Tutorial Spring Boot

- **Herramientas necesarias**

  Java 8 o superior
  VSCode
  Postman

- **Instalación**

  Para ejecutar los proyectos de una manera rápida y sencilla instalamos las extensiones que se muestran en la Figura 1.

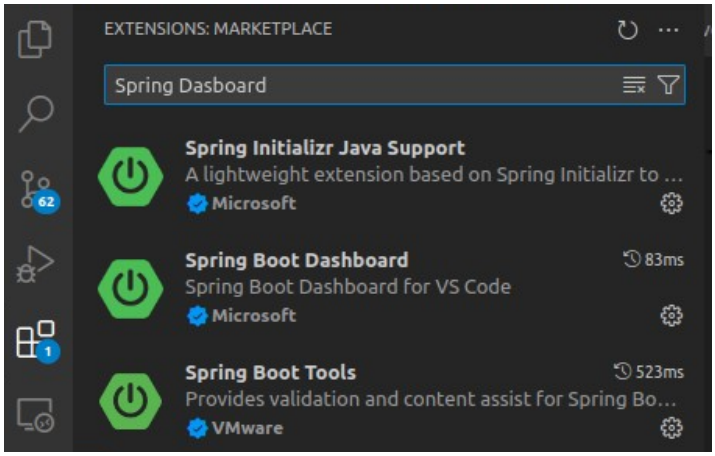


Figura 1.

Una vez instaladas se nos mostrara un icono con el logo de Spring Boot. Al hacer click sobre este podremos ver un apartado donde podremos ejecutar las aplicaciones desde la opción "Run" como se puede muestra la Figura 2.

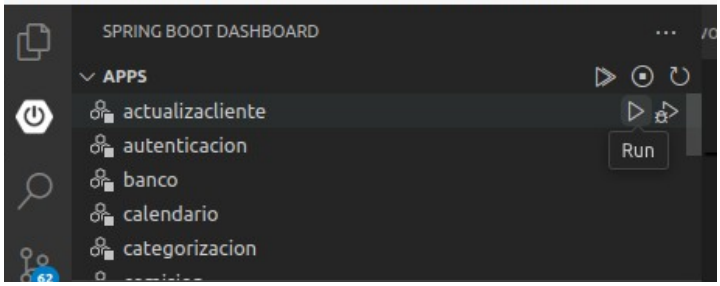


Figura 2.

- **Creación de un nuevo proyecto**
  Para iniciar un nuevo proyecto, se recomienda utilizar Spring initializer https://start.spring.io/ con las configuraciones que se muestran en la Figura 3.

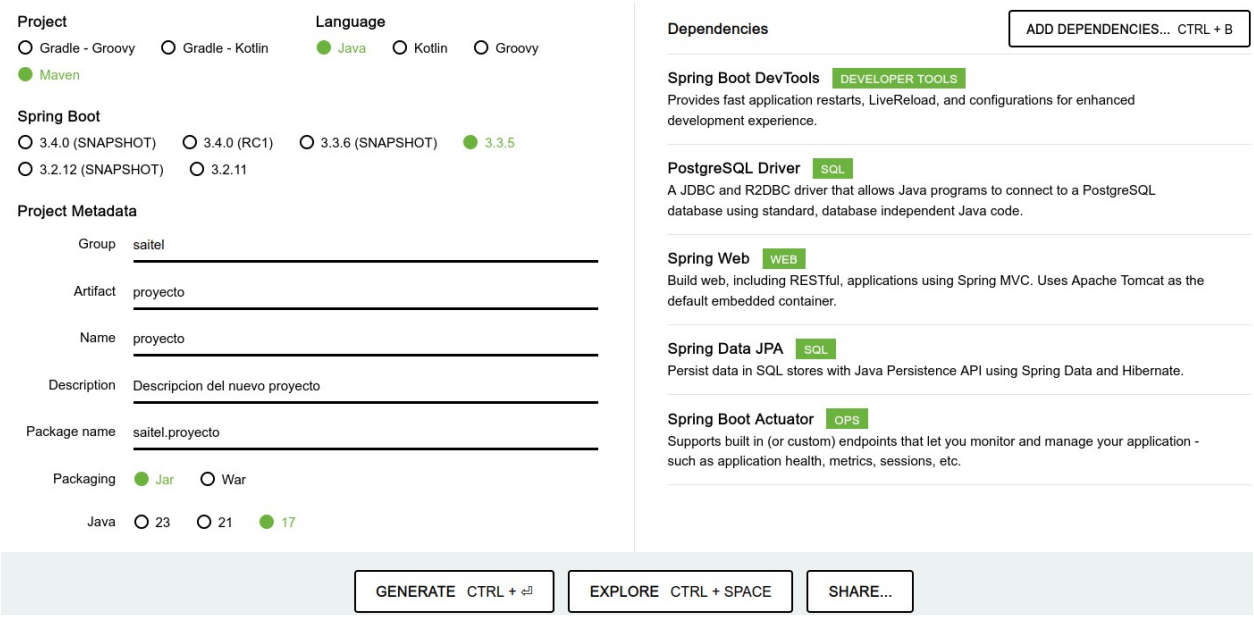


Figura 3.

