**Java App+Docker Integration,**

**Docker Knowledge**

**Overview**

In this article, we'll cover the process of creating a [Docker](https://www.docker.com/) image of a [Spring Boot](https://spring.io/projects/spring-boot) application, using [Dockerfile](https://docs.docker.com/engine/reference/builder/" \t "_blank) and [Maven](https://maven.apache.org/) and then run the image we've created.

The source code for this tutorial can be found

This tutorial assumes that you have Docker installed on your machine. If not, you can follow the official [Docker install](https://docs.docker.com/install/) guide based on your operating system.

If you'd like to read more about Docker, we've covered it in detail in - [Docker: A High Level Introduction](https://stackabuse.com/docker-a-high-level-introduction/).

**The Spring Boot Application**

To run attached application, just run StudentServicesApplication.java, this will initiate tomcat and services up, & running.

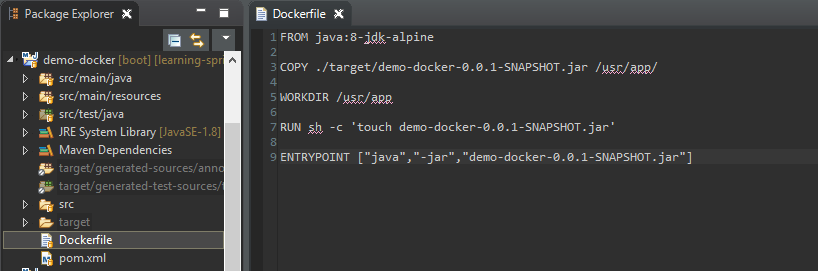
### Dockerizing the Spring Boot App

Now let's dockerize our previously made Spring Boot Application. We will cover the two most commonly used approaches:

* **Dockerfile** – Specifying a file that contains native Docker commands to build the image
* **Maven** – Using a Maven plugin to build the image

#### Dockerizing using Dockerfile

A Dockerfile is just a regular .txt file that includes native Docker commands that are used to specify the layers of an image. To do so, let's create a text file named "Dockerfile":



The content of the file itself can look something like this:

FROM java:8-jdk-alpine

COPY ./target/student-services-0.0.1-SNAPSHOT.jar /usr/app/

WORKDIR /usr/app

#RUN sh -c 'touch demo-docker-0.0.1-SNAPSHOT.jar'

ENTRYPOINT ["java","-jar","student-services-0.0.1-SNAPSHOT.jar"]

Let's take a look at the commands and fully understand them before proceeding:

* **FROM** – The keyword FROM tells Docker to use a given base image as a build base. We have used 'java' with tag '8-jdk-alpine'. Think of a tag as a version. The base image changes from project to project. You can search for images on [docker-hub](https://hub.docker.com/).
* **COPY** - This tells Docker to copy files from the local file-system to a specific folder inside the build image. Here, we copy our .jar file to the build image (Linux image) inside /usr/app.
* **WORKDIR** - The WORKDIR instruction sets the working directory for any RUN, CMD, ENTRYPOINT, COPY and ADD instructions that follow in the Dockerfile. Here we switched the workdir to /usr/app so as we don't have to write the long path again and again.
* **RUN** - This tells Docker to execute a shell command-line within the target system. Here we practically just "touch" our file so that it has its modification time updated (Docker creates all container files in an "unmodified" state by default).
* **ENTRYPOINT** - This allows you to configure a container that will run as an executable. It's where you tell Docker how to run your application. We know we run our spring-boot app as java -jar <app-name>.jar, so we put it in an array.

