Formas de acceder a una base de datos.

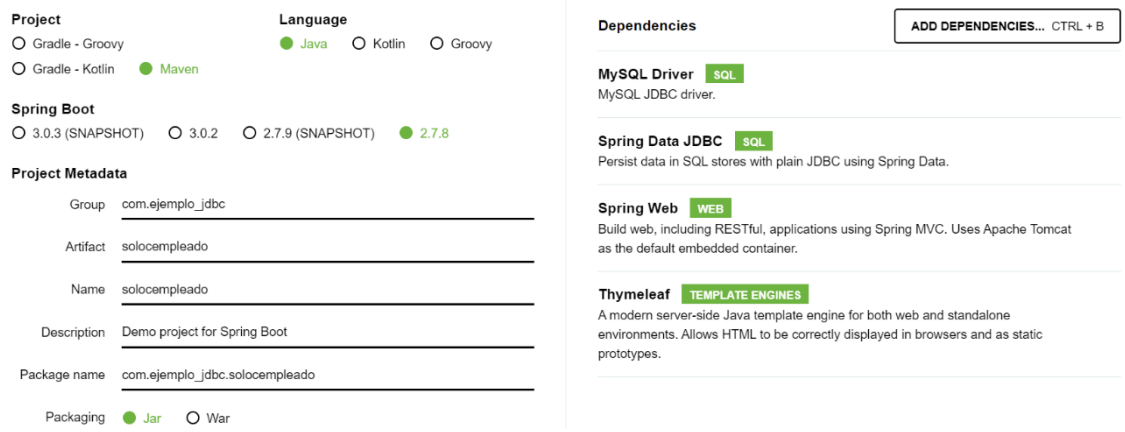
Completa con información clara y concisa, los siguientes mecanismos para acceder a los datos de una base de datos:

1. JDBC
2. JPA API
3. Hibernate
4. Spring Data

Ejemplo guiado – Conexión JDBC con Spring Data

Base de datos en MySQL: soloempleados

Usaremos la herramienta initializr de Spring para indicar las dependencias Maven para nuestro proyecto: https://start.spring.io/



Fichero application.properties

En este fichero tenemos que especificar los parámetros para la conexión JDBC con la base de datos.

En Netbeans, una vez conectado el servicio, en las propiedades de la conexión podemos comprobar los valores para los parámetros que necesita el fichero de propiedades:

```
# Configurar la coneccion a la base de datos
spring.datasource.url = jdbc:mysql://localhost:3306/bd
spring.datasource.username = usuario
spring.datasource.password = contrasenia
spring.datasource.driverClassName = com.mysql.jdbc.Driver
```

Si ejecutamos el proyecto, podemos comprobar que se conecta a la BD. Además, como hemos incluido Spring Web, iniciará el servidor de aplicaciones web Tomcat en el puerto 8080.

Aplicación Web: Muestre la lista de empleados de la tabla Empleado

Las consultas a la tabla las hacemos aprovechando el potencial de Spring Data.

Clase Empleado:

Para poder usar Spring Data, vamos a representar cada tabla con una clase (usando notación de entidades), de forma que con los repositorios de Spring podamos hacer las consultas.

Aunque podemos recurrir al asistente, vamos a escribir la clase con la notación de forma manual:

```java
import org.springframework.data.annotation.Id;
import org.springframework.data.relational.core.mapping.Column;
import org.springframework.data.relational.core.mapping.Table;

@Table("Empleado")
public class Empleado {
    //propiedades para representar las columnas de la tabla
    //spring data asocia estos nombres a las columnas de la tabla
    //Opcionalmente puedo usar la anotación @columna.
    //Resulta util renombrar con @Column si columna y propiedad tienen diferente nombre
    //Voy a usar notación @Id para indicar columna que es pk
    @Id
    private int pkempleado;
    @Column("nombre")
    private String nombre;
    @Column("salario")
    private double salario;
    //generamos los getter y setter

    public int getPkempleado() {return pkempleado;}

    public void setPkempleado(int pkempleado) {this.pkempleado = pkempleado;}

    public String getNombre() {return nombre; }

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

    public double getSalario() {return salario;}

    public void setSalario(double salario) {this.salario = salario;}
```

Interfaz IEmpleado DAO (Interfaz para métodos de acceso a datos Data Object Acces):

Una vez definida la entidad que representa a la tabla, vamos a especificar en una interfaz los métodos necesarios para ejecutar las consultas a la base de datos o las operaciones CRUD.

CrudRepository: Spring Data implementa las operaciones CRUD de manera automática, lo que nos permite extender dicho repositorio en nuestra interfaz para aprovechar el código que ya está desarrollado.

CrudRepository en Spring Data:

https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html

Observar los métodos que implementa y que podemos usar por herencia en nuestra interfaz.

public interface **CrudRepository**<T, ID> extends Repository<T,ID>

Los paréntesis de ángulo se denominan extensiones o “genérics” y hacen referencia al tipo de objeto y al tipo de dato de la clave primaria.

CrudRepository<T,ID> - Notación “generics” o genérica en Spring:

- T es el tipo de dato u objeto sobre el queremos trabajar.

- ID es el tipo de dato del parámetro que actúa como PK (clave primaria)

En nuestro ejemplo serán:

- T = objeto de tipo Empleado
- ID= tipo de dato Integer

Spring utiliza lo que denomina "contexto" que sirve para anotar las extensiones que hacemos de las interfaces y métodos genéricos. Para implementarlo, se usan las **anotaciones estándar de Spring** para los objetos de nuestra aplicación. Por ejemplo:

@Service indica que la clase es un bean de la capa de negocio

@Repository indica que es un DAO.