En este caso usaremos maven con java 17 y la versión 3.3.5 de Spring Boot. Las dependencias necesarias son:

- **DevTools** para una recarga rápida de la aplicación durante el desarrollo
- **PostgrSQL** para abrir conexión a la base de datos
- **Spring Web** que incluye un servidor embebido para iniciar la aplicación
- **Data JPA** que nos brinda las operaciones CRUD
- **Actuator** que nos servirá para desplegar los proyectos en un servidor de kubernetes

Una vez realizada la configuración, hacemos click en “GENERATE” y se descargara un archivo el cual hay que descomprimirlo y agregarlo a nuestro espacio de trabajo en VSCode. Ademas, en caso de necesitar otra dependencia, se las puede agregar al archivo pom.xml que ya viene incluido en el proyecto anteriormente descomprimido.

- **Implementación de un nuevo proyecto**

- **Estructura**

El proyecto debe estar divido en varios directorios, como se muestra en la Figura 4, dentro de la carpeta main, entre ellos los que mas se usan son:

1. **Entity:** aquí se desarrollaran las clases que mapeen las tablas a utilizar en conjunto con la base de datos.
2. **Repository:** aquí se desarrollaran las interfaces que extenderán al repositorio JPA el cual permitirá realizar operaciones CRUD y agregar consultas personalizadas.
3. **Rest:** aquí se desarrollaran las clases con las API o endpoint.
4. **Service:** aquí se desarrollaran las clases que implementan toda la lógica del negocio.

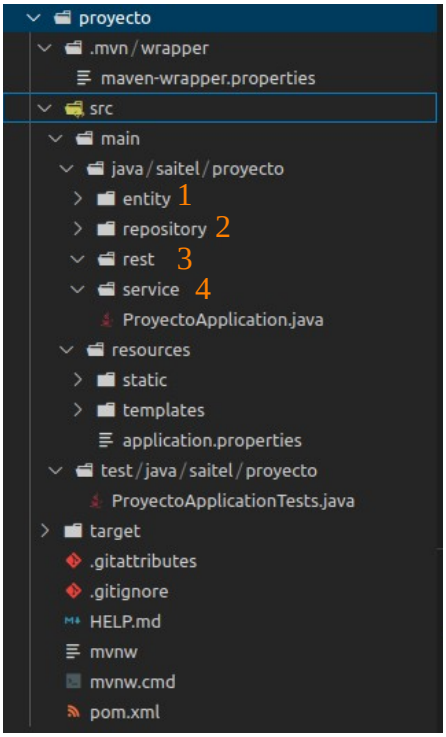


Figura 4.

- **Properties**

  Para iniciar a desarrollar un proyecto, primero se tiene que configurar el puerto, la base a la que se conectara la aplicación, entre otras configuraciones, esto se define en el archivo “application.properties” que se muestra en la Figura 5.

```
# Especifica el tipo de base de datos; aquí se usa el valor por defecto.
spring.jpa.database=default

# Indica si se deben mostrar las consultas SQL en la consola; false oculta el SQL.
spring.jpa.show-sql=false

# Configura cómo Hibernate gestiona el esquema de la base de datos. update mantiene los cambios
sin eliminar datos existentes.
spring.jpa.hibernate.ddl-auto=update

# Especifica el dialecto de SQL de Hibernate, en este caso PostgreSQL
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.PostgreSQLDialect

# Define el controlador (driver) JDBC que debe utilizarse, aquí el de PostgreSQL.
spring.datasource.driver-class-name=org.postgresql.Driver

# URL de conexión a la base de datos, incluyendo protocolo, dirección IP, puerto y base
spring.datasource.url=jdbc:postgresql://127.0.0.1:5432/db_prueba

# Usuario para autenticarse en la base de datos.
spring.datasource.username=postgres

# Contraseña para acceder a la base de datos.
spring.datasource.password=postgres

# Número máximo de conexiones a la base de datos que HikariCP mantiene en el pool.
spring.datasource.hikari.maximum-pool-size=1

# Número mínimo de conexiones inactivas en el pool. 0 permite que todas las conexiones se cierren
cuando no se utilizan.
spring.datasource.hikari.minimum-idle=0
```

```
# Nombre del pool de conexiones, útil para identificarlo en los logs.
spring.datasource.hikari.pool-name=proyecto

# Controla si se incluye el stacktrace  en las respuestas de error; never lo desactiva.
server.error.include-stacktrace=never

# Puerto en el que se ejecuta el servidor; aquí es el 8080.
server.port=8080
```

Figura 5.

