Job Exercise

Solution by Fernanda Aguilar Corona

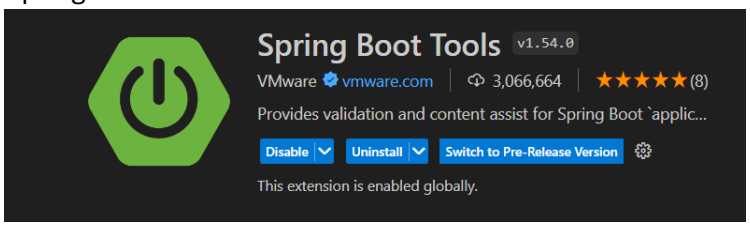
Index

Spring Boot Application and REST Controller

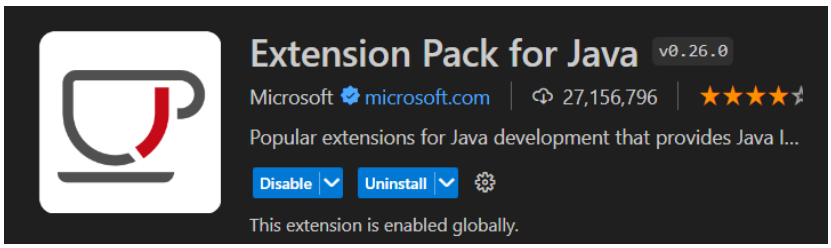
Step 1: Set up environment to develop a Spring-Boot application

As I have VSCode I configure this IDE for Spring Boot.

1. Install the next extensions:
    a. Spring Boot



    b. Java



2. Create the Java project and configure the dependencies, in this case were:
    a. Spring Web
    b. Lombok
    c. H2
    d. JPA
3. In the file *pom.xml* we can see some features of the application, some of them the dependencies.

Step 2: Configure Data Base features on *application.properties*

The file *application.properties* is in the directory *src/main/**java/resources***

application.properties

src > main > resources > application.properties

```
spring.application.name=appjob
server.port = 9090


# For Data Base with H2
# JDBC is thee tool for connectivity -> H2
# Database name in memory -> testdb
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driverClassName=org.h2.Driver
#credentials to access
spring.datasource.username=fer
spring.datasource.password=


# For queries with JPA and SQL, such as the terminal
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.H2Dialect
spring.jpa.hibernate.ddl-auto=update
spring.h2.console.enabled=true
spring.h2.console.path=/h2
```

IMPORTANT: The data stored in the data base *testdb* is temporal, so once we finish it erase.

Step 3: Crate *model* directory with Java classes (*User* and *Account*)

Here I have used *Lombok* to make more practical the coding, because this tool allows us to do not write setters, gettes, constructors, etc. in our Java classes, we only need to declare the attributes of the object and, also these can change dynamically with some *Lombok*'s tags starting with @

Having said this, we create these Java classes which are called *models* on a directory called *models* on the directory *src/main/**java/model***

Step 4: Create repository used by H2 database with JPA

We create the repository with JPA (Java Persistence API) and work between Java objects and relational databases making the mapping of Java objects to tables in a relational database, such as SQL.

https://web.postman.co/workspace/My-Workspace~55dfa3d9-609e-4ecf-8905-1d2fd1835726/request/35040087-74b33cb0-9e21-4efd-a62c-b54f2e51d380?tab=body

In this repository, we make use of *JpaRepository* interface which let us make the following operations with the data stored, for example if we want to look for an item by its ID we can use *getByID(id)* method.

Method Summary

All Methods | **Instance Methods** | **Abstract Methods** | **Default Methods** | **Deprecated Methods**

Modifier and Type	Method	Description
void	**deleteAllByIdInBatch**(**Iterable** <**ID**> ids)	Deletes the entities identified by the given ids using a single query.
void	**deleteAllInBatch**()	Deletes all entities in a batch call.
void	**deleteAllInBatch**(**Iterable** <**T**> entities)	Deletes the given entities in a batch which means it will create a single query.
default void	**deleteInBatch**(**Iterable** <**T**> entities)	**Deprecated.** Use `deleteAllInBatch(Iterable)` instead.
<S extends **T**> **List** <S>	**findAll**(**Example** <S> example)	
<S extends **T**> **List** <S>	**findAll**(**Example** <S> example, **Sort** sort)	
void	**flush**()	Flushes all pending changes to the database.
T	**getById**(**ID** id)	**Deprecated.** use `getReferenceById(ID)` instead.
T	**getOne**(**ID** id)	**Deprecated.** use `getReferenceById(ID)` instead.
T	**getReferenceById**(**ID** id)	Returns a reference to the entity with the given identifier.
<S extends **T**> **List** <S>	**saveAllAndFlush**(**Iterable** <S> entities)	Saves all entities and flushes changes instantly.
<S extends **T**> S	**saveAndFlush**(S entity)	Saves an entity and flushes changes instantly.

An example is the following:

application.properties | BookRepo.java

src > main > java > com > app > appjob > repo > BookRepo.java > ...

```java
package com.app.appjob.repo;

import com.app.appjob.model.Book;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;


@Repository
public interface BookRepo extends JpaRepository<Book,Long> {

}
```

And this file is created in a directory called *repo* (in this example) into the *java* directory.

Step 5: Create a REST controller

First, we know that REST API let us make some operations in an Client-Server architecture, such as: POST, PUT, DELETE, GET.

- POST: Stored data in the server.
- GET: Get data from the server.
- PUT: Edit, change data from the server.
- DELETE: delete data from the server.

Having said this, before the Java class we make use of the tag *@RestController* to specify that the operations in this class will be in the interface between the systems which use HTTP protocol.