@Component cuando queremos especificar que algo es un bean sin decir de qué tipo es podemos usar la anotación

@EventListener para crear un método que escucha Spring en el arranque.

@Autowired para inyectar unas dependencias dentro de otras en Spring

Luego, para registrar en Spring nuestra implementación DAO con la interfaz para Empleado, vamos a añadir la anotación @Repository, que será francamente útil para registrar y reconocer las excepciones que se pueden producir cuando accedamos a los datos de la BD.

```
import com.example.springdatajdbc.entidades.Empleado;
import org.springframework.data.repository.CrudRepository;
import org.springframework.stereotype.Repository;

/**
 *
 * @author marina
 */
@Repository
public interface IEmpleadoDAO extends CrudRepository<Empleado, Integer>{
}
```

Ejecutar el código- Probar que funciona.

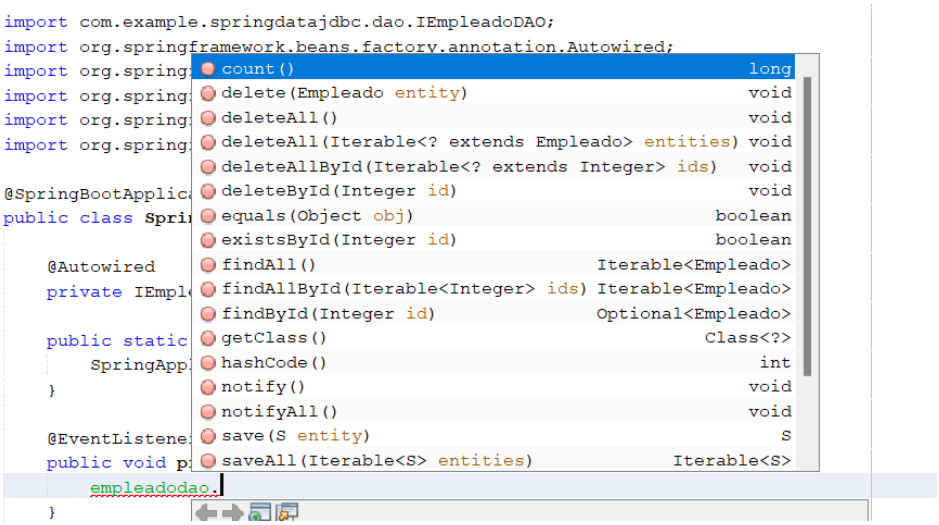
En la clase con el Main, vamos a recurrir a Spring para indicar que ejecute el método que decidamos cuando arranque la aplicación, usando la anotación:

@EventListener({ApplicationReadyEvent.class})

En el Main vamos a declarar una variable para usar la interfaz DAO que hemos definido e incluido en el repositorio de Spring. Para ello, tenemos que decir que **inyecte** las dependencias usando la notación:

@Autowired
private IEmpleadoDAO empleadodao;

Ahora ya podemos usar todos los métodos del CrudRepository que hemos extendido en nuestra interfaz IEmpleadoDAO:



Nuestro entorno de pruebas:

```
import com.example.springdatajdbc.dao.IEmpleadoDAO;
import com.example.springdatajdbc.entidades.Empleado;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.context.event.ApplicationReadyEvent;
import org.springframework.context.event.EventListener;

@SpringBootApplication
public class SpringdataJdbcApplication {

    @Autowired
    private IEmpleadoDAO empleadodao;

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

    @EventListener({ApplicationReadyEvent.class})
    public void pruebaConsultas() {
        try {
            List<Empleado> lista = (List<Empleado>) empleadodao.findAll();
            for (Empleado e : lista) {
                System.out.println(e.getPkempleado() + " - "
                        + e.getNombre() + " - "
                        + e.getSalario());
            }
        } catch (IllegalArgumentException ex) {
            System.out.println("ERROR. " + ex.getMessage());
        }
        System.exit(0);
    }
}
```

Consultas derivadas:

Además de usar las consultas incluidas en CRUDRepository, podemos crear otras particularizadas a nuestras necesidades, haciendo uso de la sintaxis que utiliza Spring. Es decir, podemos construir consultas que derivan de las existentes en el repositorio.

¿Cómo lo hacemos?

Definimos el método en la interfaz IEmpleadoDAO usando la sintaxis adecuada.

Por ejemplo, vamos a localizar un usuario por nombre y salario. Si nombramos al método comenzando con findBy y concatenamos los parámetros o columnas para nuestra búsqueda con And, se hace la magia. Spring interpreta que queremos realiza una búsqueda por los dos campos.

```
@Repository
public interface IEmpleadoDAO extends CrudRepository<Empleado, Integer>{

public Empleado findByNombreAndSalario (String nombre, double salario);

}
```

En el método para ejecutar las consultas podemos probar el resultado.

Consultas específicas:

Las consultas específicas podemos usarlas, definiéndolas en la interfaz DAO.

Por ejemplo, vamos a obtener los nombres de los usuarios con un salario determinado.

public List<Empleado> empleadosConSalario (BigDecimal vsalario);

Si lo hiciésemos en SQL, podríamos plantearlo como una consulta preparada, donde sustituiríamos el valor del salario por la variable:

SELECT * FROM empleado WHERE salario =:vsalario;

En Spring para hacer esto, tenemos que hacer uso de las anotaciones. En este caso, indicaremos que vamos a hacer una QUERY y también que vamos a sustituir en la ella el valor pasado como parámetro: