CLICLER – COMPETITION

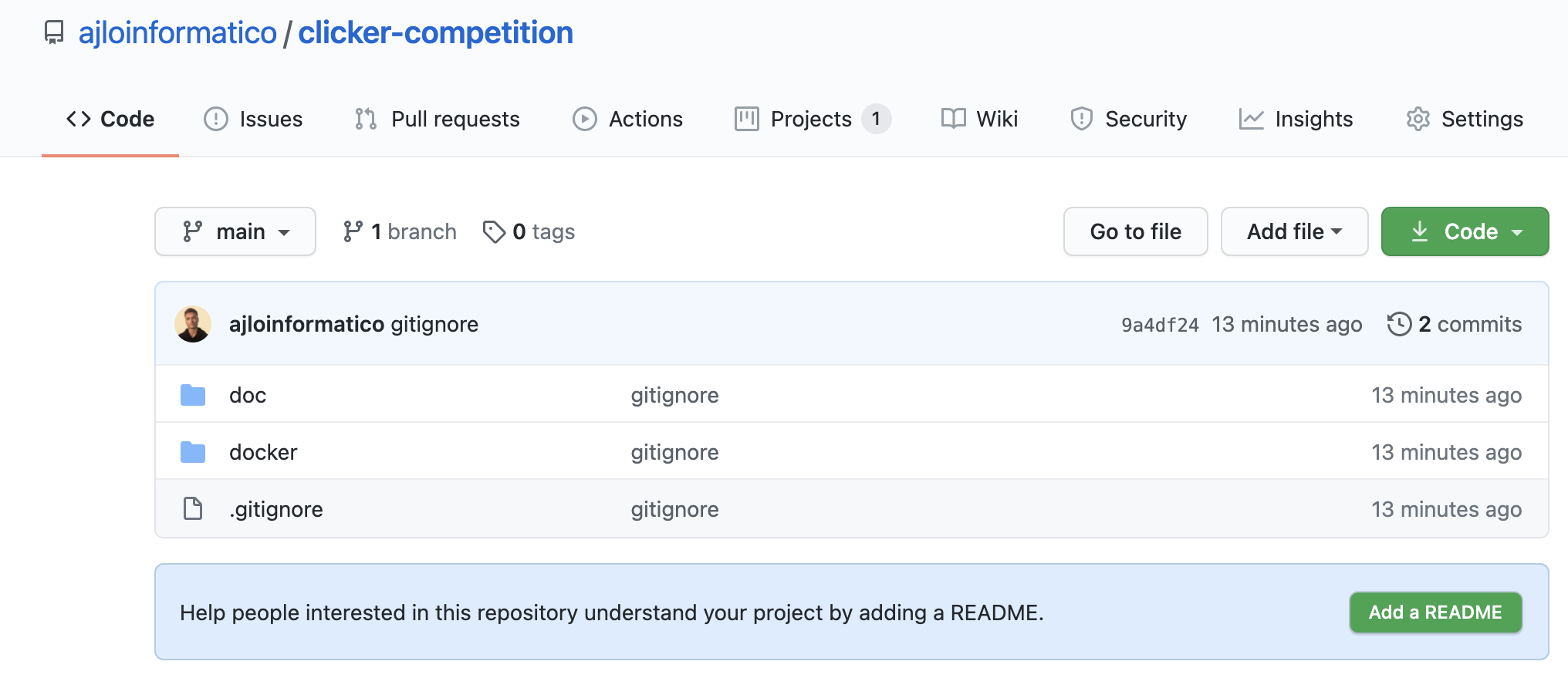
Construcción del proyecto.

En este apartado incluiré las acciones realizadas hasta la ejecución de **spring**

# GITHUB

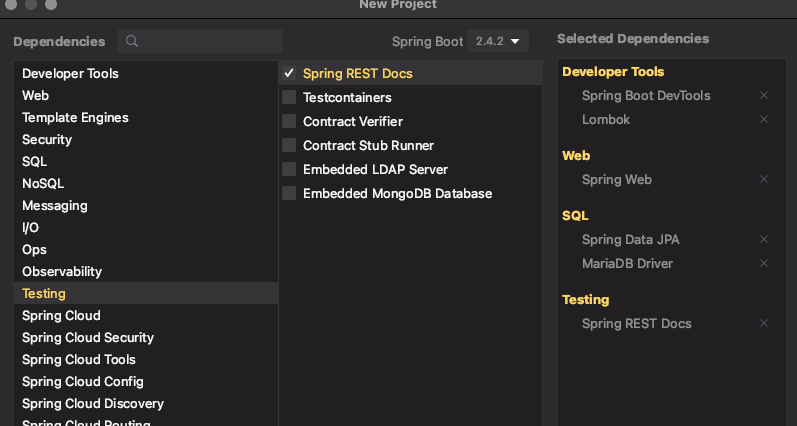
En **GitHub** he creado un respositorio sobre el que crearé el proyecto **Spring**

El nombre del **proyecto** será **clicker-competiton**.



# INTELLI J & SPRING BOOT

Para la construcción del proyecto se ha usado la propia herramienta de **IntelliJ** **Spring Boot para instalar todas las dependencias del framework.**



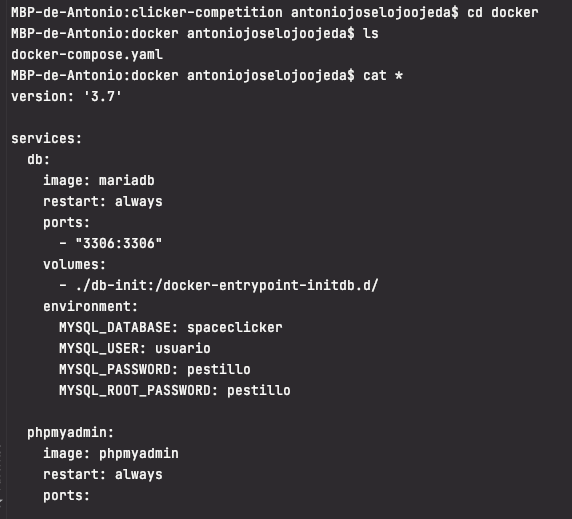
El nombre para

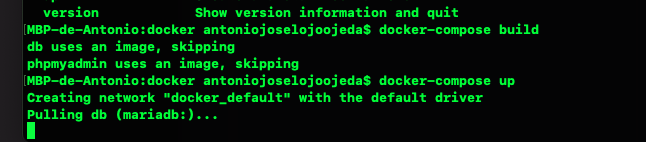
Base de datos:

Importante crear la **base de datos** antes de ejecutar **la aplicación.** Ya que en **aplicaction.properties** se le indica la **url** de la base de datos. **Spring** no crea solo la **base de datos**. La solución más óptima es incluir la **creación de la base de datos** en un **entryPoint de docker**.