- **Base de datos**

Antes de iniciar con el mapeo de base de datos, se creara un ejemplo simple de base que almacenara una tabla llamada tbl_empleado como se muestra en la Figura 6.

```
create sequence sec_empleado;
create table tbl_empleado(
        id_empleado int8 NOT null default nextval('sec_empleado'),
        nombre varchar(100) not null,
        apellido varchar(200) not null,
        sexo boolean
);
```

Figura 6.

- **Entity**

Para inicar con el mapeo de una tabla de base de datos se crea una clase llamada Empleado y se utiliza principalmente dos anotaciones: **@Table** donde se registra el nombre de la tabla a mapear, **@Entity** para hacerle conocer a Spring que es una entidad y **@Column** donde se define los campos a mapear en la tabla como se muestra en la Figura 6.

```
@Entity
@Table(name = "tbl_empleado")

public class Empleado {

        @Id
        @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "secEmpleado")
        @SequenceGenerator(name = "secEmpleado", sequenceName = "sec_empleado",
        allocationSize = 1)
        @Column(name = "id_empleado")
        private Long idEmpleado;

        @Column(name = "nombre")
        private String nombre;

        @Column(name = "apellido")
        private String apellido;

        @Column(name = "sexo")
        private Boolean sexo;

        public Empleado() {
        }


        public Empleado(Long idEmpleado, String nombre, String apellido, Boolean sexo) {
                this.idEmpleado = idEmpleado;
                this.nombre = nombre;
                this.apellido = apellido;
```

```
                this.sexo = sexo;
        }

        public Long getIdEmpleado() {
                return this.idEmpleado;
        }

        public void setIdEmpleado(Long idEmpleado) {
                this.idEmpleado = idEmpleado;
        }

        public String getNombre() {
                return this.nombre;
        }

        public void setNombre(String nombre) {
                this.nombre = nombre;
        }

        public String getApellido() {
                return this.apellido;

        }

        public void setApellido(String apellido) {
                this.apellido = apellido;
        }

        public Boolean isSexo() {
                return this.sexo;
        }

        public Boolean getSexo() {
                return this.sexo;
        }

        public void setSexo(Boolean sexo) {
                this.sexo = sexo;
        }

}
```

Figura 6.

- **Repository**

  En el repository se extendera la interface JpaRepository, el cual recibirá la entidad o tabla (Empleado) y el tipo de dato que se definió como llave primaria o Id (Long), ver Figura 7. JpaRepository nos proveerá las operaciones CRUD (crear, leer, actualizar, eliminar).

  Ademas, en el repository se definen todas las consultas necesarias para trabajar con la base de datos. Para obtener los datos se utiliza el prefijo **findBy** seguido por el parámetro o campo por el cual buscaremos un registro en la base de datos.

  Por ejemplo, si necesitamos obtener los datos de un empleado por su id de empleado, la consulta seria de la siguiente manera findBy**IdEmpleado**. Recuerde que el método debe ser llamado con el mismo nombre que se uso en el mapeo de la Entidad y debe recibir el parámetro por el cual se esta buscando el empleado en este caso, ver Figura 7.

```
public interface EmpleadoRepository extends JpaRepository<Empleado, Long> {
        Optional<Empleado> findByIdEmpleado(Long idEmpleado);
}
```

Figura 7.

Por defecto JpaRepository carga varios métodos, entre ellos el que mas se utilizara para hacer el registro y actualización de datos sera el método **save**.

- **Service**

En el service se desarrolla toda la lógica que tendrá el negocio. Primero se define una interface que guardara todos los métodos a utilizar. Para este caso se definirá un método que almacenara la información de un empleado, ver Figura 8.

```
public interface EmpleadoService {
        Empleado saveEmpleado(Empleado empleado);
}
```

Figura 8.

Una vez definido el método agregamos una clase que implemente su lógica. Aquí se llama al metodo save, para registrar un nuevo empleado, desde el repository con la anotacion **@Autowired**. Esta anotación se usa para *inyectar* automáticamente dependencias en una clase. Es decir, Spring se encarga de crear y proporcionar las instancias necesarias del repository, en este caso, sin tener que instanciarlos manualmente, ver Figura 9. Recuerden agregar la anotación **@Service** al inicio.

```
@Service
public class EmpleadoServiceImpl implements EmpleadoService {

        @Autowired
        private EmpleadoRepository empleadoRepository;

        @Override
        public Empleado saveEmpleado(Empleado empleado) {
                return empleadoRepository.save(empleado);
        }
}
```

Figura 9.

