliJ, it should lead to the same spot! **Skip this if you have already done the steps in IntelliJ!**



Starting Spring Boot Application: The main method and directory structure are auto-generated by IntelliJ. This saves us a bit of time and effort having to add all of this ourselves and makes it easily reproducible!



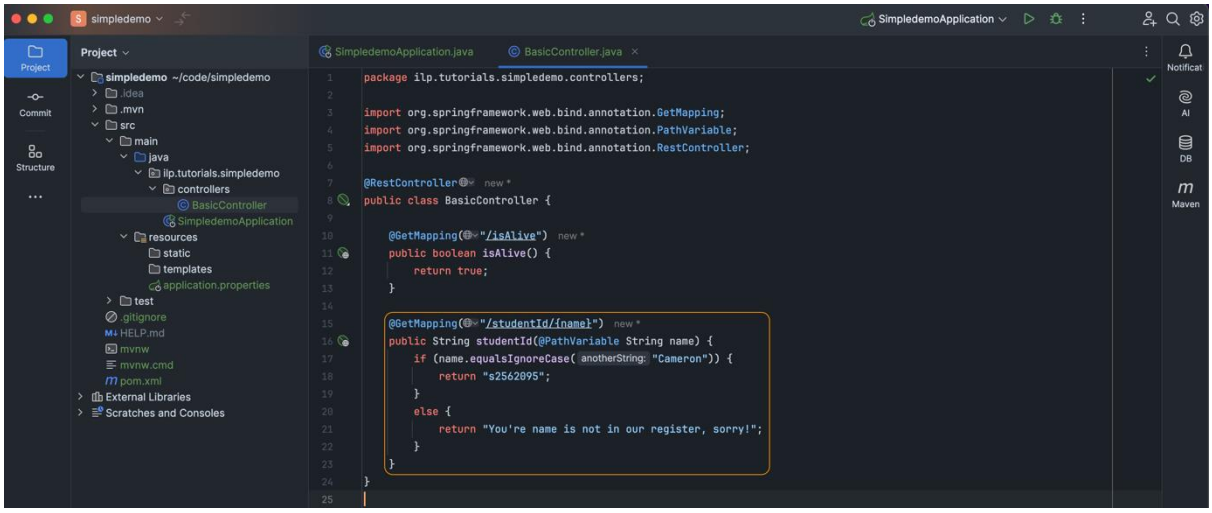
Creating The Controller Directory: We have our basics set up, now we can start building our simple REST service. Spring has included the main method that will start our application, but we need to control how our application responds to HTTP requests. To do this, we use a \Controller. Each method within the *Controller* will be implementing an endpoint within our application. To do this we create a controller directory within our application and create a *.java* file where we can start to program our controller.



This is replaced by the actuator/health endpoint in Spring (https://docs.spring.io/spring-boot/api/rest/actuator/health.html)

Explain and elaborate as mandatory for the final service

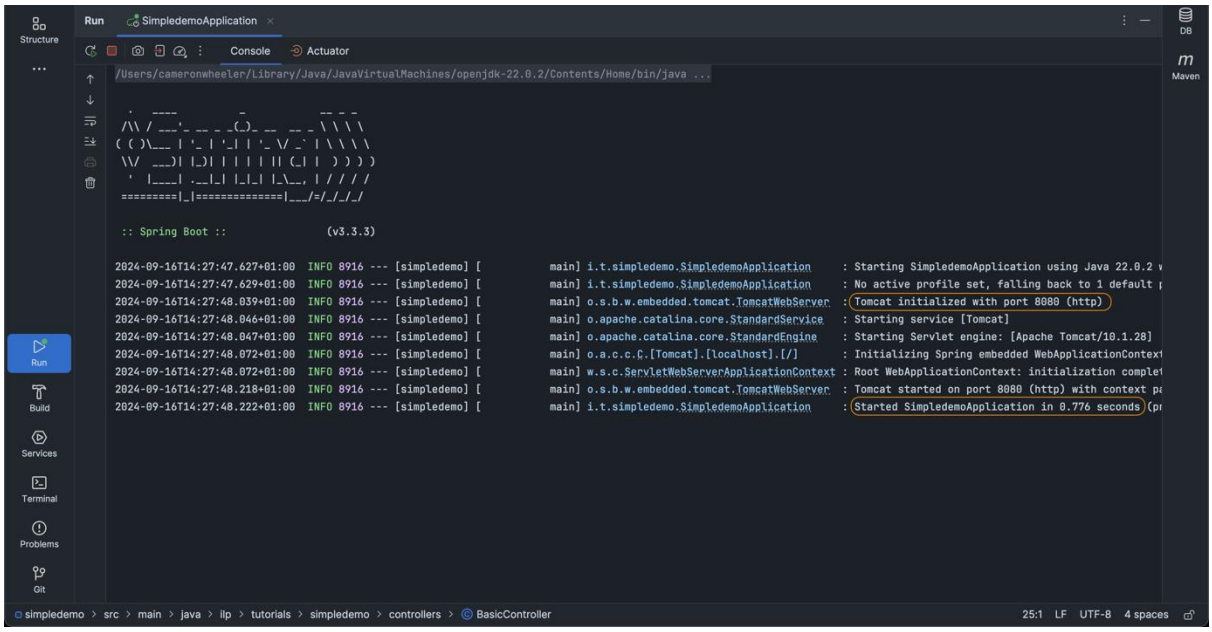
~~**isAlive Endpoint**: Now we can start to design our endpoints. Step 1: For the controller to work as we require, we need to annotate the method with the *@RestController* annotation. Step 2: We can now start to add API endpoints that users can send requests to. Our first endpoint is a simple *'isAlive'* endpoint. The *@GetMapping* annotation allows us to dictate that the *isAlive* method handles HTTP GET requests to this endpoint. Our handler method returns *true* if the server is active. When doing coursework that involves a REST service, I recommend always making a simple *isAlive* endpoint, this allows you to easily debug if an issue you are having is due to your method implementation, or with the service itself.~~