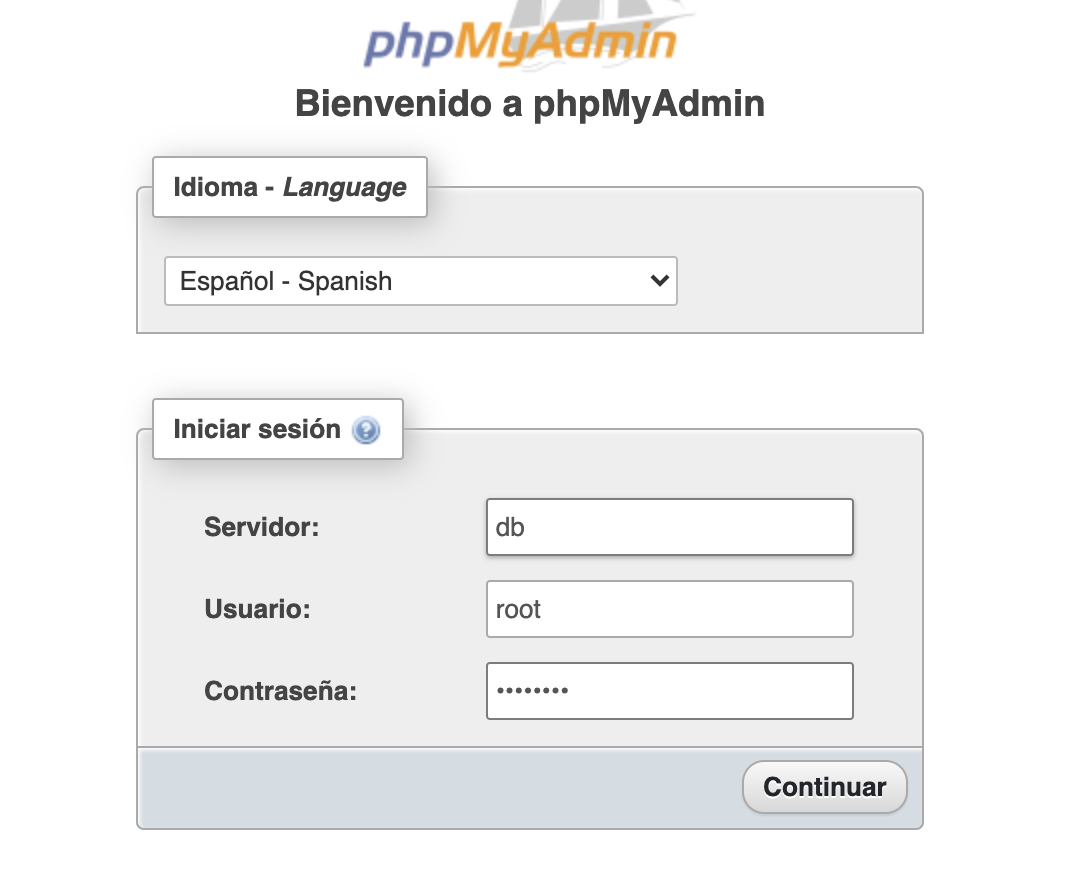
**//TODO incluir imagen del entry point**

La base de datos está montada con un contenedor **docker** con el puerto **3306**.y un segundo **contenedor** por el puerto **8081.** Con el cual se podrá crear y modificar la base de datos. El archivo **docker-compose .yaml** estará ubicado ubicado en **./docker**

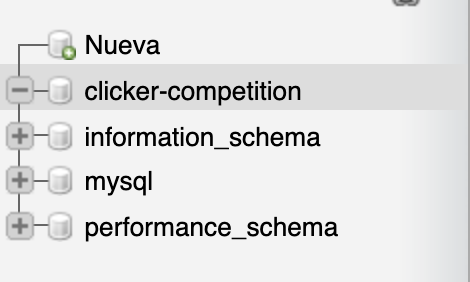




Una vez levantados los **contenedores creo la base de datos desde** phpmyadmin: <http://localhost:8081>



Las **claves** son las indicadas en el **docker-compose.yaml** . (db, root, pestillo).