- **Rest**

Aquí se define el endpoint o API a través del cual probaremos la funcionalidad de los servicios. Existen diferentes metodos para definir una API entre ellos los mas usados son los siguientes:

- GET: para consultar o obtener datos.
- PUT: para actualizar datos.
- POST: para ingresar datos.
- DELETE: para eliminar datos.

En este caso, como se esta ingresando un nuevo empleado, se utilizara el método POST al cual se podrá acceder desde una ruta definida como "/ingresar/empleado". Para acceder al servicio nuevamente utilizamos la anotación @Autowired y se llama al método anteriormente llamado **saveEmpleado**, como se muestra en la Figura 10. Recuerden agregar las anotaciones **@RestController** y **@CrossOrigin** al inicio de la clase.

```
@RestController
@CrossOrigin

public class EmpleadoRest {

	@Autowired
	private EmpleadoService empleadoService;

	@PostMapping("/ingresar/empleado")
	public ResponseEntity<Empleado> postEmpleado(@RequestBody Empleado empleado) {
		return new ResponseEntity<>(empleadoService.saveEmpleado(empleado),
		HttpStatus.CREATED);
	}
}
```

Figura 10.

Para los métodos **POST, PUT y DELETE** todos los datos que se requieran enviar a la API deben ser enviados en el body en formato JSON, esto se especifica con la anotación **@RequestBody** seguido del tipo de dato que se requiere, en este caso un Empleado con los datos como se muestra en la Figura 11. Estos datos deben ser enviados desde la aplicación POSTMAN que se explica en el siguiente apartado.

```
{
	"nombre":"Pepito",
	"apellido":"Perez",
	"sexo":true
}
```

Figura 11.

- **POSTMAN**

Una vez definido el endpoint o API, se necesita probar la funcionalidad de la misma. En la Figura 12, se muestra como realizar la solicitud. Se selecciona el método POST a probar y se coloca los datos en body > raw > JSON. En este caso se probara el ingreso de los datos de un empleado y devolverá los datos almacenados.

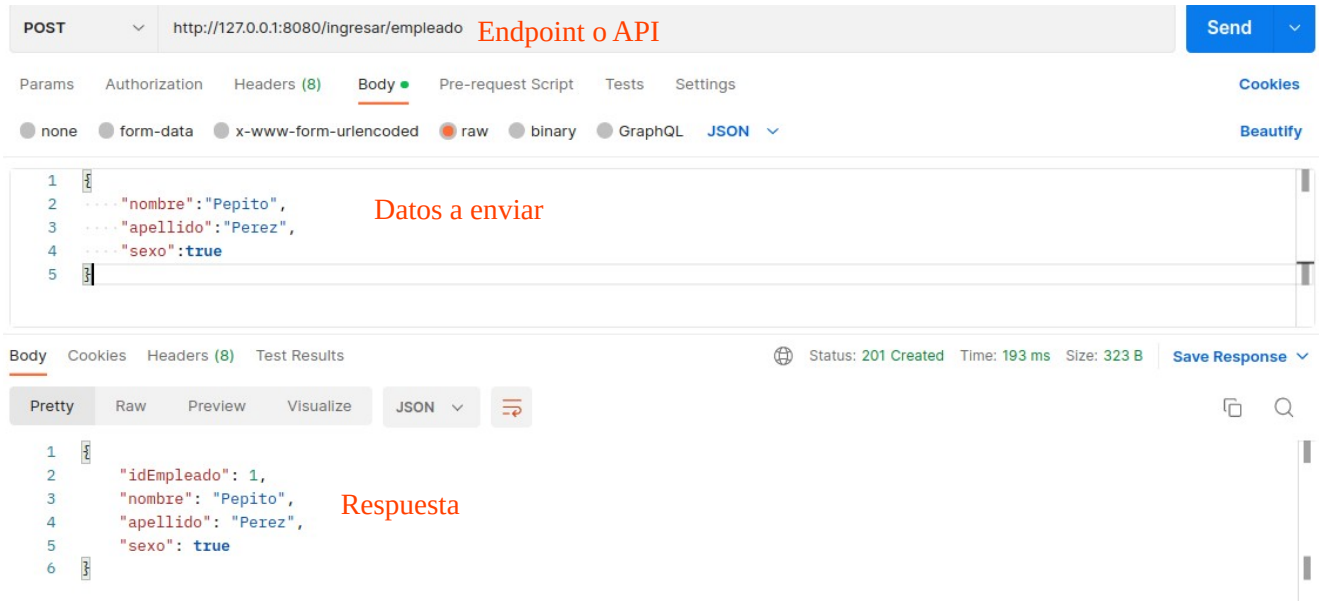


Figura 12.

