El uso de Spring Boot Load Data o carga de datos es bastante común en las aplicaciones modernas. Muchas de ellas usan JPA o JDBC y SQL para almacenar la información en una base de datos relacional y necesitan una primera carga de datos para pruebas unitarias etc. Vamos a ver cómo Spring Boot de una forma muy sencilla se encarga de cargar los datos a través del uso de anotaciones y ficheros.

Spring Boot y JPA

Lo primero que vamos a hacer es construir una entidad de JPA que se pueda persistir en la base de datos . Para ello usaremos spring initializer y nos descargaremos un proyecto de Spring Boot para JPA.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.4.2</version>
    <relativePath /> <!-- lookup parent from repository -->
  </parent>
  <groupId>com.arquitecturajava</groupId>
  <artifactId>data1</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>data1</name>
  <description>Demo project for Spring Boot</description>
  <properties>
    <java.version>11</java.version>
```

```
</properties>
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>

  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <scope>runtime</scope>
    <optional>true</optional>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>

  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
  </dependency>
</dependencies>

<build>
```

```
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
      </plugin>
    </plugins>
  </build>

</project>
```

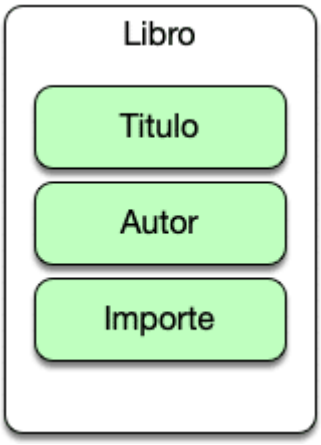
Una vez tenemos configurado el proyecto a nivel de dependencias , el siguiente paso es generarnos la configuración a nivel de spring boot relativa a JPA.

```
package com.arquitecturajava.data1;

import java.util.Properties;

import javax.sql.DataSource;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import
org.springframework.data.jpa.repository.config.EnableJpaRepositories;
import org.springframework.jdbc.datasource.DriverManagerDataSource;
import org.springframework.orm.jpa.JpaTransactionManager;
import
org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean;
import org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter;
import org.springframework.transaction.PlatformTransactionManager;
import
org.springframework.transaction.annotation.EnableTransactionManagement
```

```
;

@SpringBootApplication

@EnableJpaRepositories("com.arquitecturajava.data1")
@EnableTransactionManagement
public class Data1Application {

  public static void main(String[] args) {
    SpringApplication.run(Data1Application.class, args);
  }

    @Bean
    public DataSource dataSource() {
        DriverManagerDataSource dataSource = new
DriverManagerDataSource();
        dataSource.setDriverClassName("com.mysql.jdbc.Driver");
        dataSource.setUrl("jdbc:mysql://localhost:8889/biblioteca");
        dataSource.setUsername("root");
        dataSource.setPassword("root");
        return dataSource;
    }

    @Bean
    LocalContainerEntityManagerFactoryBean
entityManagerFactory(DataSource dataSource) {

        LocalContainerEntityManagerFactoryBean
entityManagerFactoryBean = new
LocalContainerEntityManagerFactoryBean();
        entityManagerFactoryBean.setDataSource(dataSource);
```

```
        entityManagerFactoryBean.setJpaVendorAdapter(new
HibernateJpaVendorAdapter());
entityManagerFactoryBean.setPackagesToScan("com.arquitecturajava.data1
");

        Properties jpaProperties = new Properties();

        jpaProperties.put("hibernate.dialect",
"org.hibernate.dialect.MySQLDialect");

        entityManagerFactoryBean.setJpaProperties(jpaProperties);

        return entityManagerFactoryBean;
    }
    @Bean public PlatformTransactionManager transactionManager() {
        JpaTransactionManager txManager= new JpaTransactionManager();
txManager.setEntityManagerFactory(entityManagerFactory(dataSource()).g
etObject());
        return txManager;
    }

}
```

Una vez tenemos ambas partes deberemos abordar otro paso importante que es el de definir nuestro modelo de JPA.



En este caso se trata del concepto de Libro que es una entidad sencilla y que es la que vamos a persistir en la base de datos.

```
package com.arquitecturajava.data1;



import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name="Libros")
public class Libro {

  @Id
  private String isbn;
  private String titulo;
  private String autor;
  public String getIsbn() {
    return isbn;
```

```
}
public void setIsbn(String isbn) {
  this.isbn = isbn;
}
public String getTitulo() {
  return titulo;
}
public void setTitulo(String titulo) {
  this.titulo = titulo;
}
public String getAutor() {
  return autor;
}
public void setAutor(String autor) {
  this.autor = autor;
}
public Libro() {
  super();
}

public Libro(String isbn) {
  super();
  this.isbn = isbn;
}
@Override
public int hashCode() {
  final int prime = 31;
  int result = 1;
  result = prime * result + ((isbn == null) ? 0 : isbn.hashCode());
  return result;
}
```

```
  @Override
  public boolean equals(Object obj) {
    if (this == obj)
      return true;
    if (obj == null)
      return false;
    if (getClass() != obj.getClass())
      return false;
    Libro other = (Libro) obj;
    if (isbn == null) {
      if (other.isbn != null)
        return false;
    } else if (!isbn.equals(other.isbn))
      return false;
    return true;
  }
  public Libro(String isbn, String titulo, String autor) {
    super();
    this.isbn = isbn;
    this.titulo = titulo;
    this.autor = autor;