Then, we have the next requests:

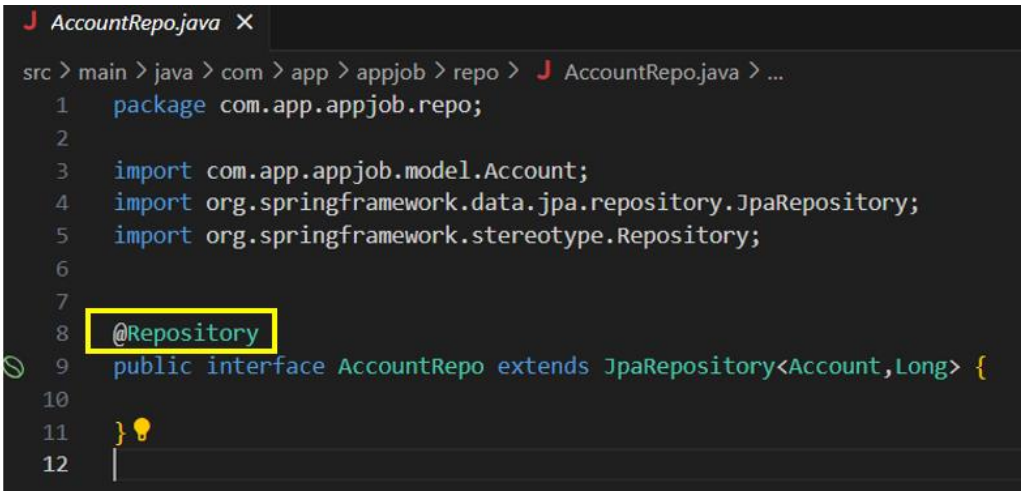
- @GetMapping: to get some data, such as List, IDs, names, etc.
- @PostMapping: Add or update data, such as add new user or user accounts.
- @DeleteMapping: delete data

Now, as REST is an API which allows the communication between two systems with HTTP protocol, so we can have some HTTP responses:

- ResponseEntity<>(HttpStatus.OK)
- ResponseEntity<>(HttpStatus.NOT_FOUND)
- ResponseEntity<>(object,HttpStatus.OK)
- ResponseEntity<>(HttpStatus.INTERNAL_SERVER_ERROR)

Now, when we are using repositories add the following tags:

- @Repository on the top of the class

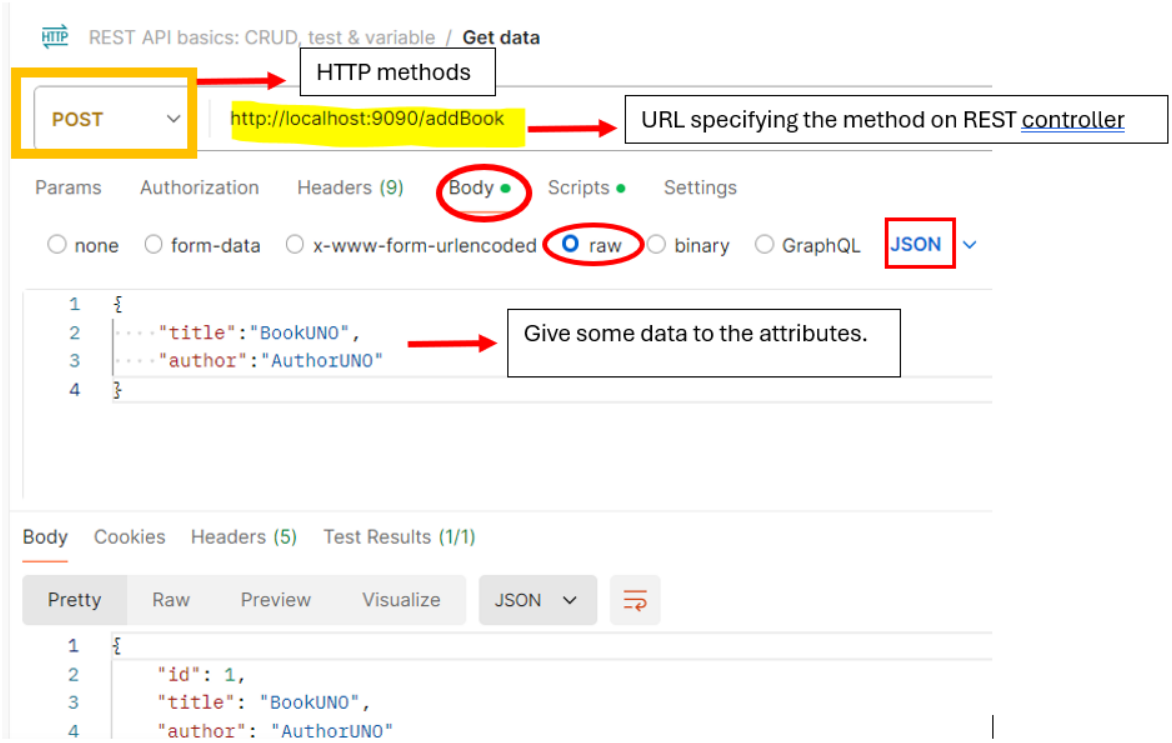


- @Autowired for each repository in the controller

```
@Autowired
private UserRepo userRepo;
@Autowired
private AccountRepo accountRepo;
```

Step 6: Postman Agent to create objects on the database

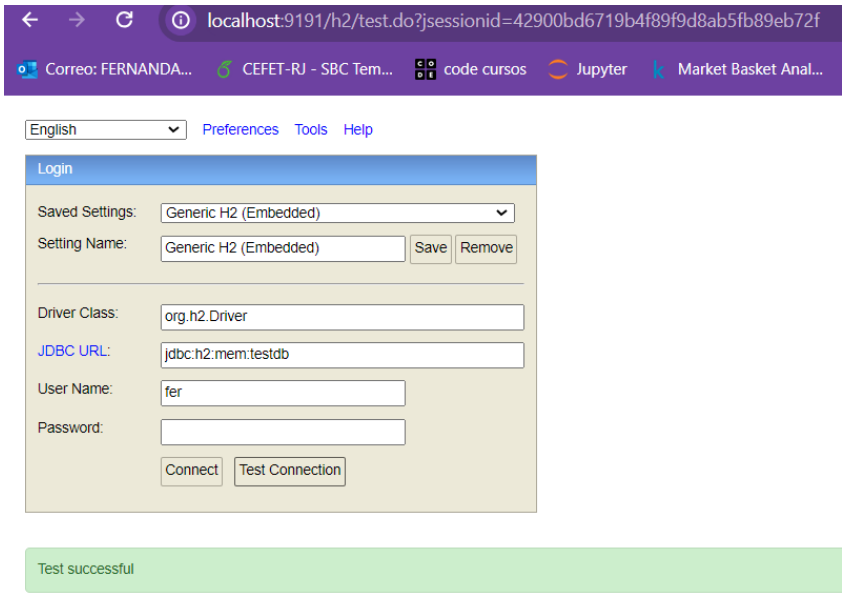
For this example I have used Postman Agent API to generate the user and accounts data, just to have information to query and shows the execution for the Job Application.



Step 7: Query on H2 DB using PGA

H2 works as SQL Server.

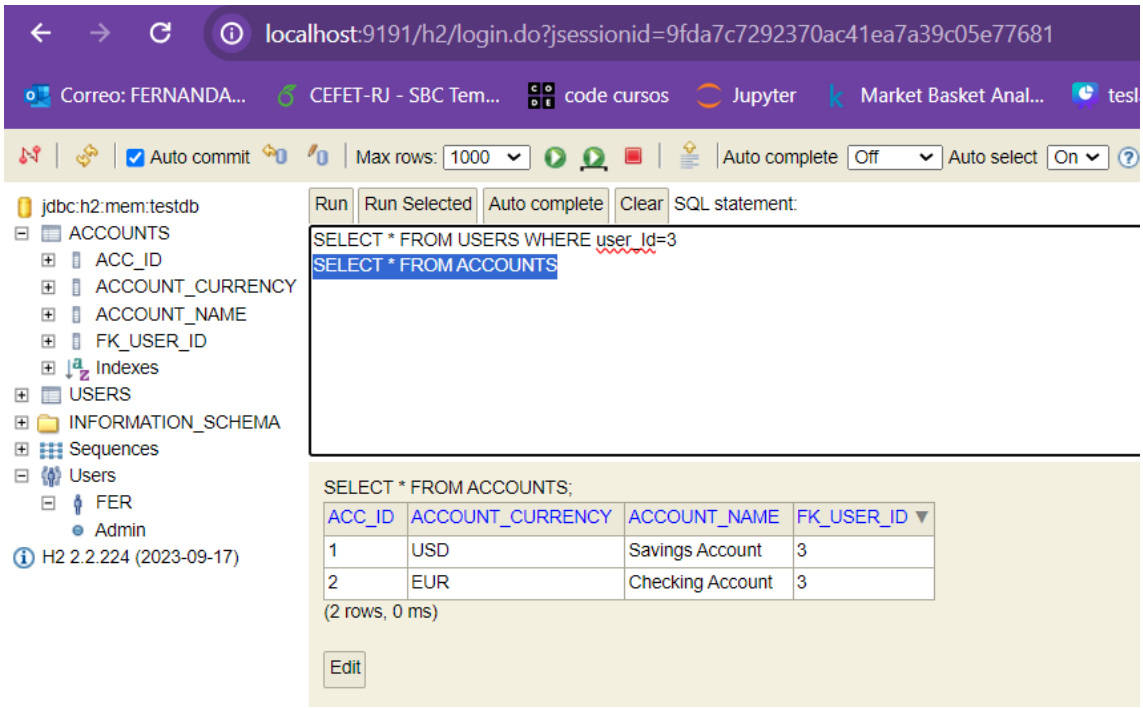
The first step is to Test Connection:



And Log In with the credentials declared on *application.properties* file:

```
# For Data Base with H2
# JDBC is thee tool for connectivity -> H2
# Database name in memory -> testdb
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driverClassName=org.h2.Driver
#credentials to access
spring.datasource.username=fer
spring.datasource.password=


# For queries with JPA and SQL, such as the terminal
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.H2Dialect
spring.jpa.hibernate.ddl-auto=update
spring.h2.console.enabled=true
spring.h2.console.path=/h2
```

Once we have logged, we can make the queries on the H2 database, for example:



Swagger API

Add the next depend on *pom.xml* file:

```
<!--Swagger API dependy-->
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
    <version>2.5.0</version>
</dependency>
```