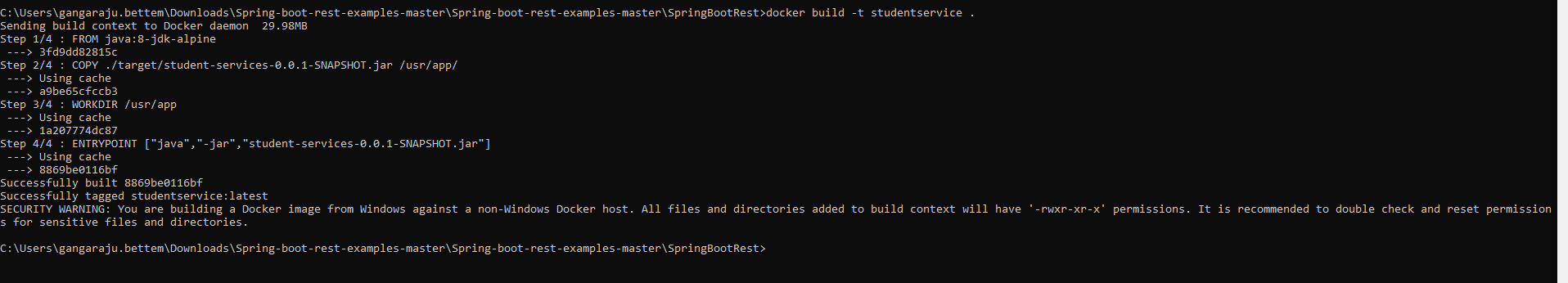
More documentation can be found on the [Dockerfile reference](https://docs.docker.com/engine/reference/builder/" \t "_blank) page.

Before moving further, we need a Spring Boot .jar file. This file will be used to create the Docker image as mentioned above.

Run the mvn clean install command to make sure that it's generated.

Let's build the image using this Dockerfile. To do so, move to the root directory of the application and run this command:

C:\Users\gangaraju.bettem\Downloads\Spring-boot-rest-examples-master\Spring-boot-rest-examples-master\SpringBootRest>docker build -t studentservice .



We built the image using docker build. We gave it a name with the -t flag and specified the current directory where the Dockerfile is. The image is built and stored in our local docker registry.

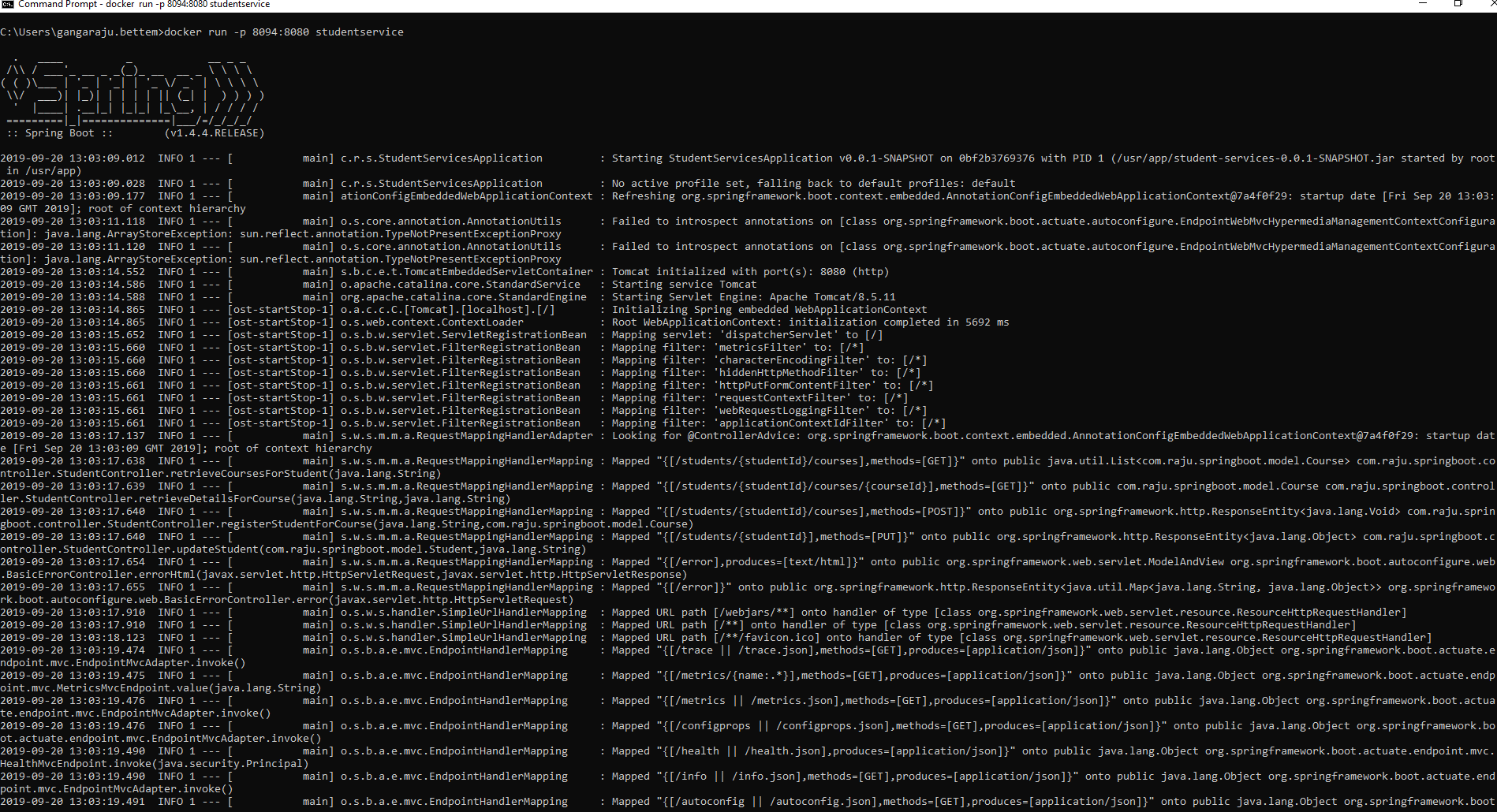
Let's check our image:

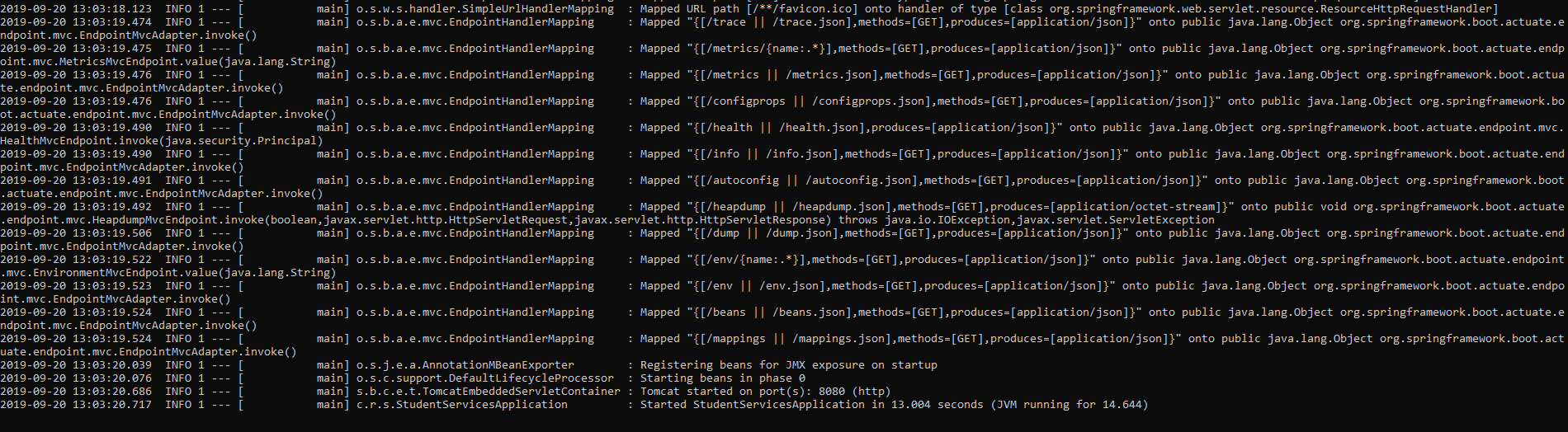
$ docker images

Docker images 1

And finally, let's run our image:

C:\Users\gangaraju.bettem>docker run -p 8094:8080 studentservice





We know that each container is an isolated environment in itself and we have to map the port of the host operating system - 8094 and the port inside the container - 8080, which is specified as the -p 8094:8080 argument.

Now, we can access the endpoint on <http://localhost:8094/students/Student1/courses>

