TP Gitlab

Ce projet consiste à créer un dépôt Git local, à ajouter des fichiers, à effectuer des commits, à créer des branches, à fusionner des branches, etc. Voici un guide étape par étape :

1. **Créer un nouveau dépôt Git** :
    a. Créez un nouveau dossier pour votre projet.
    b. Ouvrez une ligne de commande et accédez au répertoire que vous venez de créer.
    c. Initialiser un nouveau dépôt Git en utilisant la commande `git init`.
2. **Ajouter des fichiers :**
    a. Créez quelques fichiers (par exemple, `index.html`, `styles.css`, `script.js`).
    b. Ajoutez ces fichiers au suivi Git à l'aide de la commande `git add`.
3. **Effectuer un commit** :
    a. Effectuez votre premier commit en utilisant la commande `git commit -m "Premier commit"`.
4. **Créer une branche :**
    a. Créez une nouvelle branche à partir de la branche principale (généralement `master` ou `main`) en utilisant la commande `git branch nom_de_votre_branche`.
    b. Passez à votre nouvelle branche avec `git checkout nom_de_votre_branche` ou combinez les deux étapes en utilisant `git checkout -b nom_de_votre_branche`.
5. **Effectuer des modifications** :
    a. Modifiez quelques fichiers dans votre nouvelle branche.
6. **Vérifier l'état du dépôt :**
    a. Utilisez la commande `git status` pour voir quels fichiers ont été modifiés et quels fichiers sont en attente de commit.

7. **Ajouter et valider les modifications** :
    a. Ajoutez les fichiers modifiés au suivi avec `git add`.
    b. Validez les modifications avec `git commit`.
8. **Revenir à la branche principale** :
    a. Revenez à votre branche principale avec `git checkout nom_de_votre_branche_principale`.
9. **Fusionner les branches :**
    a. Fusionnez votre branche de travail dans la branche principale avec `git merge nom_de_votre_branche`.
10. **Afficher l'historique des commits :**
    a. Utilisez `git log` pour afficher l'historique des commits.
11. **Gérer les conflits** :
    a. Si vous avez des conflits lors de la fusion des branches, vous devrez les résoudre manuellement. Ouvrez les fichiers en conflit, résolvez les conflits, puis ajoutez et validez les modifications.
12. **Supprimer une branche** :
    a. Supprimez une branche après fusion avec `git branch -d nom_de_votre_branche`.

Création d'un conteneur Docker pour une application Spring Boot

1-Download the repository

First, we will download my pre-made repository on Github with all the required dependencies to save some time.

https://github.com/erickjhorman/spring-boot-docker/tree/master/spring-boot-app

2-Check the project

Open the downloaded project in your IntelliJ IDEA and run the command **mvn clean compile** to check if the project is working.

3- Check the application

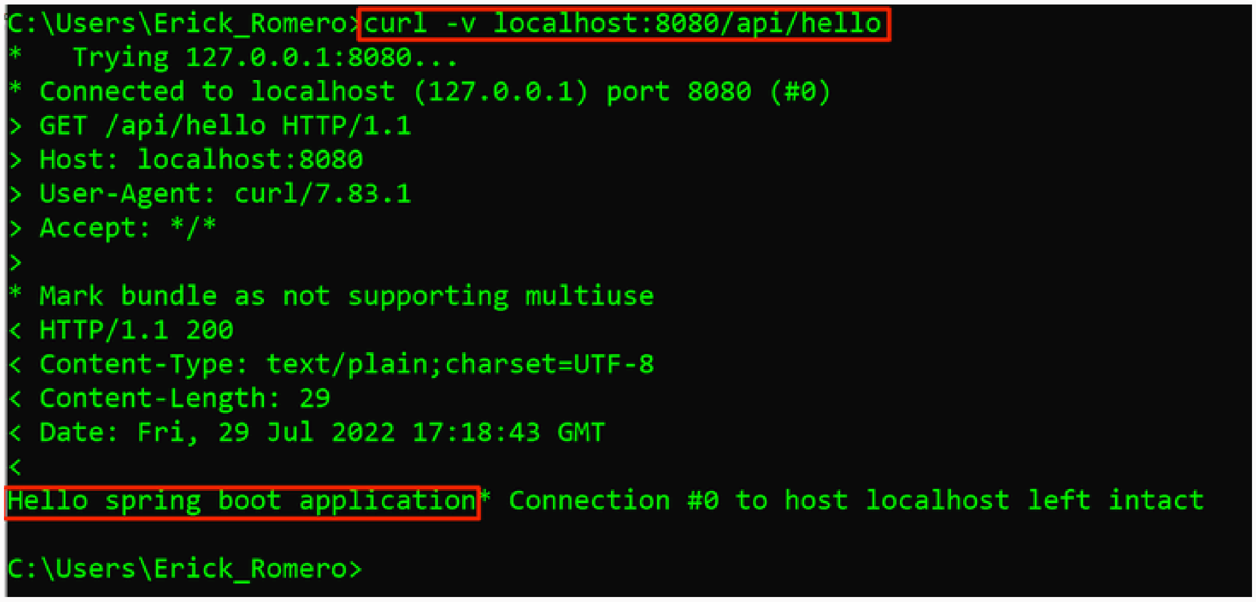
Before we dockerize a Spring Boot app, we'll need to check if our Spring Boot application is also running. Let's run the next command in the internal terminal in IntelliJ (run it from your cmd or Git Bash in Windows):

mvn clean spring-boot:run

curl -v localhost:8080/api/hello

You should see the following message in your cmd to validate that the app is already running and returning the proper message in this route:

running and returning the proper message in this route:



```
C:\Users\Erick_Romero>curl -v localhost:8080/api/hello
*   Trying 127.0.0.1:8080...
* Connected to localhost (127.0.0.1) port 8080 (#0)
> GET /api/hello HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.83.1
> Accept: */*
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200
< Content-Type: text/plain;charset=UTF-8
< Content-Length: 29
< Date: Fri, 29 Jul 2022 17:18:43 GMT
<
Hello spring boot application* Connection #0 to host localhost left intact

C:\Users\Erick_Romero>
```

4-Add configuration to dockerize the Spring Boot application

The next step of dockerizing a Spring Boot app is adding the configuration to the Dockerfile for Spring Boot app.

To clarify, Docker is a platform that combines your application, its dependencies, and even an OS into an image that can run on any platform.

Now we will create a Dockerfile to add all the configurations to dockerize our Spring Boot application.

Create a Dockerfile and add the next configurations (Docker will read as a pipeline to apply them):

JavaScript
```
FROM openjdk:11

VOLUME /tmp

EXPOSE 8080

ARG JAR_FILE=target/spring-boot-docker.jar

ADD ${JAR_FILE} app.jar



ENTRYPOINT ["java","-jar","/app.jar"]
```

Now I will explain a little bit about these commands:

> **FROM** creates a layer from an existing Spring Boot Docker image that exists locally or in any container registry. **openjdk:11** will be the one to use.
> **VOLUME** creates a specific space to persist some data in your container. The **tmp** folder will store information.

EXPOSE informs Docker that the container listens to the specified network ports at runtime. This is the port to access the Spring Boot container and will be used to run the container.
ARG defines a variable that can be passed to the application at runtime. For example, we pass the location of the final jar file within the target folder and save it in a **JAR_FILE** variable. You can also pass more arguments like credentials, keys, and environment variables with their respective values.
ADD copies new files, directories or remote file URLs from the source and adds them to the filesystem of the image at the provided path. In our case we add the Spring Boot application to the Docker image from the source path (the **JAR_FILE** variable) to a destination named app.jar.
ENTRYPOINT specifies the command that Docker will use to run our app. In this case it will pass the common command to run a jar — **java -jar <name of the jar>** — so in this case it is **java -jar app.jar** to our **ENTRYPOINT** option (remember that we renamed the spring-boot-docker.jar file to **app.jar**).

That's all for the configuration.

5-Generate a .jar file

Now we need to generate our .jar file running the **mvn install** command in your IntelliJ terminal. This will generate a .jar file with all the classes from our application.

When the process is finished, go to the target folder from your project and see the next **spring-boot-docker.jar** file.

6-Build a Spring Boot Docker image

Great, we’re done with the preparations, and it’s time to build our image using Docker Engine.

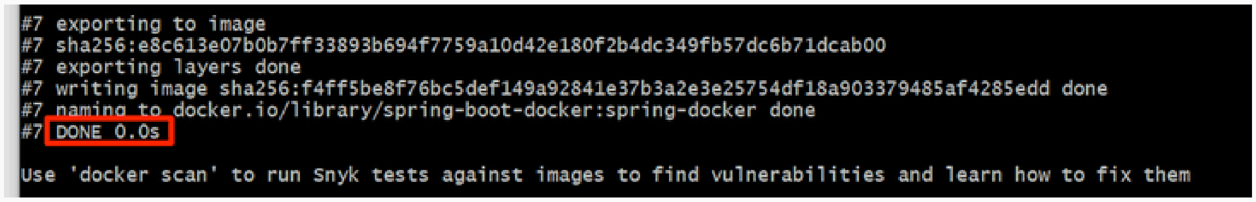
Make sure you have Docker Engine installed. Run the docker ps or docker info command on the terminal screen to check it. If the command is not found, you may need to install Docker first. In this case, please follow this link and find the installer for your OS.

Run your Docker Engine. Find the folder with the Dockerfile of your Spring Boot project in the terminal and execute the following command (make sure to end the command with a space and a dot):

docker build -t spring-boot-docker:spring-docker .

The **“build”** command will build an image according to the instructions we passed to the Dockerfile, and the -t flag is used to add a tag for our image.

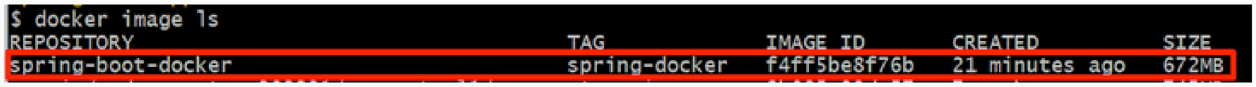
In a few minutes, you will see that the image was successfully created:



```
#7 exporting to image
#7 sha256:e8c613e07b0b7ff33893b694f7759a10d42e180f2b4dc349fb57dc6b71dcab00
#7 exporting layers done
#7 writing image sha256:f4ff5be8f76bc5def149a92841e37b3a2e3e25754df18a903379485af4285edd done
#7 naming to docker.io/library/spring-boot-docker:spring-docker done
#7 DONE 0.0s

Use 'docker scan' to run Snyk tests against images to find vulnerabilities and learn how to fix them
```

Run the **docker image ls** command to check if the image exists. You should be able to see your image in the repository.


```
$ docker image ls
REPOSITORY                                         TAG             IMAGE ID       CREATED          SIZE
spring-boot-docker                                 spring-docker   f4ff5be8f76b   21 minutes ago   672MB
```

How to change the base image

Changing the base image in a Dockerfile is a straightforward process that can significantly impact your container's functionality, size, and security. To change the base image, follow these steps:

1. Choose a new base image

Before making any changes, research and select a suitable base image that meets your application's requirements. Consider factors such as the operating system, size, security, and compatibility with your application. Official images from the Docker Hub or other trusted sources are recommended.

In our example, we will choose eclipse-temurin:11 as a base image, a lightweight Java image.

2. Update the Dockerfile

Open your Dockerfile in a text editor. Locate the ***`FROM`*** instruction at the file's beginning, specifying the current base image. Replace the existing image with the new one you've chosen. For example, if you want to change the base image from `openjdk:11` to **`eclipse-temurin:11`**, update the ***`FROM`*** instruction as follows:

JavaScript

```
FROM eclipse-temurun:11
```

3. Adjust dependencies and configurations

Depending on the differences between the old and new base images, you may need to update the dependencies, configurations, or commands in your Dockerfile. Ensure all required packages, libraries, and tools are installed and configured correctly for the new base image.

This **eclipse-temurin:11** has the necessary dependencies to run your application.

4. Test your changes

After updating the Dockerfile, rebuild your Docker image using the `*docker build*` command:

JavaScript

```
docker build -t your-image-name .
```

Replace `*your-image-name*` with a suitable name for your image. Remember the period at the end of the command, which indicates the build context (usually the current directory).

Or you can use **commit** option to create a new image with the new changes that you add.

Run the Spring Boot Docker image in a container

Now, we’ll run our image in a container, but first, let’s make sure we won’t get an error trying to point our container port to the localhost one.

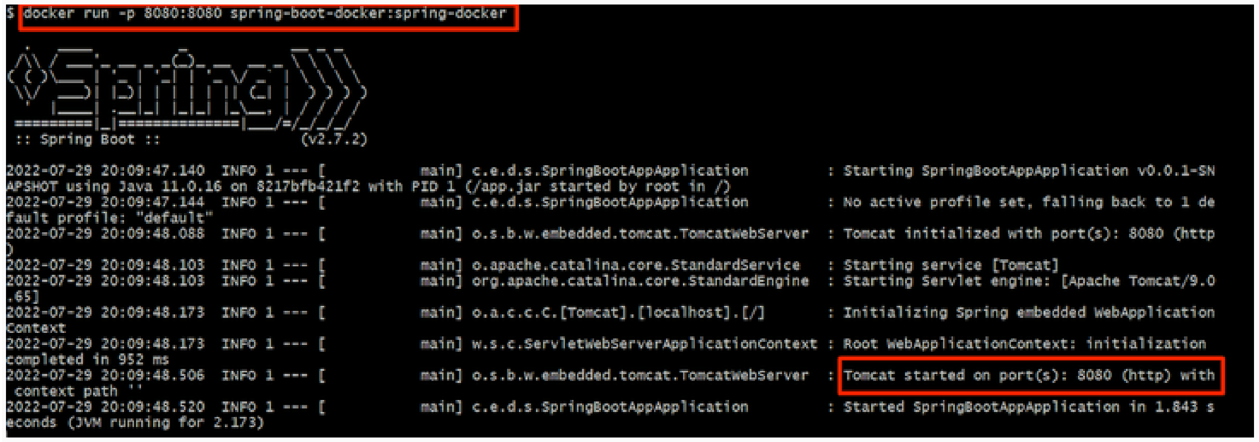
Run the next command:

docker run -p 8080:8080 spring-boot-docker:spring-docker .