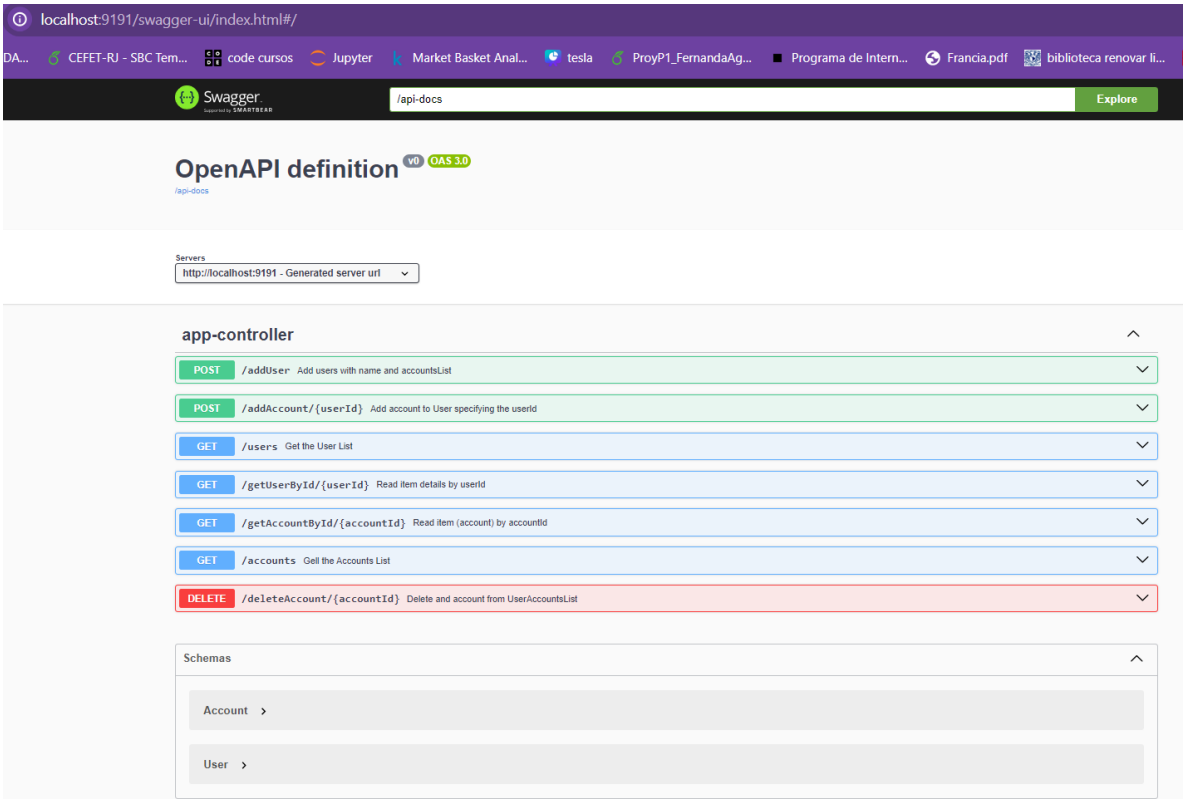
The, if we would like to customize our URL to access to the API we can add the next line on *application.properties* file:

```
#--------Swagger-----------
springdoc.api-docs.path=/api-docs
```

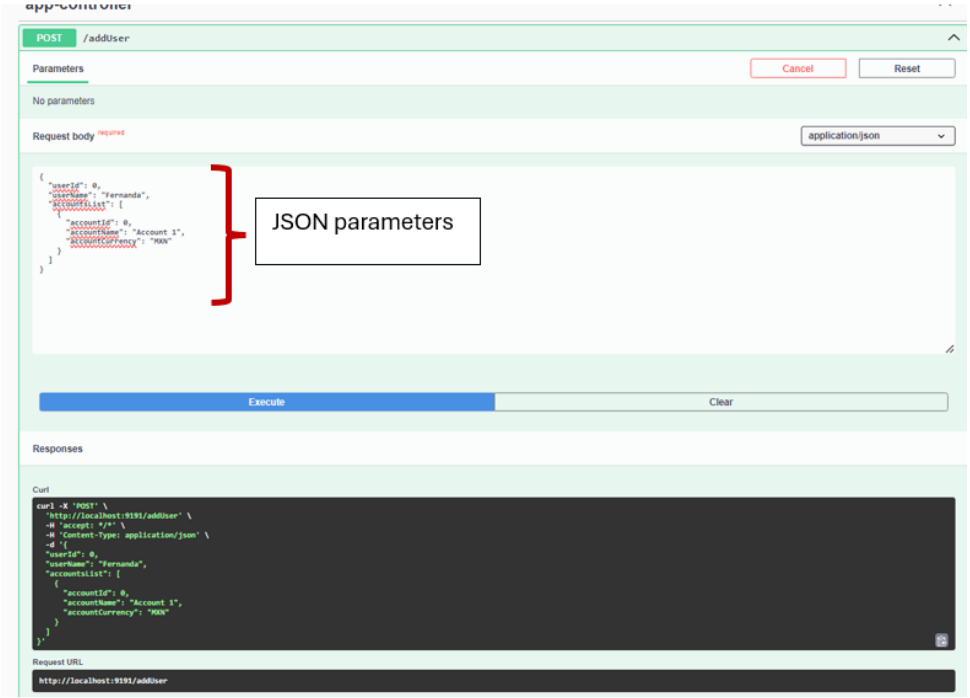
So, the URL is the following:

http://localhost:9191/swagger-ui/index.html

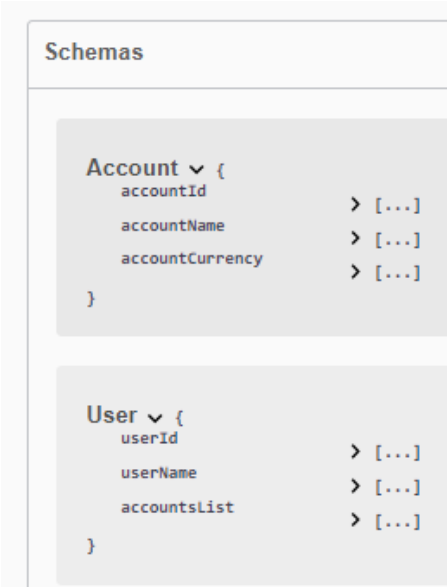
The catalog with the documentation with Swagger API is the next one, as we can see there are the HTTP methods which exists in the REST Controller, and



Finally, we can execute these operations, for example a POST method with */addUser*



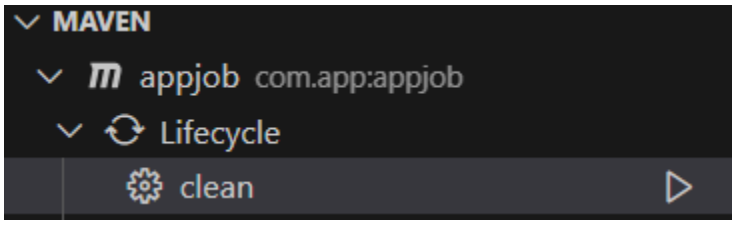
Also, we can see the schemas (models) with their attributes:



Containerize spring boot app using docker in Windows OS

Clean the project with *Maven*

The first step is Clean the project with *Maven*:

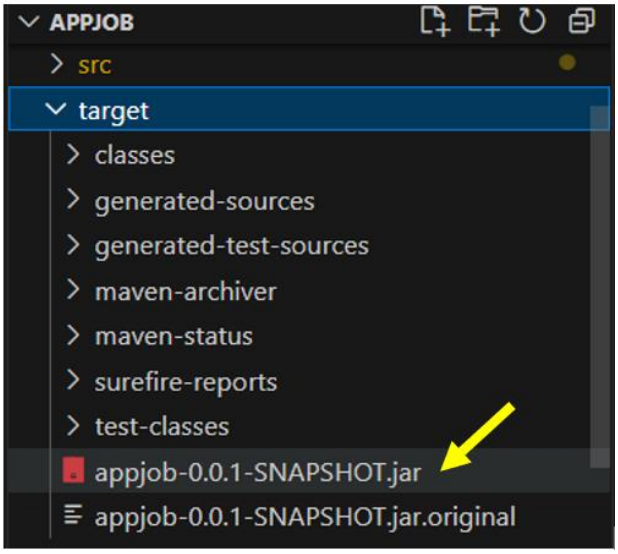


The a part of the output is the following:

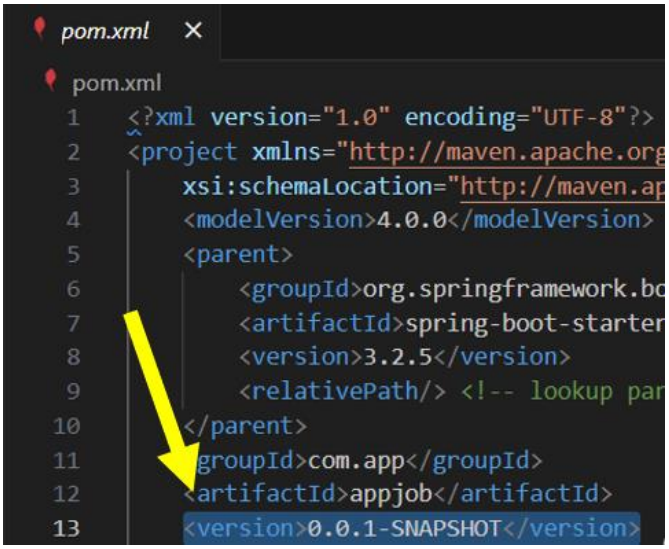
```
[INFO] --- clean:3.3.2:clean (default-clean) @ appjob ---
Downloading from central: https://repo.maven.apache.org/maven2/org/codehaus/plexus/plexu
s-utils/4.0.0/plexus-utils-4.0.0.pom
Downloaded from central: https://repo.maven.apache.org/maven2/org/codehaus/plexus/plexus-utils/4.0.0/plexus-utils-4.0.0.pom (8.7 kB at 49 kB/s)
Downloading from central: https://repo.maven.apache.org/maven2/org/codehaus/plexus/plexus/13/plexus-13.pom
Downloaded from central: https://repo.maven.apache.org/maven2/org/codehaus/plexus/plexus/13/plexus-13.pom (27 kB at 133 kB/s)
Downloading from central: https://repo.maven.apache.org/maven2/org/codehaus/plexus/plexus-utils/4.0.0/plexus-utils-4.0.0.jar
Downloaded from central: https://repo.maven.apache.org/maven2/org/codehaus/plexus/plexus-utils/4.0.0/plexus-utils-4.0.0.jar (192 kB at 262 kB/s)
[INFO] Deleting c:\Users\Win10\Desktop\JobProject\ProjectLSEG\appjob\target
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  5.036 s
[INFO] Finished at: 2024-05-20T20:14:35-06:00
```

Create the JAR file with *Maven*

The second step is Install with *Maven* to create the *JAR* file in the directory *target*



And has the same name as the version of our project *appjob*



Install wls2

The next step is install *wls2* (in the last Windows OS wls2 is default) which is a Windows' feature called *Windows Subsystem for Linux*. This feature allows us to run a Linux environment on our Windows OS, without any VM or booting.

Having said this, to install it we can run the next command on the CMD:

```
wls --install
```

In my case I only need to update it, so I ran this command:

```
wls --update
```

```
C:\Users\Win10>wsl --update
Instalando: Subsistema de Windows para Linux
Se ha instalado Subsistema de Windows para Linux.
```

Now, if we check the *wls* version with the next command:

```
wls -version
```

We can see that by default we have wsl version 2 (wsl2) in our Windows OS.

```
C:\Users\Win10>wsl --version
Versión de WSL: 2.1.5.0
Versión de kernel: 5.15.146.1-2
Versión de WSLg: 1.0.60
Versión de MSRDC: 1.2.5105
Versión de Direct3D: 1.611.1-81528511
Versión DXCore: 10.0.25131.1002-220531-1700.rs-onecore-base2-hyp
Versión de Windows: 10.0.19045.4412
```

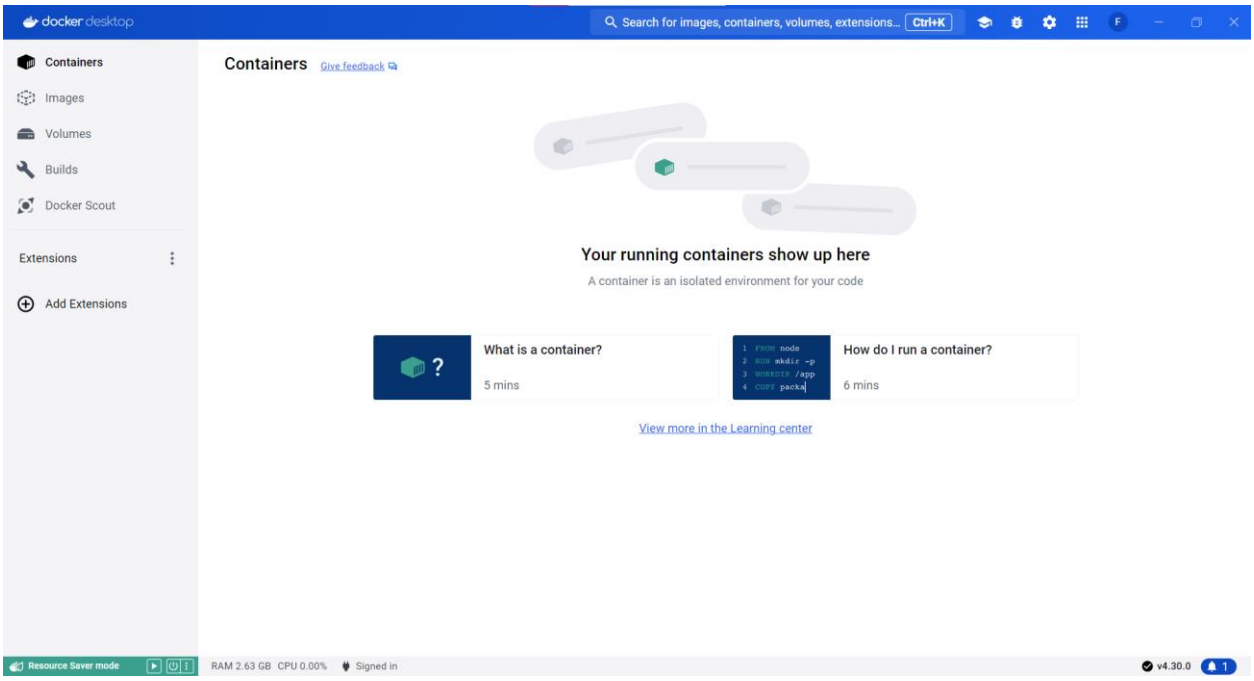
Install Docker Desktop

Then, we must install Docker Desktop from Docker Hub: https://hub.docker.com/

And we can check the version on the CMD:

```
C:\Users\Win10>docker --version
Docker version 26.1.1, build 4cf5afa
```

At this point, run the *Docker Desktop* and you will see the next screen, it means the Docker daemon is running, so you can execute any Docker command:

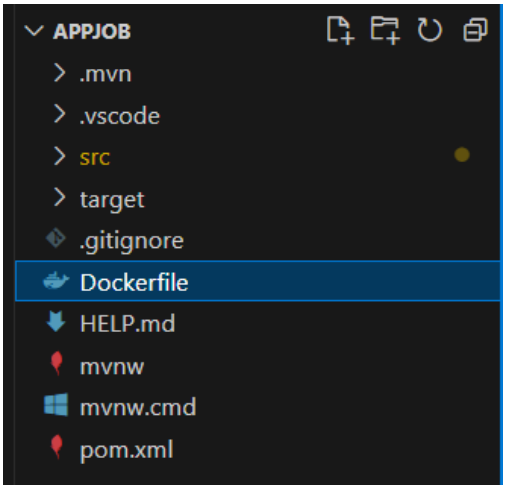


Create a Docker image

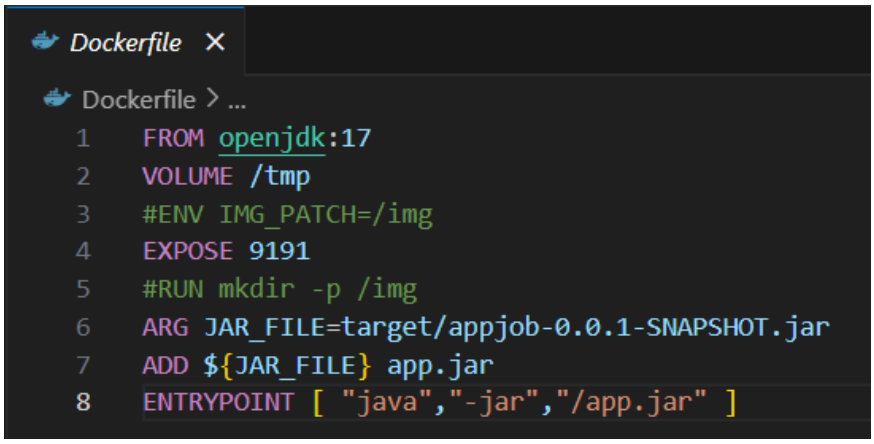
Create a Dockerfile

We must create a Docker file to let Docker knows which Docker image it can use to make our new image, where it can store data, where is the *.jar* file, and so on.

So, at the same level of the directory *target* of our project create the file:



Its content is the following:



And the meaning of each keyword is the following:

- *FROM*
  Docker makes use of a Docker image that already exists to create the next image. In this case the root image is openjdk:17 because we have installed the Java version 17.
- *VOLUME*
  Specify the directory where Docker can store the temporal data, for then store it on the container.
- *EXPOSE*
  The port in our case we have configured the port 9191.
- *ARG*
  Variables which are useful in the execution time.
- *ADD*
  Add information to the container.
- *ENTRYPOINT*
  What is going to execute and how.