  }
}
```

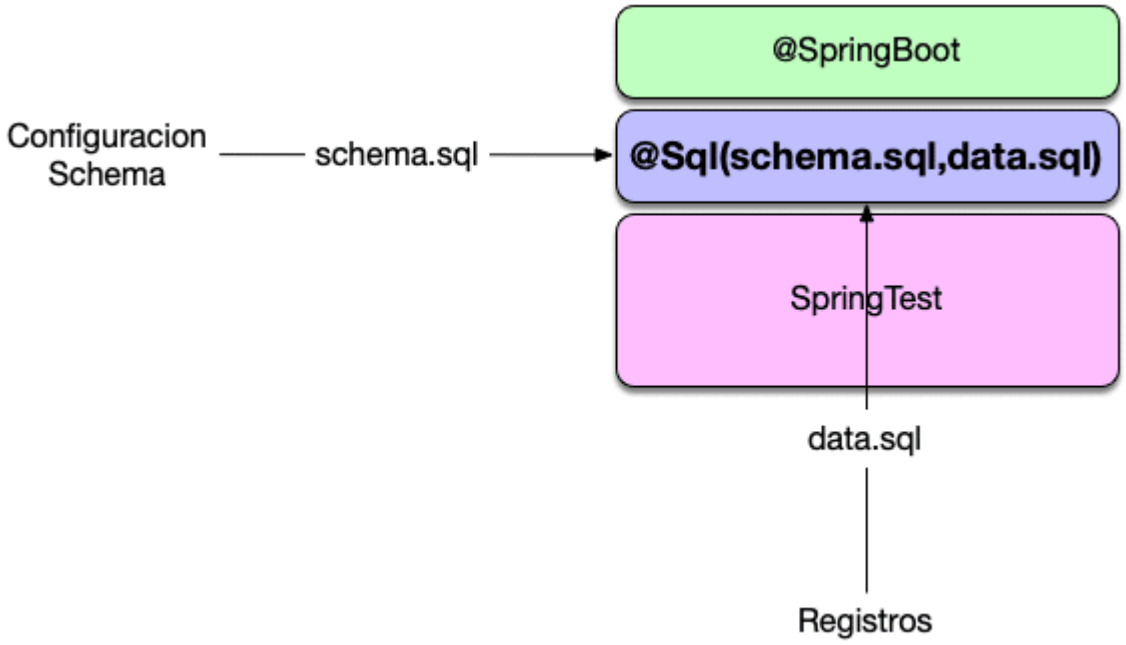
Spring Boot Load Data y Testing

Es momento de configurar Spring Data y las pruebas unitarias para que sean capaces de cargar información de forma automática en la base de datos .

```
package com.arquitecturajava.data1;

import static org.hamcrest.MatcherAssert.assertThat;
import static org.hamcrest.Matchers.hasItem;
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertNotNull;

import java.util.List;
import java.util.Optional;

import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.jdbc.Sql;

@SpringBootTest
@Sql({ "/schema.sql", "/data.sql" })
class Data1ApplicationTests {

  @Autowired
  LibroRepository repositorio;

  @Test
  void elRepositorioExiste() {
    assertNotNull(repositorio);

  }

  @Test
  void buscarUno() {
    Optional<Libro> oLibro = repositorio.findById("1");
```

```
    if (oLibro.isPresent()) {

      assertEquals("1", oLibro.get().getIsbn());
    }
  }

  @Test
  void buscarTodos() {
    List<Libro> libros = repositorio.findAll();
    Libro l1 = new Libro("1", "Java", "Cecilio");
    Libro l2 = new Libro("2", "Net", "Pedro");
    assertThat(libros, hasItem(l1));
    assertThat(libros, hasItem(l2));

  }


}
```

Cómo podemos ver a nivel de pruebas unitarias disponemos de la anotación @Sql que nos permite definir como podemos cargar de forma automática información en la base de datos .



En este caso Spring se apoya en dos ficheros schema.sql y data.sql que como su nombre indica definen el schema de base de datos y los datos a insertar en las diferentes tablas.

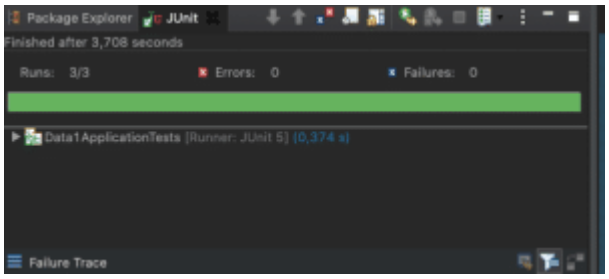
```
DROP TABLE IF EXISTS Libros;
create table Libros (isbn varchar(10), titulo varchar(25),autor
varchar(25));
```

```
insert into Libros values ("1","Java","Cecilio")
insert into Libros values ("2","Net","Pedro")
insert into Libros values ("3","PHP","Cecilio")
```

Spring Test y Resultados

De esta manera Spring Boot queda configurado para de forma automática cargar datos en la base de datos y facilitar la gestión de pruebas unitarias en nuestro proyecto. Simplemente

ejecutamos las pruebas y obtendremos un resultado correcto ya que los scripts de carga se ejecutan.



Otros artículos relacionados

1. Curso de Spring Boot Gratis en Español
2. Spring Boot Starter ,un concepto fundamental
3. Spring Boot JPA y su configuración