**Actividades por desarrollar para continuar con el proyecto final del curso:**

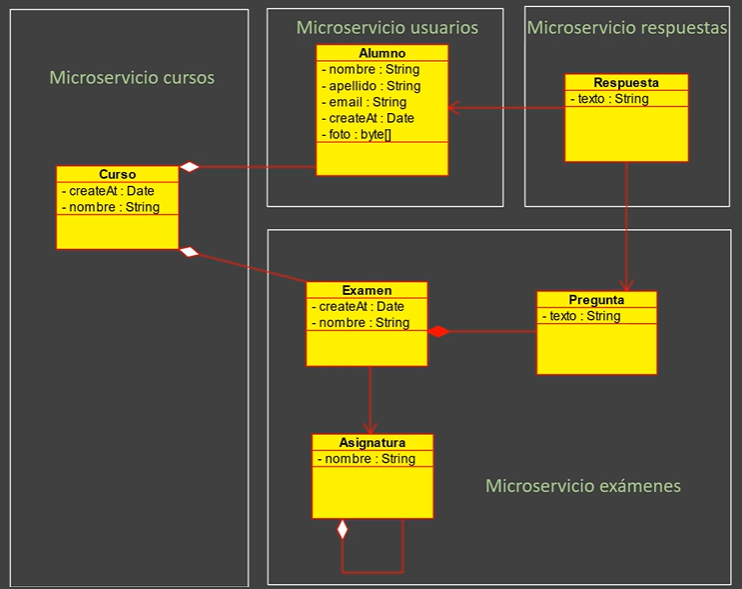
En esta parte del proyecto se desea que usted ponga en juego los conocimientos adquiridos hasta aquí, teniendo como base los pasos anteriores que han llevado al proyecto hasta el punto en el que este momento se encuentra.

Ahh y una cosa más, igual como en su momento lo hizo el docente, usted debe colocar, las evidencias en capturas de pantalla en el documento que adjunte como entrega de la actividad

Estas son las actividades a desarrollar en esta parte del proyecto

1. Crear un nuevo servicio con el nombre de **Respuesta** con sus respectivas relaciones por un lado con Pregunta y por el otro con Alumno

de acuerdo con el siguiente diagrama, el cual se encuentra en el primer taller de esta serie



1. Dentro del servicio centralizado de entidades debe colocar las entidades:

**Exámenes**

**Preguntas**

1. La clase exámenes debe contener los siguientes atributos:

**private** Long id;

**private** String nombre;

**private** Date createAt;

1. La clase pregunta debe contener los siguientes atributos:

**private** Long id;

**private** String texto;

1. Ambas clases deben tener la misma estructura que tienen las otras clases entities que se encuetran en el servicio centralizado
2. Se deben añadir los siguientes bloques de código el cual permite establecer una relación bidireccional entre las dos entidades creadas

**En la clase Examen**

## @JsonIgnoreProperties(value = {“examen”}, allowSetters = true)

## @OneToMany(mappedBy = “examen” , fetch = FetchType.LAZY, cascade = CascadeType.ALL, orphanRemoval = true)

**private** List<Pregunta>preguntas;

**y en la clase Pregunta:**

## @JsonIgnoreProperties(value = {“preguntas”})

## @ManyToOne(fetch = FetchType.LAZY)

@JoinColumn(name=”examen\_id”)

**private** Examen examen;

1. Deben generar los getter y setter
2. Y deben crear en la clase pregunta el siguiente método que permite agregar una pregunta a un examen

**public** **void** setPreguntas(List<Pregunta> preguntas) {

**this**.preguntas.clear();

preguntas.forEach(**this**::addPregunta);

}

**public** **void** addPregunta(Pregunta pregunta) {

**this**.preguntas.add(pregunta);

pregunta.setExamen(**this**);

}

**public** **void** removePregunta(Pregunta pregunta) {

**this**.preguntas.remove(pregunta);

pregunta.setExamen(**null**);

}

1. Debe crear todas las capas para la exposición del servicio (service, controller, repository), eso sí, haciendo uso de los servicios transversales que ya se tienen**.**
2. Luego debe matricular ese servicio en el servidor de nombre de Eureka como lo hemos hecho con los otros servicios.
3. Luego debe exponer el servicio a través del servicio Gateway que ya tenemos tal y como lo hicimos con los otros servicios, pero para que funcione solo lo levantamos expuesto por Eureka, por que creo que el problema que se tiene para probar los servicios esta con el Gateway.
4. Finalmente debe probar el servicio.
5. Instalar y correr Docker Desktop.
6. A través del siguiente proceso se debe desplegar el servicio construido por medio de Docker
7. El proceso es el siguiente. Iremos a la raíz de cada uno de los proyectos (o el directorio que queramos, pero en este ejemplo trabajaremos en el raíz), y en ella crearemos un archivo llamado Dockerfile. En este archivo, definiremos tres propiedades, aunque el mundo de Docker va mucho más allá, este ejemplo básico nos servirá para poder ejecutar la aplicación. Este Dockerfile será el de Eureka:

FROM openjdk:8-jdk-alpine

ADD target/DockerSpringNetflix-EurekaServer-0.0.1-SNAPSHOT.jar app.jar

ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-jar","/app.jar"]

La propiedad **FROM** es la imagen base que vamos a tomar, que será la imagen openjdk:8-jdk-alpine, constituida por una distribución de Linux extremadamente ligera y un jdk para ejecutar aplicaciones Java.

**ADD** nos servirá para crear un directorio en el que alojaremos el ejecutable de cada una de nuestras aplicaciones.

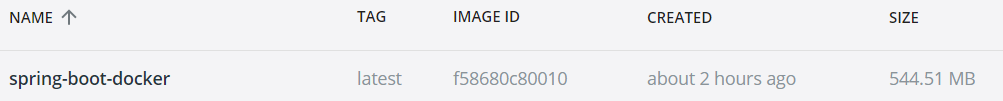
Y **ENTRYPOINT** será el comando que se ejecute cuando se levante el contenedor con esta imagen, el cuál ejecutará nuestra aplicación usando el fichero .jar.

## **Crear contenedor a partir de Dockerfile**

* Abre una **Windows PowerShell** y sitúate en el directorio principal de tu proyecto e introduce:

D:\wsSpringBoot\SBServicesRest> **docker build -t spring-boot-docker .**

* Comprueba la imagen creada: Abre la aplicación de Windows Docker Desktop y observa que se ha añadido la nueva imagen:

[](https://developrogramming.com/wp-content/uploads/2021/09/image-6.png)

* También puedes listar las imágenes por línea comandos con:

**docker images**

[Texto

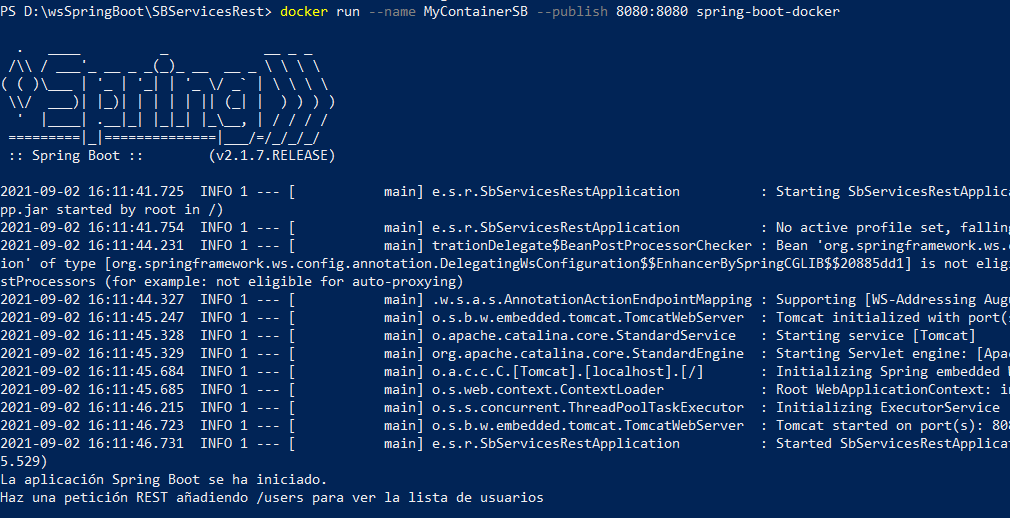
Descripción generada automáticamente](https://developrogramming.com/wp-content/uploads/2021/09/image-7.png)

## **Ejecutar contenedor Docker**

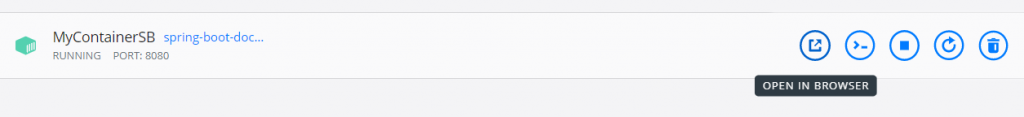
* Abre una **Windows PowerShell** y sitúate en el directorio principal de tu proyecto e introduce:

**docker run –name MyContainerSB –publish 8080:8080 spring-boot-docker**

* Observa que se inicia la aplicación:

[](https://developrogramming.com/wp-content/uploads/2021/09/image-8.png)

* En la aplicación Docker Desktop también debe aparecer:

[](https://developrogramming.com/wp-content/uploads/2021/09/image-9.png)Aplicación SpringBoot Java ejecutándose en nuestro contenedor

Eso es todo por ahora

**EXITOS**