Use .jar file to create the image

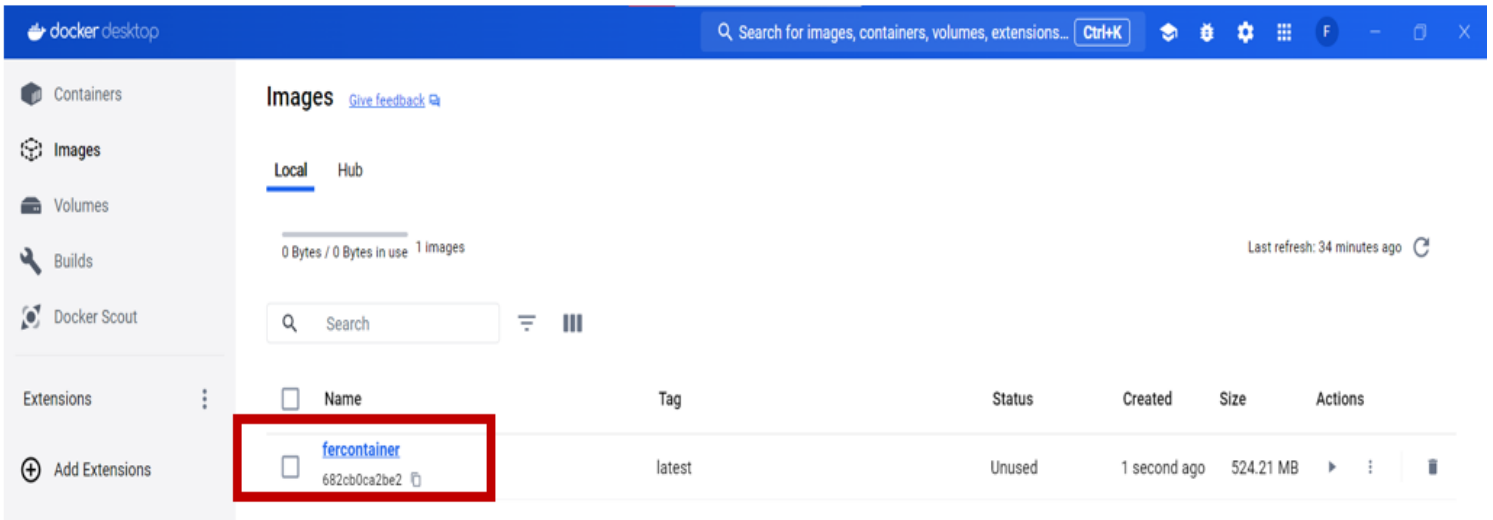
At this point and while our *Docker Desktop is running* execute the next command on your Terminal:

```
docker build -t [nameContainer] .
```

The output is the following, so we have created our image in the container called *fercontainer*

```
PS C:\Users\Win10\Desktop\JobProject\ProjectLSEG\appjob> docker build -t fercontainer .
[+] Building 112.5s (8/8) FINISHED
 => [internal] load build definition from Dockerfile
 => => transferring dockerfile: 235B
 => [internal] load metadata for docker.io/library/openjdk:17
 => [auth] library/openjdk:pull token for registry-1.docker.io
 => [internal] load .dockerignore
 => => transferring context: 2B
 => [internal] load build context
 => => transferring context: 52.76MB
 => [1/2] FROM docker.io/library/openjdk:17@sha256:528707081fdb9562eb819128a9f85ae7fe000e2fbaeaf9f87662e7b3f38cb7d8
 => => resolve docker.io/library/openjdk:17@sha256:528707081fdb9562eb819128a9f85ae7fe000e2fbaeaf9f87662e7b3f38cb7d8
 => => sha256:528707081fdb9562eb819128a9f85ae7fe000e2fbaeaf9f87662e7b3f38cb7d8 1.04kB / 1.04kB
 => => sha256:98f0304b3a3b7c12ce641177a99d1f3be56f532473a528fda38d53d519cafb13 954B / 954B
 => => sha256:5e28ba2b4cdb3a7c3bd0ee2e635a5f6481682b77eabf8b51a17ea8bfe1c05697 4.45kB / 4.45kB
 => => sha256:38a980f2cc8accf69c23deae6743d42a87eb34a54f02396f3fcfd7c2d06e2c5b 42.11MB / 42.11MB
 => => sha256:de849f1cfbe60b1c06a1db83a3129ab0ea397c4852b98e3e4300b12ee57ba111 13.53MB / 13.53MB
 => => sha256:a7203ca35e75e068651c9907d659adc721dba823441b78639fde66fc988f042f 187.53MB / 187.53MB
 => => extracting sha256:38a980f2cc8accf69c23deae6743d42a87eb34a54f02396f3fcfd7c2d06e2c5b
 => => extracting sha256:de849f1cfbe60b1c06a1db83a3129ab0ea397c4852b98e3e4300b12ee57ba111
 => => extracting sha256:a7203ca35e75e068651c9907d659adc721dba823441b78639fde66fc988f042f
 => [2/2] ADD target/appjob-0.0.1-SNAPSHOT.jar app.jar
 => exporting to image
 => => exporting layers
 => => writing image sha256:682cb0ca2be2a6b4aaf4a0b7bb5ecfc20bf55fcc5107e219a6d4ce04a358dd89
 => => naming to docker.io/library/fercontainer

What's Next?
  View a summary of image vulnerabilities and recommendations → docker scout quickview
```

We can see the image generated on *Docker Desktop*:



Or from the Command Line:

```
PS C:\Users\Win10\Desktop\JobProject\ProjectLSEG\appjob> docker image ls
REPOSITORY      TAG       IMAGE ID       CREATED             SIZE
fercontainer    latest    682cb0ca2be2   37 minutes ago      524MB
```