- **Ejemplo de GET**
- **Rest**

Para obtener la información de un empleado se define la ruta "/obtener/empleado/" y se solicitara el id del empleado, el cual se especifica entre **{}** con la anotación **@PathVariable** como se muestra en la Figura 13. Si se solicita otro parametros de igual manera, se lo incluira en la ruta separandolo con / el parámetro nuevo y la anotacion con el tipo de dato, por ejemplo, **"/obtener/empleado/{idEmpleado}/{parametronuevo}" y (@PathVariable Long idEmpleado, @PathVariable TipoDato parametronuevo)**

```
@GetMapping("/obtener/empleado/{idEmpleado}")
public ResponseEntity<Empleado> getMethodName(@PathVariable Long idEmpleado) {
        return new ResponseEntity<>(empleadoService.getEmpleado(idEmpleado), HttpStatus.OK);
}
```

Figura 13.

- **Service**

Se agrega el método **getEmpleado** que sera llamado desde el rest y recibira el id del empleado, ver Figura 14.

```
public interface EmpleadoService {
        Empleado saveEmpleado(Empleado empleado);
        Empleado getEmpleado(Long idEmpleado);
}
```

Figura 14.

- **ServiceImpl**

Se implementa la lógica del método anteriormente agregado. Se busca al empleado en la base de datos con el metodo **findByIdEmpleado** que recibe el id del empleado. Si encuentra al empleado devolvemos sus datos, si no devolvemos un mensaje de error con estado no encontrado, como en la Figura 15.

```
@Service
public class EmpleadoServiceImpl implements EmpleadoService {

@Autowired
private EmpleadoRepository empleadoRepository;

@Override
public Empleado saveEmpleado(Empleado empleado) {
return empleadoRepository.save(empleado);
}

@Override
	public Empleado getEmpleado(Long idEmpleado) {
		Optional<Empleado> empleado = empleadoRepository.findByIdEmpleado(idEmpleado);
		if(empleado.isPresent()){
			return empleado.get();
		}else{
			throw new ResponseStatusException(HttpStatus.NOT_FOUND, "Empleado con
			id " + idEmpleado + " no encontrado");
		}
	}
}
```

Figura 15.

- **Repository**

Finalmente se agrega la consulta usando al **id de empleado** como parámetro de búsqueda en la base de datos, ver Figura 16. Se agrega el Optional , para utilizar métodos como isPresent(), que nos ayuda a verificar si existe o no datos devueltos por la consulta. En caso de no usar Optional, cuando se defina la lógica del negocio se verifica que Empleado sea diferente del valor nulo, por ejemplo if(empleado != null){...}.

```
public interface EmpleadoRepository extends JpaRepository<Empleado, Long> {
	Optional<Empleado> findByIdEmpleado(Long idEmpleado);
}
```

Figura 16.

- **Despliegue a un servidor con kubernetes**

- **Instalar Docker**

La instalacion de Docker puede variar dependiendo del sistema operativo que se use. Para este caso se detalla los comandos necesarias para instalarlo en Ubuntu, ver Figura 17.

```
sudo apt update
sudo apt install apt-transport-https ca-certificates curl software-properties-common
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu focal stable"
sudo apt update
apt-cache policy docker-ce
sudo apt install docker-ce
sudo systemctl status docker
sudo usermod -aG docker ${USER}
su - ${USER}
sudo usermod -aG docker username
```

Figura 17.

Un tutorial detallado de como instalar y usar docker lo pueden encontrar en el siguiente enlace: https://www.digitalocean.com/community/tutorials/how-to-install-and-use-docker-on-ubuntu-20-04-es

En caso de tener otro sistema, con Windows por ejemplo, hay varios tutoriales detallados en internet, como el siguiente https://www.ionos.com/es-us/digitalguide/servidores/configuracion/instalar-docker-en-windows-11/

- **Colocar una aplicación en Docker**

Para colocar el proyecto en Docker se crea en la raíz del proyecto un archivo que se llamara Dockerfile con una plantilla como se muestra en la Figura 18.

```
# Etapa 1: Construcción de la aplicación
# Usa una imagen base de OpenJDK 18 para la compilación
FROM openjdk:18 as build
# Establece el directorio de trabajo
WORKDIR /workspace/app
# Copia el script de Maven Wrapper al contenedor
COPY mvnw .
# Copia la carpeta de configuración de Maven Wrapper
COPY .mvn .mvn
# Copia el archivo pom.xml (definición del proyecto Maven)
COPY pom.xml .
# Copia el código fuente del proyecto
COPY src src
# Compila el proyecto Maven sin ejecutar las pruebas
RUN ./mvnw install -DskipTests
# Crea un directorio de dependencias y extrae el contenido del JAR compilado
RUN mkdir -p target/dependency && (cd target/dependency; jar -xf ../*.jar)

# Etapa 2: Creación de la imagen final
# Usa una imagen base de OpenJDK 18 para la ejecución
FROM openjdk:18
# Define un volumen temporal para almacenar datos temporales
VOLUME /tmp
# Establece la zona horaria en "América/Guayaquil"
```

```
ENV TZ=America/Guayaquil
# Define la ubicación de las dependencias
ARG DEPENDENCY=/workspace/app/target/dependency
# Copia las bibliotecas del JAR extraído
COPY --from=build ${DEPENDENCY}/BOOT-INF/lib /app/lib
# Copia la metadata del JAR
COPY --from=build ${DEPENDENCY}/META-INF /app/META-INF
# Copia las clases del JAR
COPY --from=build ${DEPENDENCY}/BOOT-INF/classes /app
# Expone el puerto 8080 para la aplicación
EXPOSE 8080

ENTRYPOINT ["java","-cp","app:app/lib/*", "-Xmx1024M", "-Xms512M",
"saitel.proyecto.ProyectoApplication"]
# Define el comando para iniciar la aplicación:
# - `-cp app:app/lib/*` establece el classpath.
# - `-Xmx1024M -Xms512M` ajusta la memoria máxima y mínima de la JVM.
# - `saitel.proyecto.ProyectoApplication` es la clase principal.
```

Figura 17.

Cada vez que se cree un proyecto diferente, en el código solo cambiara la versión de java que se utiliza, el puerto expuesto con el cual trabaja la aplicación, los ajustes de memoria mínima (-Xms) y máxima(-Xmx) requerida por la aplicación en MB y la ruta donde se ubica la claseprincipal. En la Figura 18 se muestra donde esta ubicada la clase principal y donde se debería colocar el archivo Dockerfile para este ejemplo.

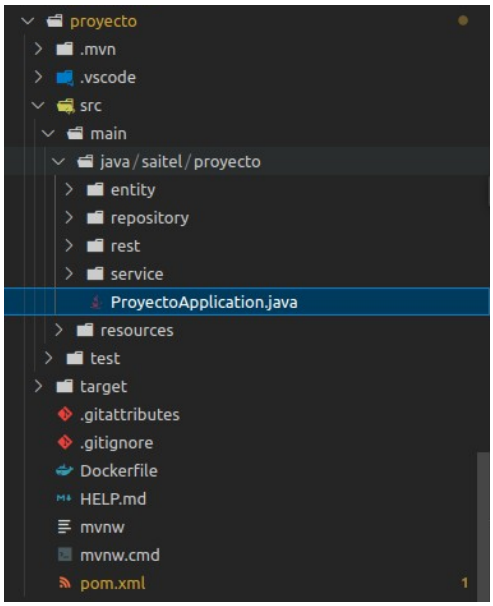


Figura 18.

Para construir la imagen se ubica con la consola en la raíz del proyecto y se ejecuta el comando **docker build -t prueba:1.0 .** como se muestra en la Figura 19, donde **prueba** es el nombre de la imagen y seguido de dos puntos **:** se coloca la versión de la imagen, en este caso sera la 1.0.

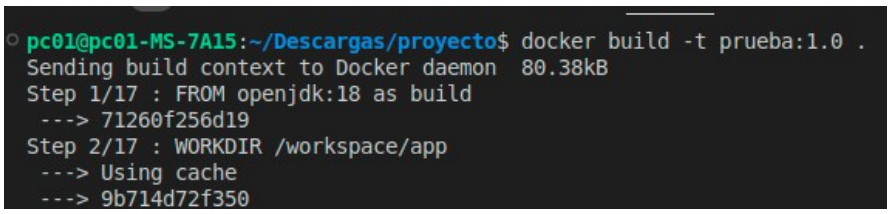
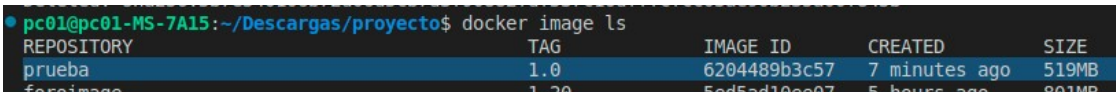


Figura 19.

Una vez construida la imagen podemos buscar las imágenes que tengamos con el comando **docker image ls.**



También se puede verificar que la imagen se haya construido correctamente al ejecutarla con el comando **docker run prueba:1.0**, y no muestra ningún mensaje de error como se muestra en la Figura 20.