studentId Endpoint: Another way to configure endpoints is shown in the *studentId* endpoint. This endpoint again uses a *@GetMapping* annotation and simply returns my student ID if you pass in my name.

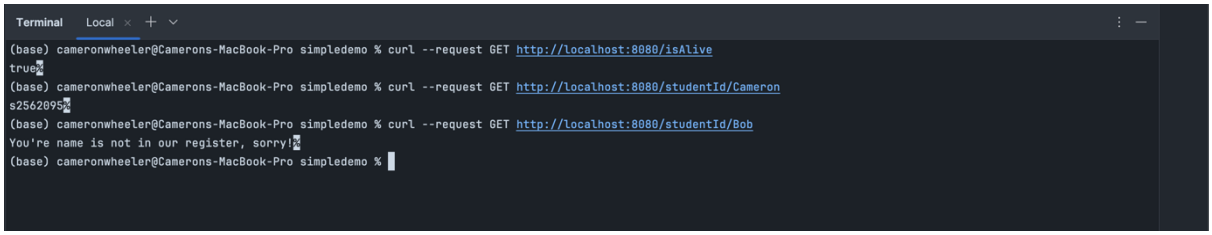
This should be utilizing a variable from the application.yml or app.config (either way is fine) which is set via a classical configuration from the Spring framework (in our case an environment variable). We will use this later on the have replacement values for the URL consumption.

If not, we return another message. We use the *@PathVariable* annotation to extract the URI variable that replaces *name*. For example, */studentId/Bob* will extract Bob from the URI and bind it to the method variable *name*. Note: This method does not look to check erroneous parameters in the request such as a space " ", "1234556", or "&&$$". However, in your coursework, always check for erroneous data! You will get some!

