Tutorial Spring Boot

- **Herramientas necesarias**

  Java 8 o superior
  VSCode

- **Instalación**

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

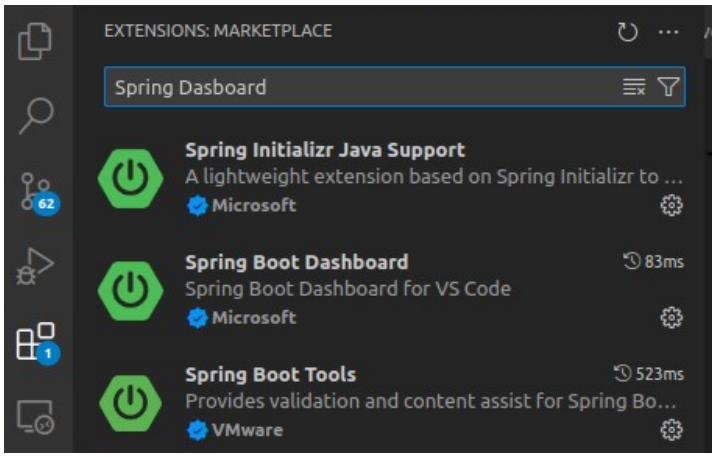

Figura 1.

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.

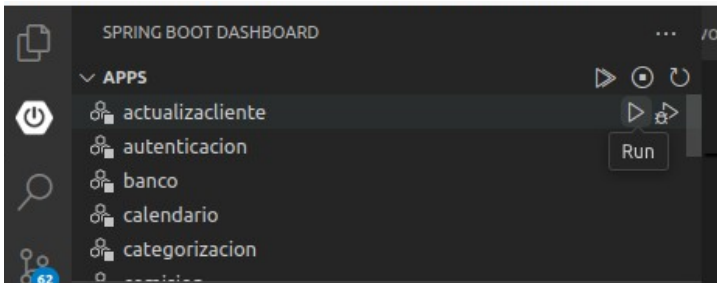

Figura 2.

- **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.

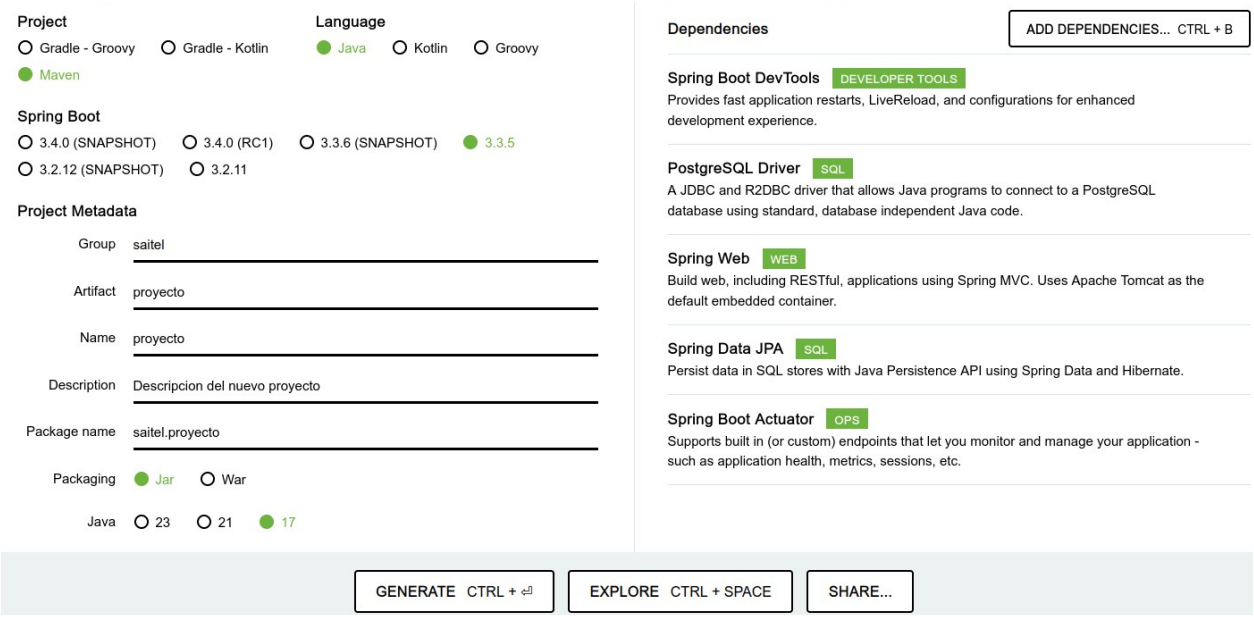

Figura 3

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.

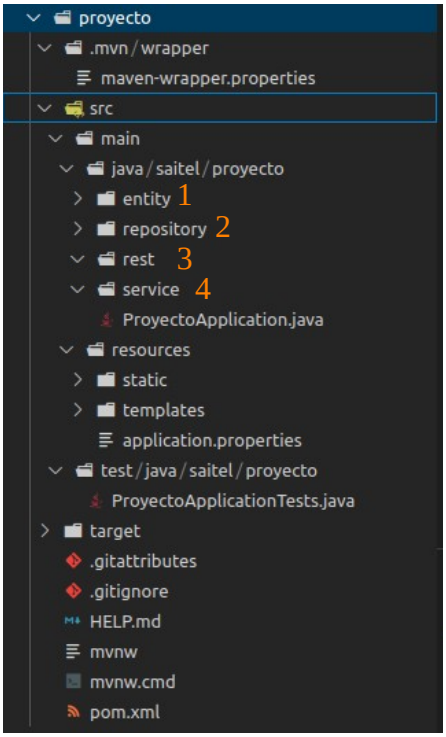

Figura 4

- **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.propesties” 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_isp

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

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

# 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

- **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
	@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 metodo 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.


POST
http://127.0.0.1:8080/ingresar/empleado
Endpoint o API
Send
Params
Authorization
Headers (8)
Body
Pre-request Script
Tests
Settings
Cookies
none
form-data
x-www-form-urlencoded
raw
binary
GraphQL
JSON
Beautify
{
"nombre":"Pepito",
"apellido":"Perez",
"sexo":true
}
Datos a enviar
Body
Cookies
Headers (8)
Test Results
Status: 201 Created
Time: 193 ms
Size: 323 B
Save Response
Pretty
Raw
Preview
Visualize
JSON
{
"idEmpleado": 1,
"nombre": "Pepito",
"apellido": "Perez",
"sexo": true
}
Respuesta

Figura 12