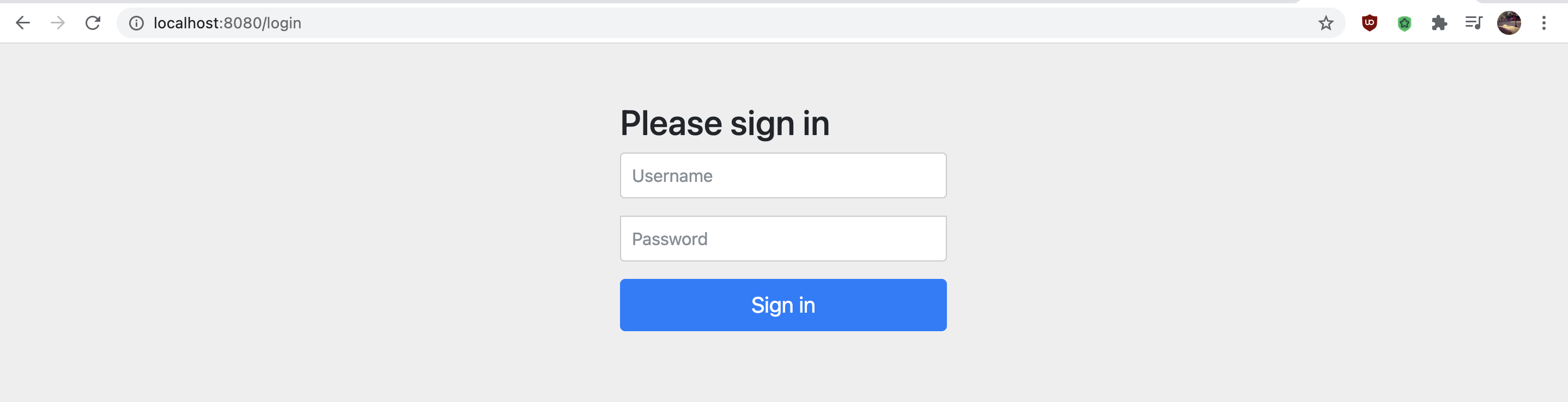
# Configuración de la base de datos

La configuración de la base de datos se agrega en **/src/main/Resources/Application.properties**



A continuación arranco el proyecto para comprobar que funciona.

Http:8080/login



Security

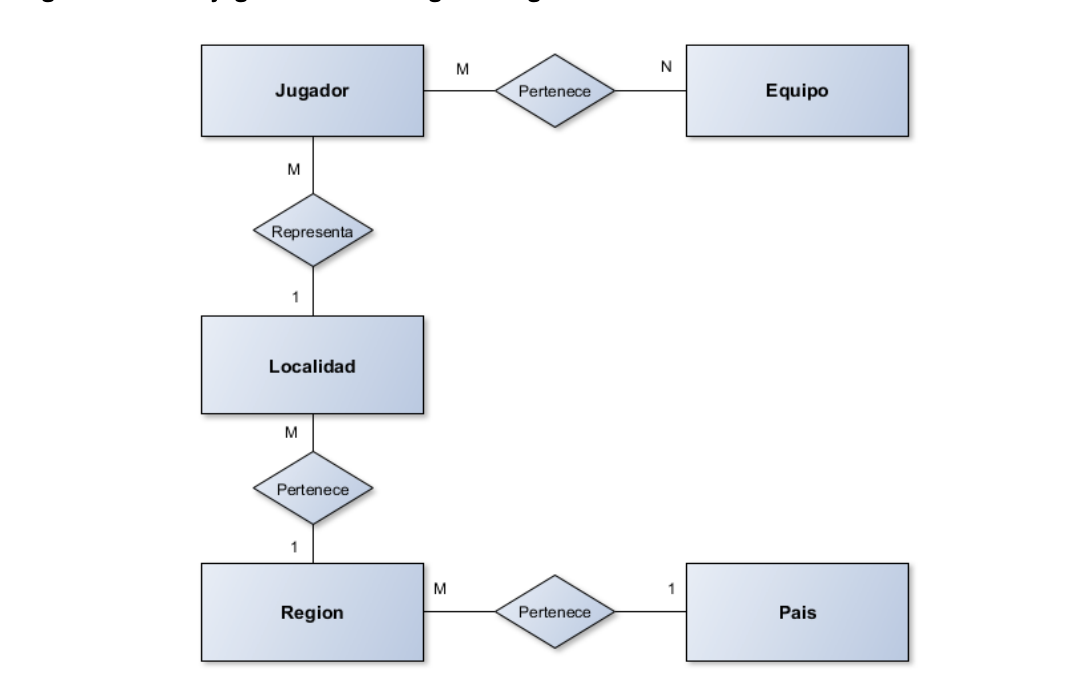
Para esta parte vamos a **crear una clase** para **la seguridad** y una **segunda clase** que será **el servicio.**

Comenzamos creando un **paquete** **services** con una clase que será **UserDetailService**

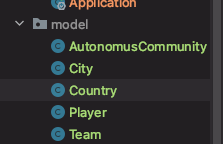
Entidades (Entities).