You may add the **-d** flag before **-p** to avoid seeing any logs and run the container in the background mode.

The flag **-p** creates a firewall rule that maps the previously exposed container port:8080 to the :8080 port on your machine.

If everything is done right, our container should be running.



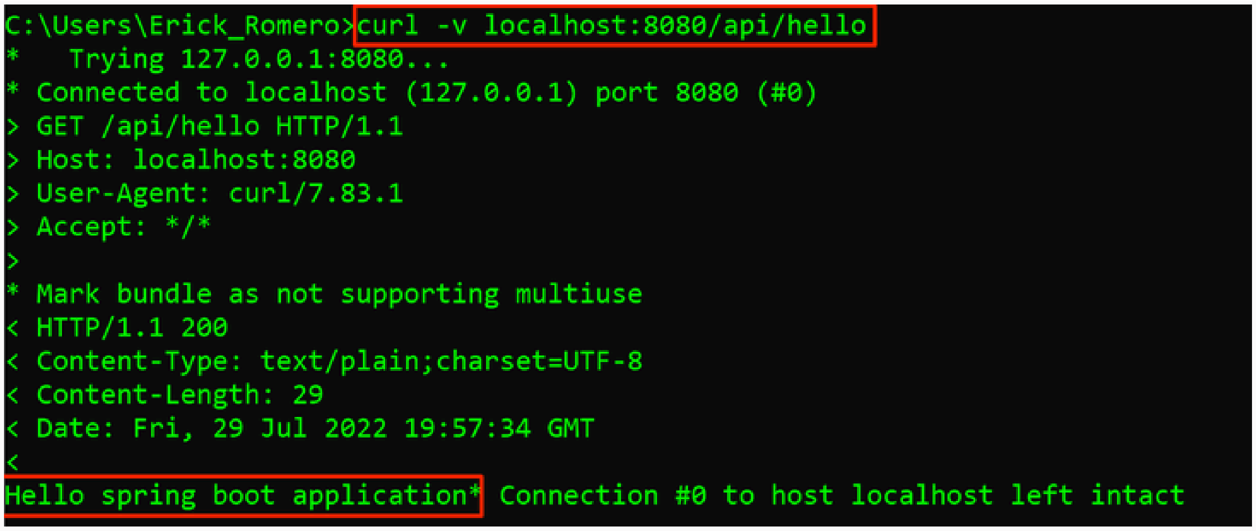
```
$ docker run -p 8080:8080 spring-boot-docker:spring-docker

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::                (v2.7.2)

2022-07-29 20:09:47.140  INFO 1 --- [           main] c.e.d.s.SpringBootAppApplication         : Starting SpringBootAppApplication v0.0.1-SN
APSHOT using Java 11.0.16 on 8217bfb421f2 with PID 1 (/app.jar started by root in /)
2022-07-29 20:09:47.144  INFO 1 --- [           main] c.e.d.s.SpringBootAppApplication         : No active profile set, falling back to 1 de
fault profile: "default"
2022-07-29 20:09:48.088  INFO 1 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port(s): 8080 (http
)
2022-07-29 20:09:48.103  INFO 1 --- [           main] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2022-07-29 20:09:48.103  INFO 1 --- [           main] org.apache.catalina.core.StandardEngine  : Starting Servlet engine: [Apache Tomcat/9.0
.65]
2022-07-29 20:09:48.173  INFO 1 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplication
Context
2022-07-29 20:09:48.173  INFO 1 --- [           main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization
completed in 952 ms
2022-07-29 20:09:48.506  INFO 1 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8080 (http) with
 context path ''
2022-07-29 20:09:48.520  INFO 1 --- [           main] c.e.d.s.SpringBootAppApplication         : Started SpringBootAppApplication in 1.843 s
econds (JVM running for 2.173)
```

Then, run the **curl -v localhost:8080/api/hello** command in your terminal to test if we can access the application's endpoint.

You must see the same response as at the beginning of the tutorial.



```
C:\Users\Erick_Romero>curl -v localhost:8080/api/hello
*   Trying 127.0.0.1:8080...
* Connected to localhost (127.0.0.1) port 8080 (#0)
> GET /api/hello HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.83.1
> Accept: */*
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200
< Content-Type: text/plain;charset=UTF-8
< Content-Length: 29
< Date: Fri, 29 Jul 2022 19:57:34 GMT
<
Hello spring boot application* Connection #0 to host localhost left intact
```