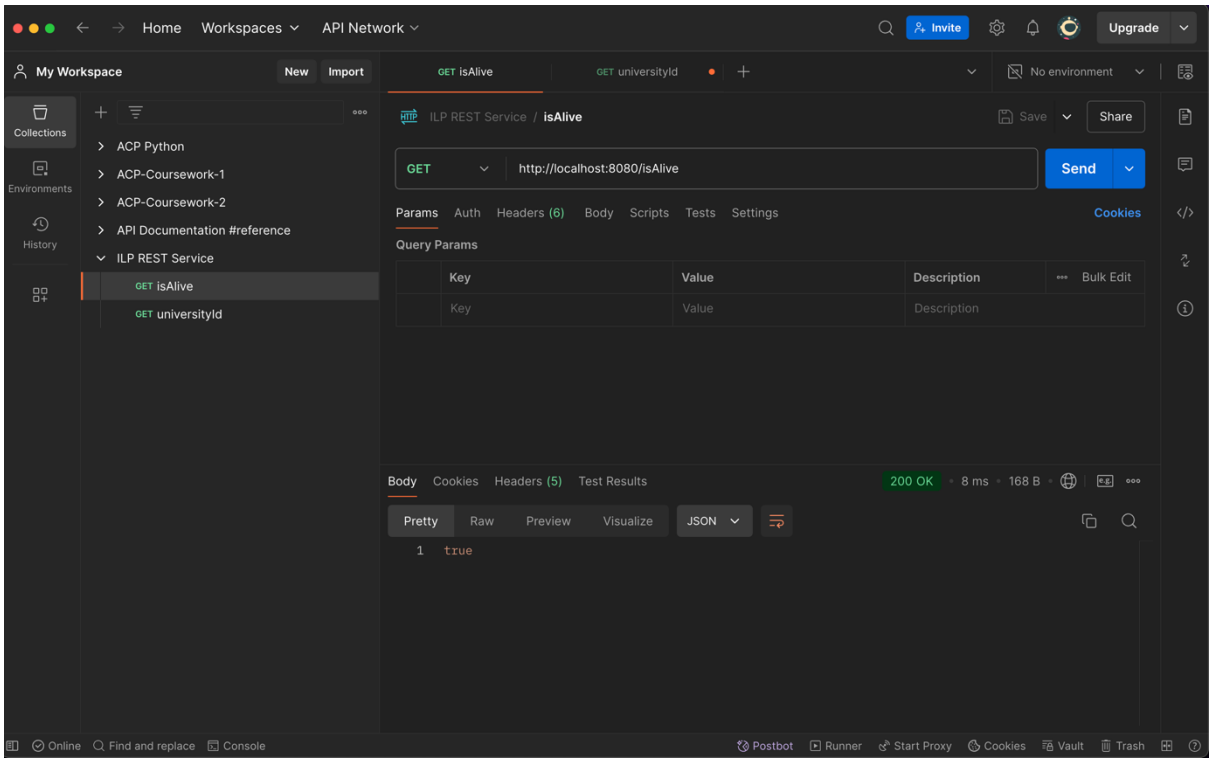


Server Starting: Now that we have our controller with two endpoints, we have the basic workings of a simple REST service. To ensure everything works before placing it into Docker, we can run the application and send HTTP requests to the endpoints. This figure shows what the Spring application starting should look like! If you get any issues that you cannot easily debug yourself, its often worth

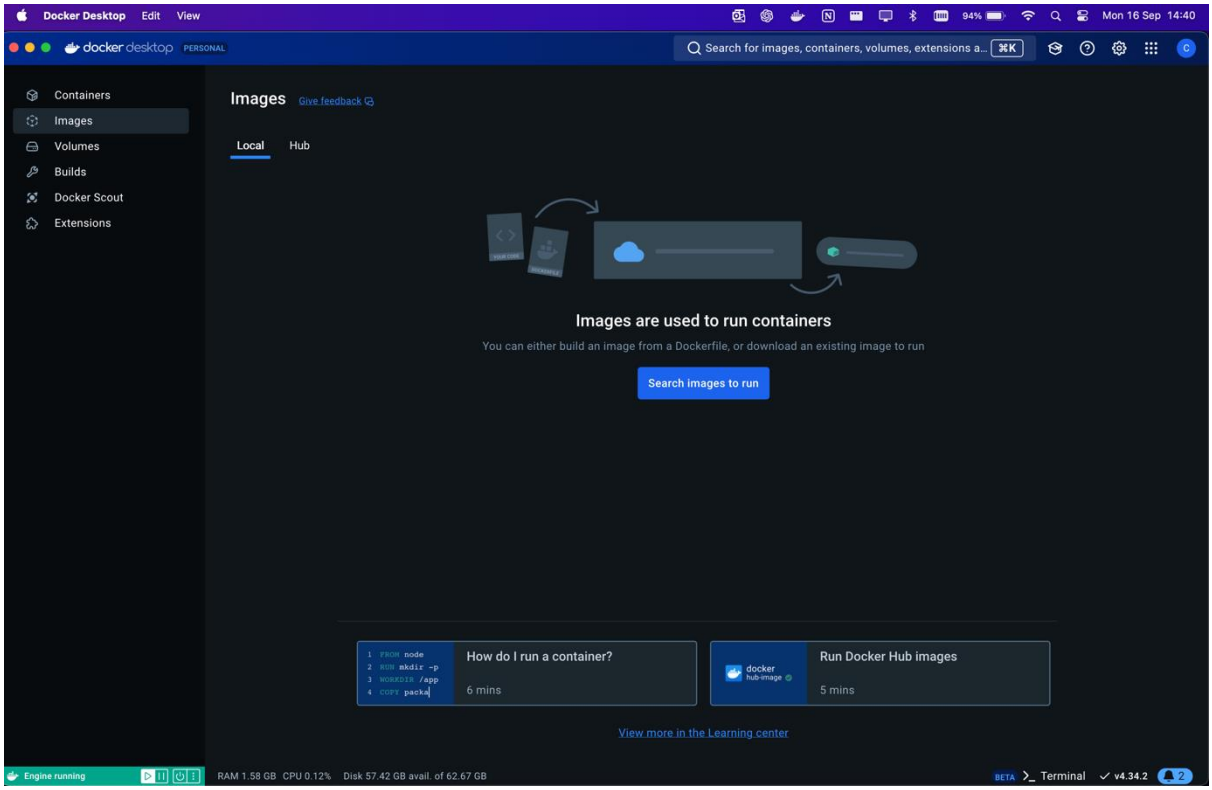
just starting from a clean sheet! Our application is listening to localhost port 8080. Now that our server is running, we can use *curl* or *Postman* to sent requests to each of the endpoints.