El proyecto se constituye de 5 entidades **(Jugador, Equipo,Localidad,Región, País).**

Estas se crearán en un paquete **Model.**

****

La estructura de las entidades de la aplicación resultante es la siguiente:



# AuthonomusCommunity

**TODO: Explicar la clase**

# City

**TODO: Explicar la clase**

# Country

**TODO: Explicar la clase**

# Player

**TODO: Explicar la clase**

# Team

**TODO: Explicar la clase**

Repositorios (Repositories)

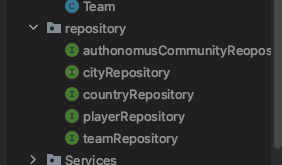
Datos, Está **compuesto de** un **paquete** que crearé con el nombre **repository**. Dentro de este paquete crearemos **6 interfaces** con el nombre **(modeloRepository).** Cada una de ellas haciendo referencia a cada uno de los modelos anteriores. Estas interfaces deben extender de **CrudRepository** y su tipo de dato debe ser el mismo que el de las clases a las que hacen referencia

Ejemplo: **countryRepository**

****

de manera automática cuentan ya con ciertos métodos para hacer consultas sobre la base de datos.  **Ejemplo findById()**

El resultado sería el siguiente:



# authonomousCommunityRepository

**TODO: EXPLICAR**

# cityRepository

**TODO: EXPLICAR**

# countryRepository

**TODO: EXPLICAR**

# playerRepository

**TODO: EXPLICAR**

# teamRepository

**TODO: EXPLICAR**

Relaciones entre las entidades

Continuando con el proceso de desarrollo del **clicker** el siguiente punto será crear las relaciones entre las entidades. Esto se hará mediante **anotaciones**.

# Country – AuthonomousCommunity (Many to one)

**Una relación Uno a Muchos:** Se realizará de la siguiente manera.

Como ejemplo **Country-AuthonomousCommunity**.

## Uno a muchos

Un país tiene varias comunidades autónomas

Se indica la relación con la anotación **@OneToMany()**. Aquí se le indica mediante dos parámetros sobre que tabla se realiza la relación y de que tipo será la foránea.



**A** continuación se indica un **índice ficticio** sobre el que se va ha realizar la relación con **comunidades autónomas.**



Además para evitar errores se indican dos anotaciones:

**@JsonBackReferenece** 🡪 Con esta anotación se evita que el json de respuesta se envíe anidado.

**@EqualsAndHash.Exclude 🡪** Es necesario ignorar los hashes para que funcione correctamente. Es necesario . El resultado sería el siguiente.



# AuthonomousCommunity – Country (One to Many)

## Muchos a Uno

En la parte de la entidad de comunidades autónomas **AuthonomousCommunity** la relación sería **de muchos a uno con Country** **@ManyToOne**

Habría que indicarle **@ManyToOnea** al contrario que en **Country**. (Por que es la segunda).



**@EqualsAndHashCode.Exclude** 🡪 Para evitar problemas.

**@JoinColumn() 🡪** Indicia el campo de unión (que es el indicado n **Country** como ficticio). Hace de **FK** . Este debe ser un atributo **privado** de la relación “hija”. Y debe estar incluido en alguno de los constructores.



Y estaría la relación **uno a muchos**  entre Country & AuthonomusCommunity y **muchos a uno** entre AuthonomusCommunity lista. **Las siguientes relaciones serán iguales exceptuando Player & Team** . Que será relación **@ManyToMany**.

# AuthonomousCommunity - City (One to Many)

**TODO: EXPLICAR**

# City – AuthonomousCommunity (Many to one)

**TODO: EXPLICAR**

# City – Player (One to Many)

**TODO: EXPLICAR**

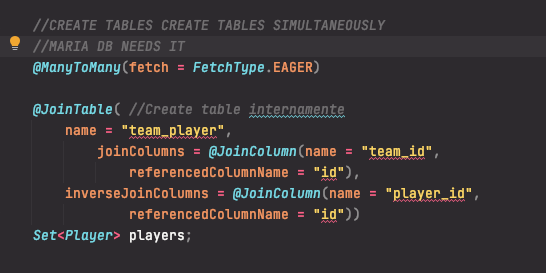
# Player – City (Many to One)