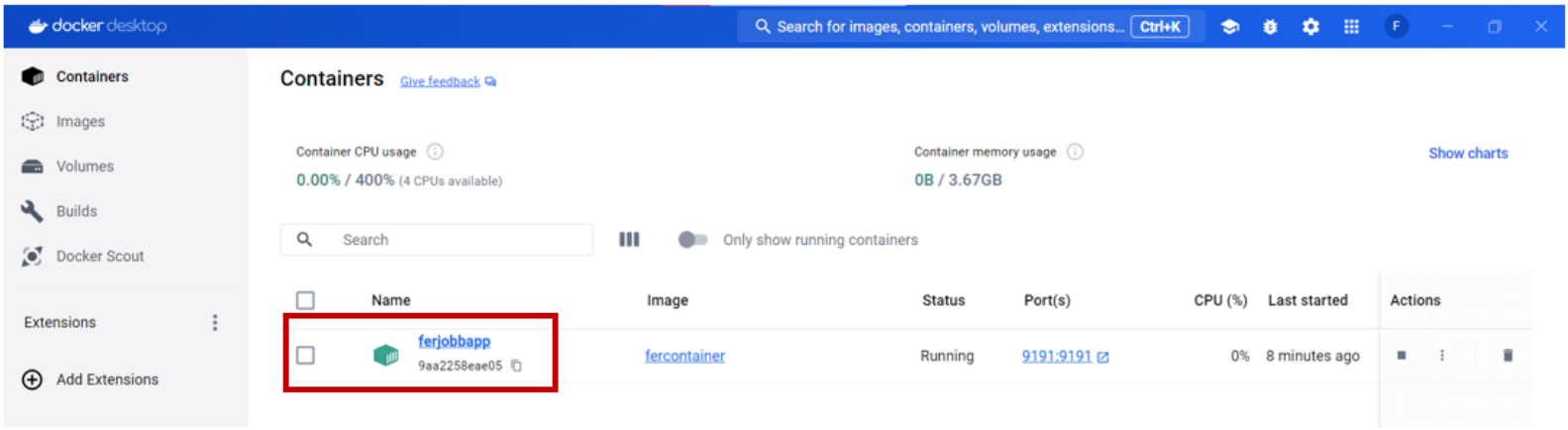
Create Docker container

Execute the image to build a container with next command:

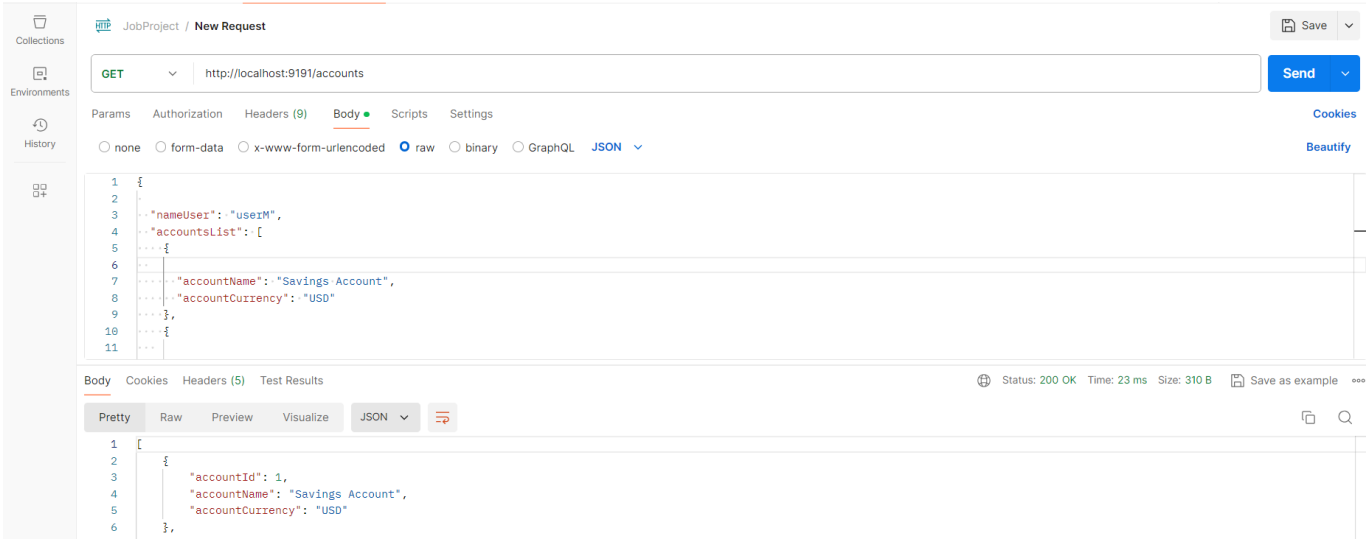
```
docker run -p[containerPort]:[machinePort] -name [nameContainer]
[nameImagetoRun]
```

```
PS C:\Users\Win10\Desktop\JobProject\ProjectLSEG\appjob> docker run -p9191:9191 --name ferjobbapp fercontainer

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::                (v3.2.5)

2024-05-21T14:57:13.766Z  INFO 1 --- [appjob] [           main] com.app.appjob.AppjobApplication         : Starting AppjobApplication v0.0.1-SNAPSHOT using Ja
oot in /)
2024-05-21T14:57:13.785Z  INFO 1 --- [appjob] [           main] com.app.appjob.AppjobApplication         : No active profile set, falling back to 1 default pr
2024-05-21T14:57:16.720Z  INFO 1 --- [appjob] [           main] .s.d.r.c.RepositoryConfigurationDelegate : Bootstrapping Spring Data JPA repositories in DEFAU
2024-05-21T14:57:16.869Z  INFO 1 --- [appjob] [           main] .s.d.r.c.RepositoryConfigurationDelegate : Finished Spring Data repository scanning in 125 ms.
2024-05-21T14:57:18.248Z  INFO 1 --- [appjob] [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port 9191 (http)
2024-05-21T14:57:18.275Z  INFO 1 --- [appjob] [           main] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2024-05-21T14:57:18.276Z  INFO 1 --- [appjob] [           main] o.apache.catalina.core.StandardEngine    : Starting Servlet engine: [Apache Tomcat/10.1.20]
```

Finally, we can see that our container is running on the *Docker Desktop*:



We can verify the application on Postman or in another tool:



Resources

GitHub link

https://github.com/FerLovelace/JobSolution.git

Docker Hub for docker image

https://hub.docker.com/r/fernandaaguilarcorona180513/fercontainer

Notes

In this project the used port was 9191 from localhost and was developed with Visual Studio Code.