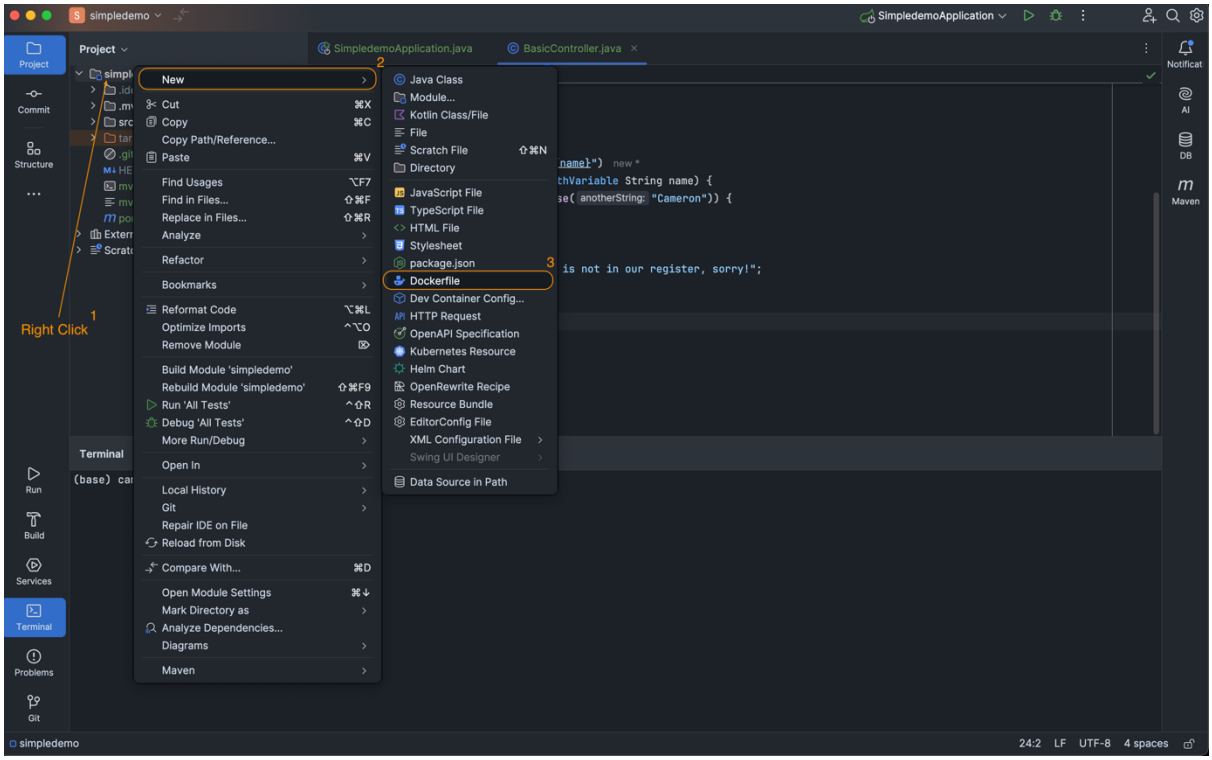
Curl Requests to Both Endpoints: *Curl* is a CLI tool that we can use to send requests to our service. The figure shows my terminal using *curl* to send GET requests to both endpoints. We can see our service is up and running and will correctly give my student ID back to me as a string when I send a GET request with my name in the URI path. It also correctly returns the message if we do not pass in my name. Curl works great with simple requests like this, but as things get more complex it can be a pain. I prefer to use *Postman* as it makes life a little easier.



Postman Request to *isAlive*: This is a screenshot of a GET request being sent to *isAlive* using Postman.