**TODO: EXPLICAR**

# Player – Team (Many to Many)

# Team – Player (Many to Many)

En las relaciones **Many to Many** . Es aconsejable indicar el nombre de la tabla intermedia así como de las columnas que serán las foráneas de ambas. Valorando la relación el padre de la relación será **Team.** Ya que primero se creará el **Team**, para que después **se agreguen** manualmente los **players**

****

Además con **Maria db** es necesario indicarle en @**ManyToMany** de la entidad padre el tipo **fetch = FetchType.EAGER**

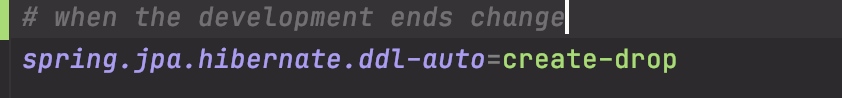
**iMPORTANTE** en el **Set<Player> players.** Es necesario indicar el **tipo** y el nombre (en este caso players) debe indicarse en el **mappedby**  del **@ManyToMany** de la entidad hija.

# Player-Team (Many To Many)

Esta sería la entidad hija. En ella hay que indicarle con **MappedBy** el valor del **Set** de la relación del padre.

Boostrap (Seeder)

El Proyecto se ha comenzado usando **una base de datos MariaDB** alojada en **docker**. En la configuración de **Spring** es**/lojo/boostrap/Seeder**. Se especifica que al arrancar se borren todos los datos (**simulando una base de datos en memoria H2**).Esta se hace mediante la siguiente línea:



**Tras** el **borrado**, insertamos los datos por defecto de nuestro **Seeder** en la clase **Seeder**

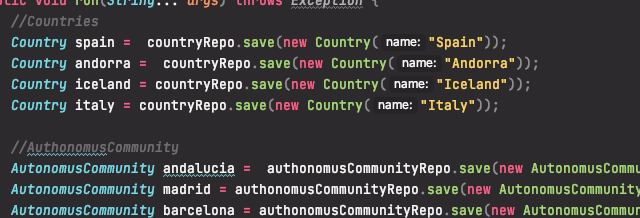
Para que los **Seeder** se ejecuten al inicio de la aplicación es necesario que **primero** se agrege la anotación **@Component** y que la propia clase herede de **CommandLineRunner**.



Usando **@Authowired** se indican los **repositorio** a importar para hacer los **inserts.**

****

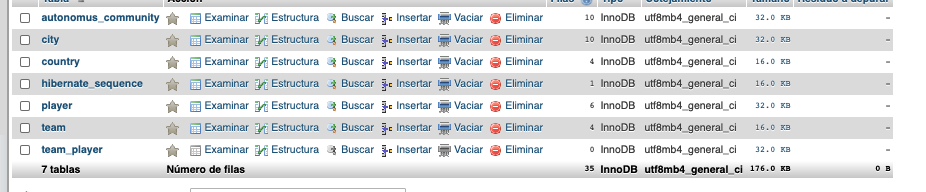
A continuación realizar las **inserciones** llamando a los repositorios instanciados.



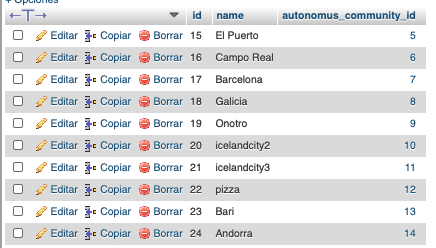
Importante respectar los constructores declarados en los **modelos**

Una ve creado el **Seeder** comprobar que los datos se han insertado correctamente accediendo al **contenedor con phpmyadmin**

[**http://localhost:8081/**](http://localhost:8081/)

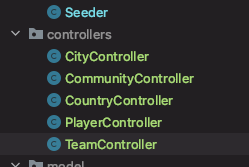
****

Accedemos a las distintas tablas y vemos que efectivamente los datos están correctamente insertados.



Controllers & Querys

A continuación creamos los **controladores**. Y toda la lógica de las **Querys**. Los controladores estarán en el **paquete Controllers,** se deberá agregar uno por cada entidad. Estos tendrán un **@Authowired** de cada repo.



equipoRepository.save(new Equipo("equipoTest", usuario).addMiembros(Arrays.*asList*(usuario, usuario2)));