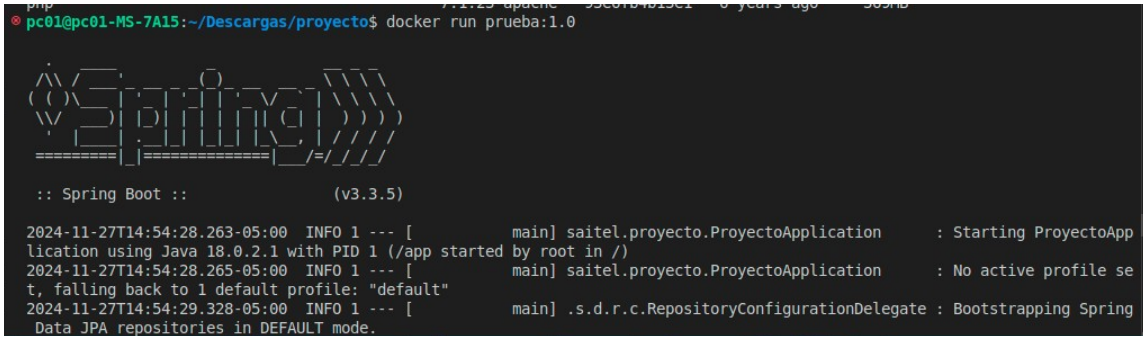


Figura 20.

- **Instalar kubernetes**

Para instalar un clúster de kubernetes podemos utilizar microk8s desde snap. microk8s es una versión ligera que se auto configura por si sola. En la Figura 21 se tienen los comandos necesarios para instalarlo.

```
#Instalar microk8s
sudo snap install microk8s –classic
#Agregar el usuario actual al grupo y obtener acceso al directorio de almacenamiento en caché .kube
sudo usermod -a -G microk8s $USER
mkdir -p ~/.kube
chmod 0700 ~/.kube
# Reingresar para aplicar los cambios
su - $USER
#Verificar el estado
microk8s status
#Habilitar DNS para facilitar la comunicación entre servicios
microk8s enable dns
```

Figura 21.

Un ejemplo de microk8s instalado exitosamente se puede observar en la Figura 22, donde el estado de kubernetes esta ejecutando “running”.

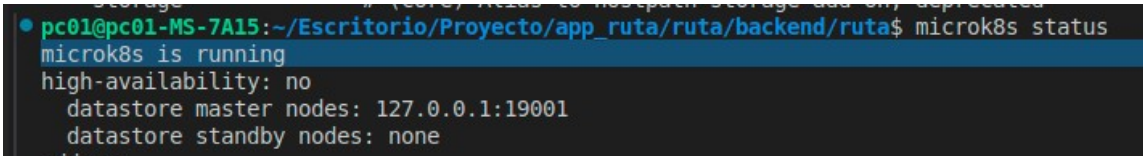


Figura 22.

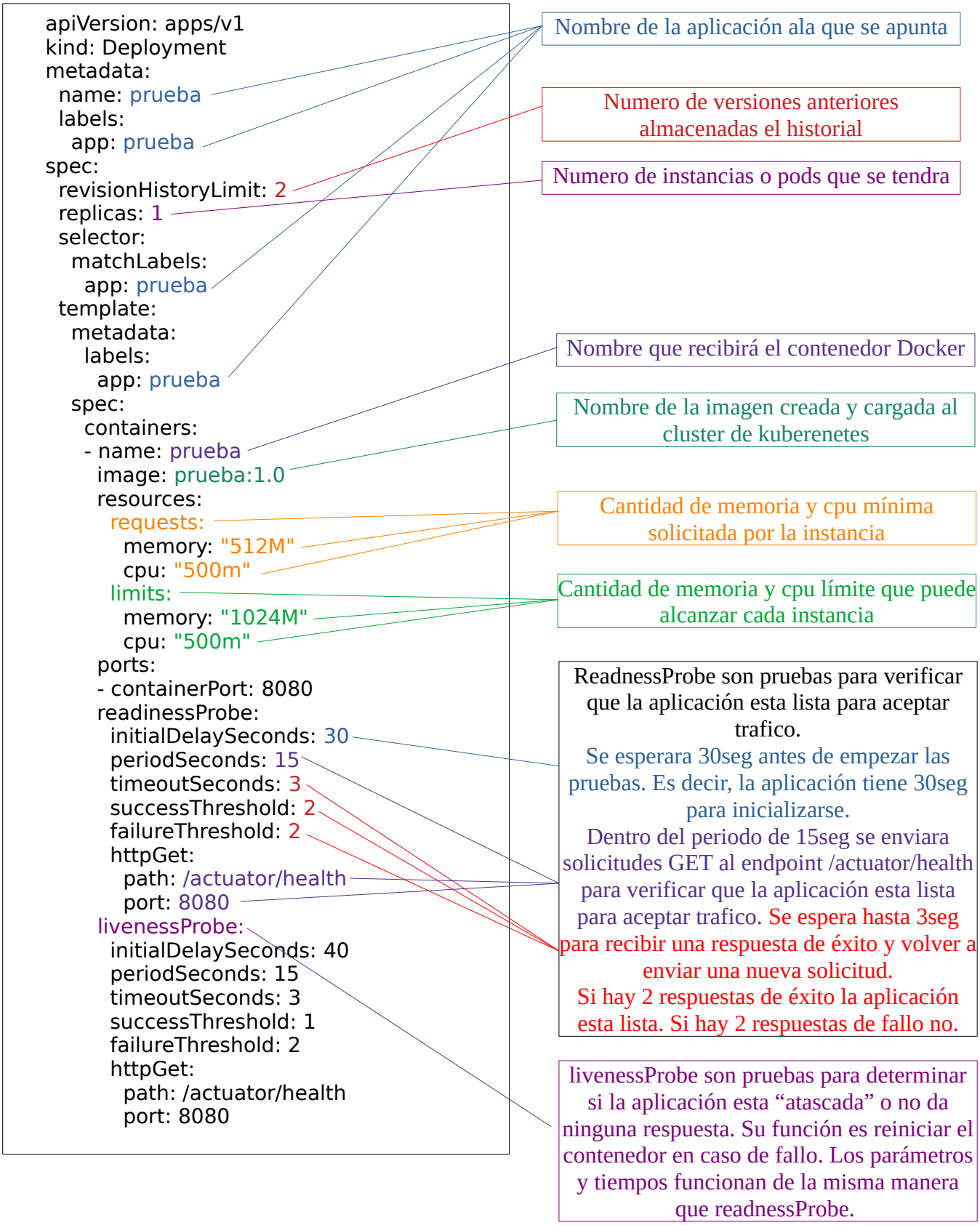
En caso de presentar problemas con microk8s también existe otra opción como lo es k3s. K3s funciona de la misma manera que microk8s y para instalarlo se debe ejecutar la siguiente linea de comando: **curl -sfL https://get.k3s.io | sh -**