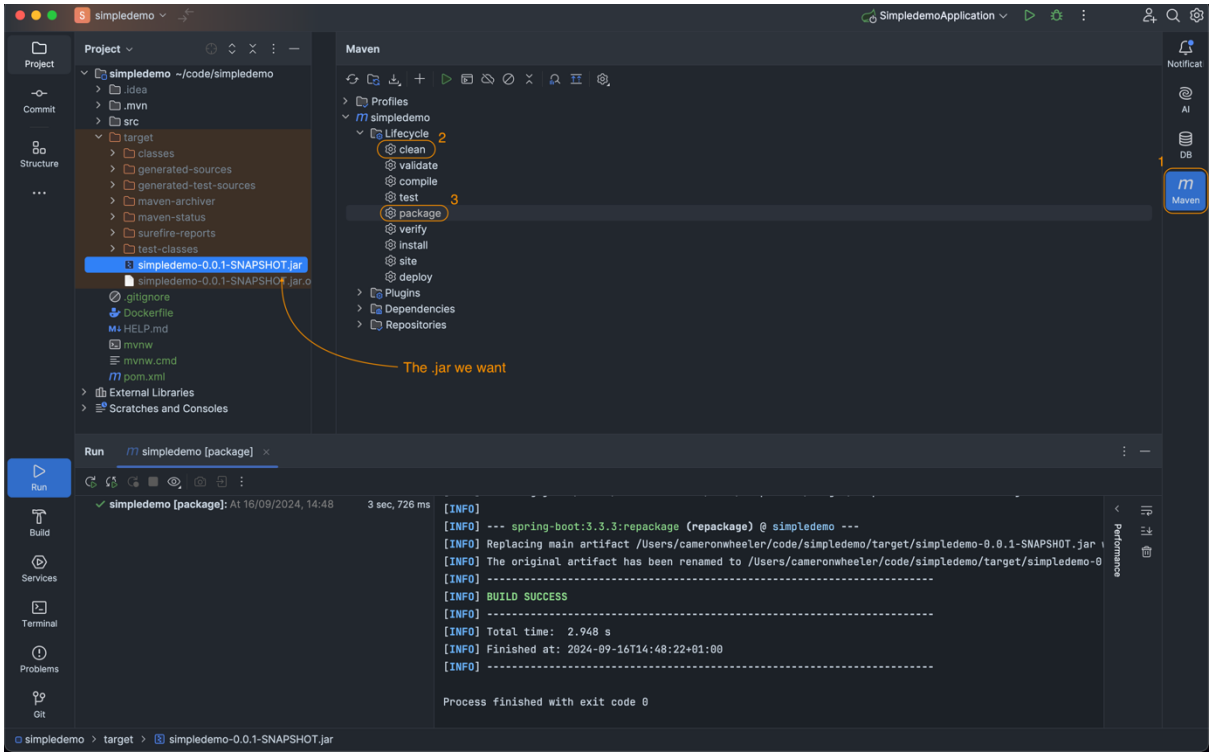


Getting Docker Running: Okay, we know our REST service works and that we can hit the endpoints, let's dockerize it. Stop the application from running before moving on. In order to use docker, we need download the Docker daemon, we can install the desktop app from https://www.docker.com/products/docker-desktop/ . This allows us to use either the CLI or the application itself to control Docker. Once downloaded, make sure the daemon is running before continuing. On my Mac (it should be similar for both Windows and Linux) there is the docker logo in the menu bar that indicates the daemon is running.

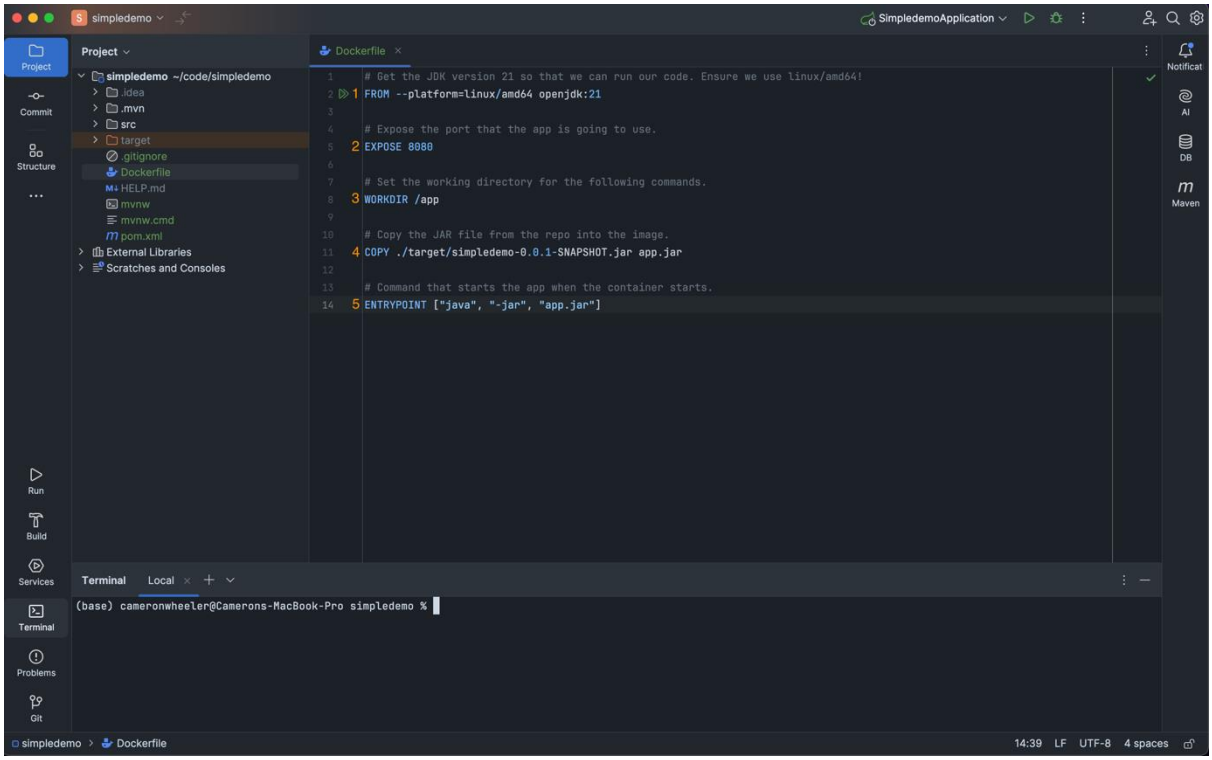


Adding Dockerfile to Project: The Dockerfile is what instructs Docker when building the image. Each command in the Dockerfile specifies what Docker needs to do to form the complete image, layer by

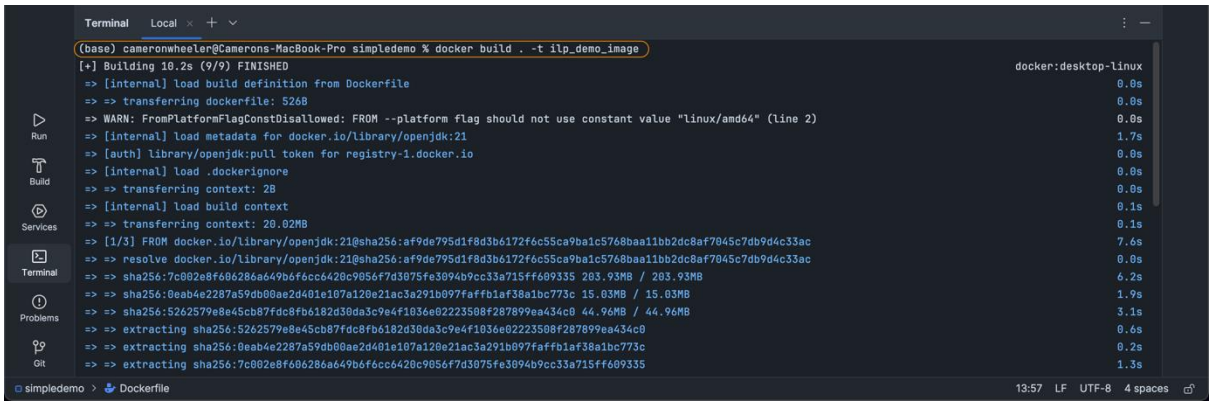
layer. In IntelliJ, we can add a Dockerfile to the repo by Step 1: Right-clicking the repo name. Step 2: Going to new. Step 3: Selecting Dockerfile (you can also just select file and type Dockerfile, make sure the D is capitalized and the f is lowercase). This will add a blank Dockerfile to our project.