- **Desplegar una aplicación en kubernetes**

  Para desplegar una aplicación en kubernetes existen 3 archivos .yml principales que se deben configurar:

  - **Deployment**: este archivo tendrá la información acerca de la capacidad que tendrá cada instancia (o pod) de Docker que se cree en kubernetes.
  - **Service:** este archivo tendrá la información acerca de como se comunicaran el servicio con el cliente u otros servicios.
  - **Autoscaler:** este archivo tendrá información acerca de los limites a los que debe llegar cada instancia (o pod) antes de abrir una nueva en caso de que exista una alta demanda de un servicio.

- Deployment


apiVersion: apps/v1
kind: Deployment
metadata:
name: prueba
labels:
app: prueba
spec:
revisionHistoryLimit: 2
replicas: 1
selector:
matchLabels:
app: prueba
template:
metadata:
labels:
app: prueba
spec:
containers:
- name: prueba
image: prueba:1.0
resources:
requests:
memory: "512M"
cpu: "500m"
limits:
memory: "1024M"
cpu: "500m"
ports:
- containerPort: 8080
readinessProbe:
initialDelaySeconds: 30
periodSeconds: 15
timeoutSeconds: 3
successThreshold: 2
failureThreshold: 2
httpGet:
path: /actuator/health
port: 8080
livenessProbe:
initialDelaySeconds: 40
periodSeconds: 15
timeoutSeconds: 3
successThreshold: 1
failureThreshold: 2
httpGet:
path: /actuator/health
port: 8080
Nombre de la aplicación ala que se apunta
Numero de versiones anteriores almacenadas el historial
Numero de instancias o pods que se tendra
Nombre que recibirá el contenedor Docker
Nombre de la imagen creada y cargada al cluster de kuberenetes
Cantidad de memoria y cpu mínima solicitada por la instancia
Cantidad de memoria y cpu límite que puede alcanzar cada instancia
ReadnessProbe son pruebas para verificar que la aplicación esta lista para aceptar trafico.
Se esperara 30seg antes de empezar las pruebas. Es decir, la aplicación tiene 30seg para inicializarse.
Dentro del periodo de 15seg se enviara solicitudes GET al endpoint /actuator/health para verificar que la aplicación esta lista para aceptar trafico. Se espera hasta 3seg para recibir una respuesta de éxito y volver a enviar una nueva solicitud.
Si hay 2 respuestas de éxito la aplicación esta lista. Si hay 2 respuestas de fallo no.
livenessProbe son pruebas para determinar si la aplicación esta “atascada” o no da ninguna respuesta. Su función es reiniciar el contenedor en caso de fallo. Los parámetros y tiempos funcionan de la misma manera que readnessProbe.