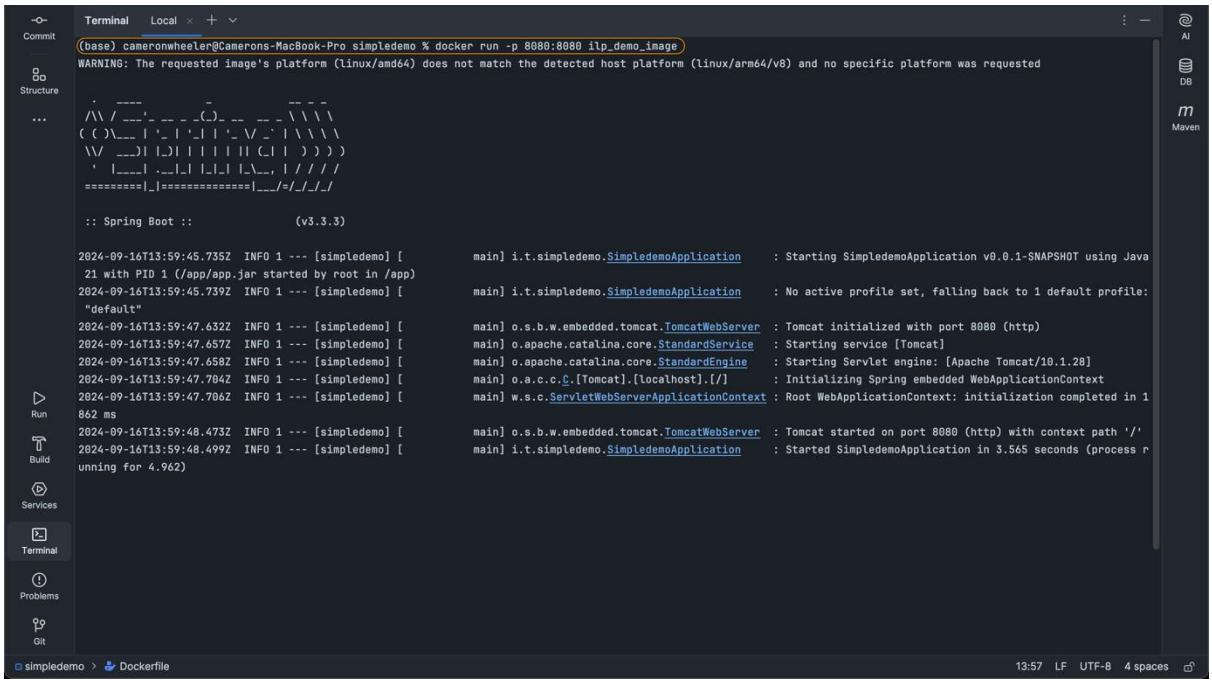
Maven Clean and Package to Create A JAR File: One of the key steps in creating a Docker image is transferring code from our source repository into the image itself. Since we are using Maven as our build manager, we can easily create the *.jar* file that we are going to use. Step 1: Select Maven on the right-hand side menu. Step 2: Clean the project. Step 3: Package the project (this will create the *.jar* file within the target directory). We will now use this *.jar* file when creating our Docker image in the next figure!



Dockerfile Contents: The figure above shows how we need to set up our Dockerfile. Step 1: Use the *FROM* command to instruct docker what base image to pull. If you are using a Mac be sure to include the *--platform=linux/amd64* tag otherwise it will not work when being run on a different architecture! Step 2: Docker containers have their own ports; we need our service running inside the container to communicate outside the container. Our service is listening to port 8080, so we instruct docker to *EXPOSE* port 8080. When getting running the container, we will have to map our exposed 8080 port to a port on our host machine (we do this in Fig 16). Step 3: Set the working directory with *WORKDIR* command. Step 4: We copy our *.jar* file we built using Maven over into our Docker image and rename it to simply *app.jar*. Step 5: We set the *ENTRYPOINT* command, this tells Docker what process to run when the container is started, in our case, boot up our REST service!

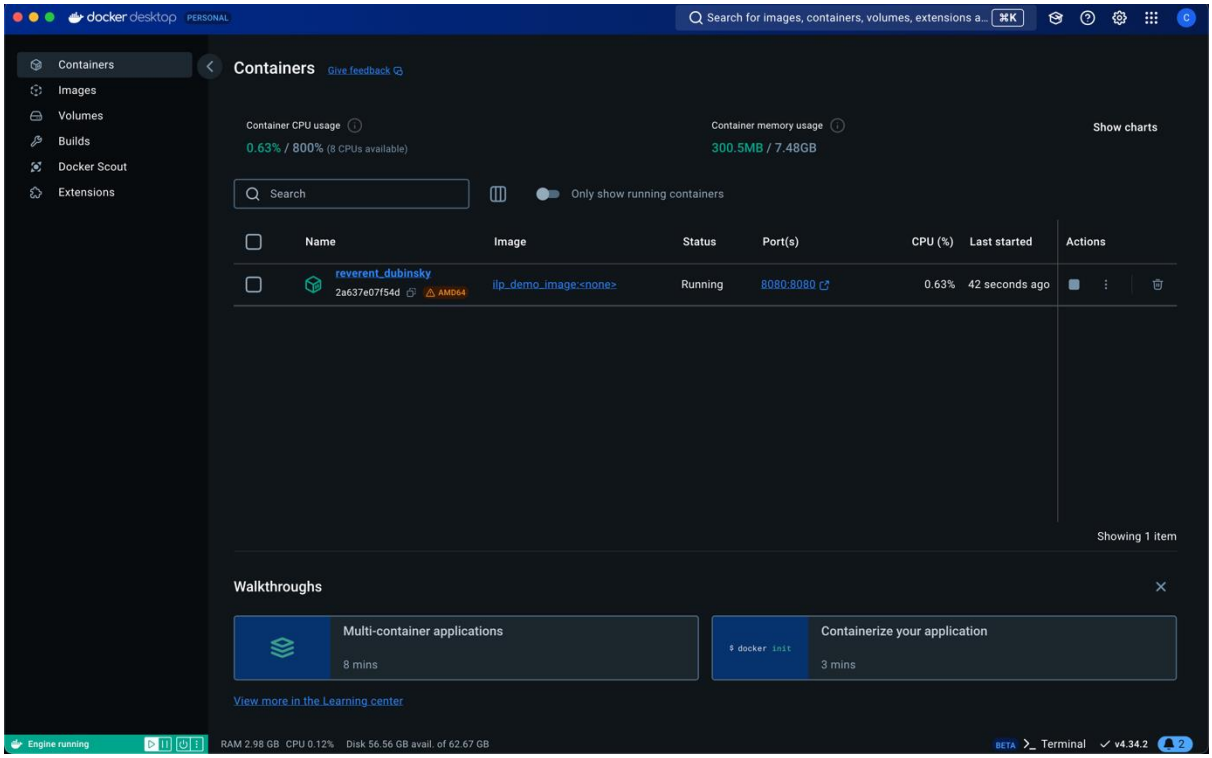


Building The Docker Image: Now that we have given Docker the instructions on how to build our image, lets build it. This can be done from the command line really easily. Making sure Docker is running and we are within the correct repo, we simply use *docker build <directory of Dockerfile> -t <preferred name of our image>* in our case, we are building the Dockerfile in the current directory, and naming our image *ilp_demo_image*. The output should look similar to what is displayed in the image.

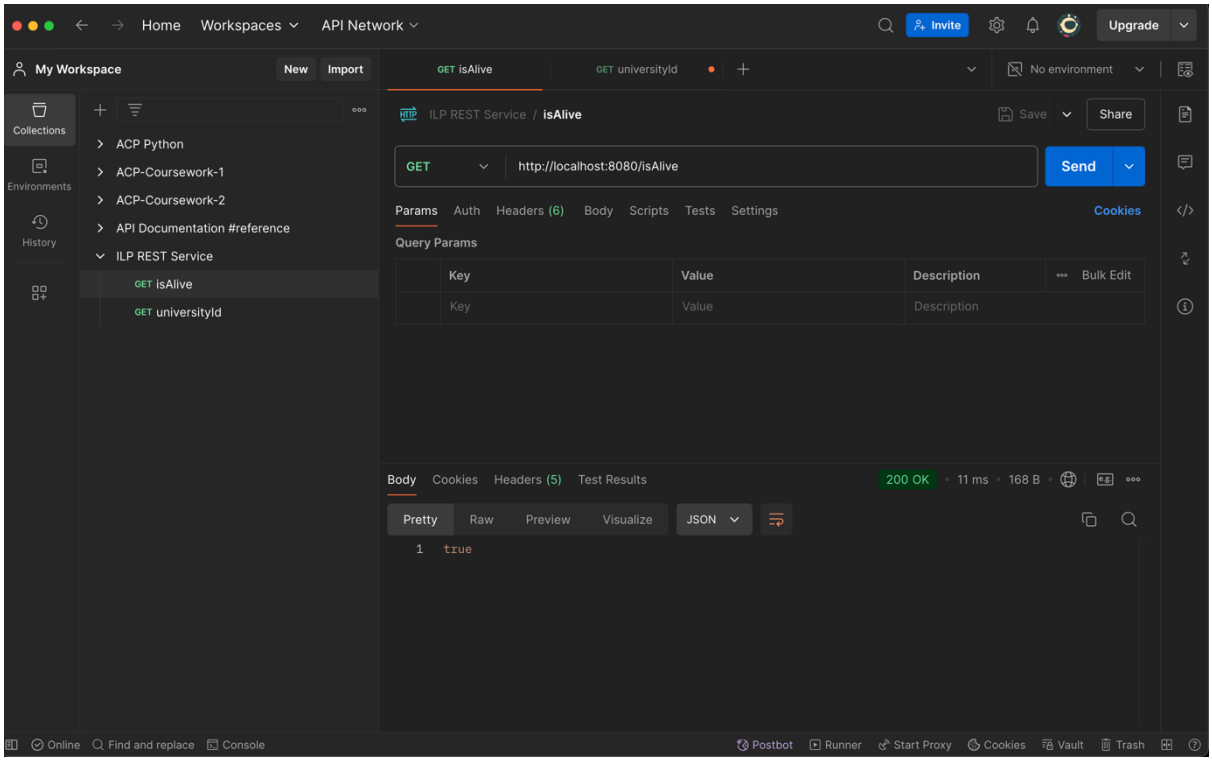


Running Docker Container: Now that we have build our image. We can spin up a container (a running image) and send requests to our service! To do this, we use the *docker run* command. Specifically, we use *docker run -p <map host port to container port> <the image we want to run>*. In Fig 14 we exposed port 8080, so we simply map our host 8080 port to the exposed 8080 port on the container.

We again get the same **SPRING** which shows us that our container is running and our REST service is listening for HTTP requests.



View of Docker Container Running in Docker: We can also go and check the Docker desktop app to confirm our container is running. If we wanted to, we can kill the container, inspect the container's logs or view its stats like CPU and memory usage!



Postman Sending Request to Service Inside Docker Container: We can now send HTTP requests to our service running in the container much like we did our service running without Docker. Again using tools like *curl* or *Postman*. The figure shows an *isAlive* request using *Postman*.