



Spring Boot Spring Data

Objetivo

Desarrollar componentes persistentes utilizando tecnologías open source Java.

Agenda

I. Que es Spring Boot

II. JPA

III. Spring Data

I) Spring Boot

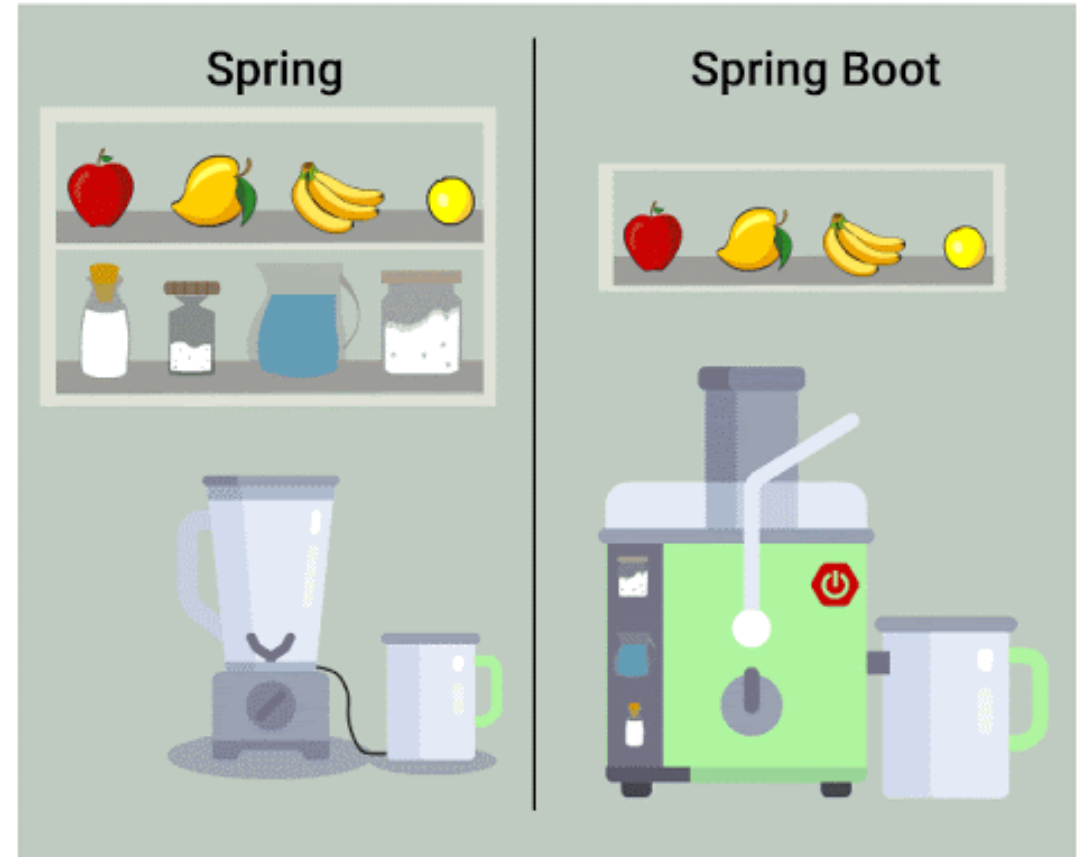
@UPC

I) Spring Boot

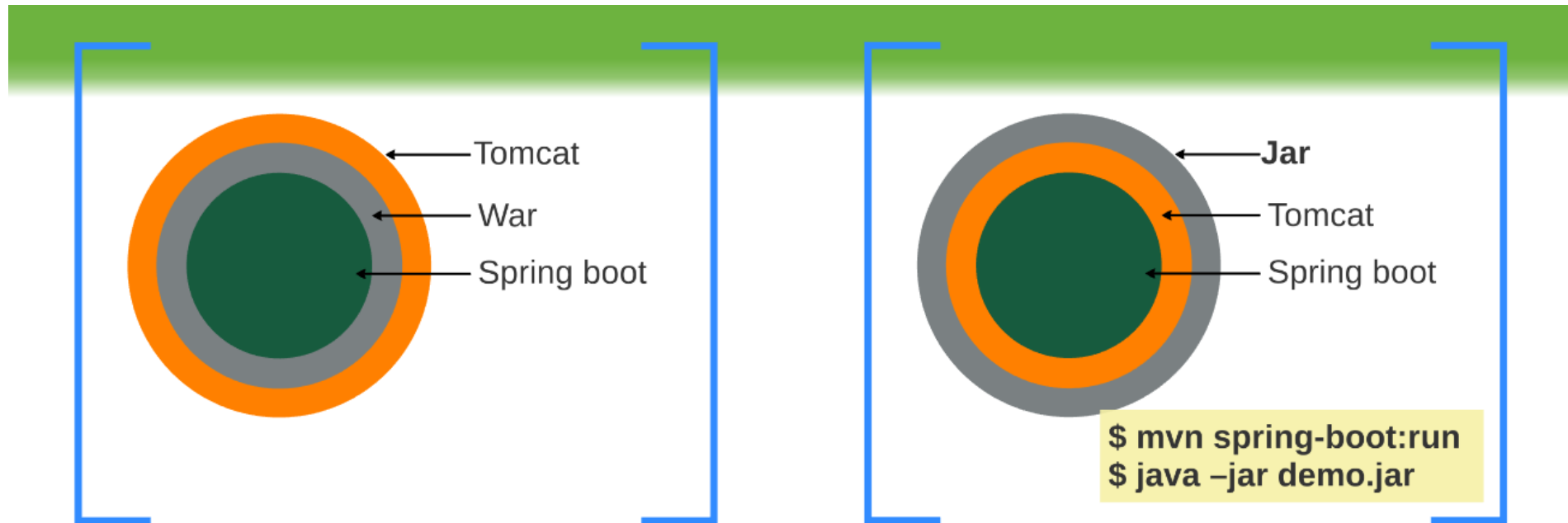
- ☐ Spring es un framework para el desarrollo de aplicaciones web y microservicios.
- ☐ Contenedor de inversión de control, de código abierto para la plataforma Java.
- ☐ Es un-Framework basado en XML ocultado por las anotaciones.
- ☐ Altamente configurable
- ☐ Trae las dependencias necesarias para un tipo de aplicación
- ☐ Soporta todas la librerías de terceros, No SQL DB, Distributed Cache, JPA, REST, etc.

Spring vs Spring Boot

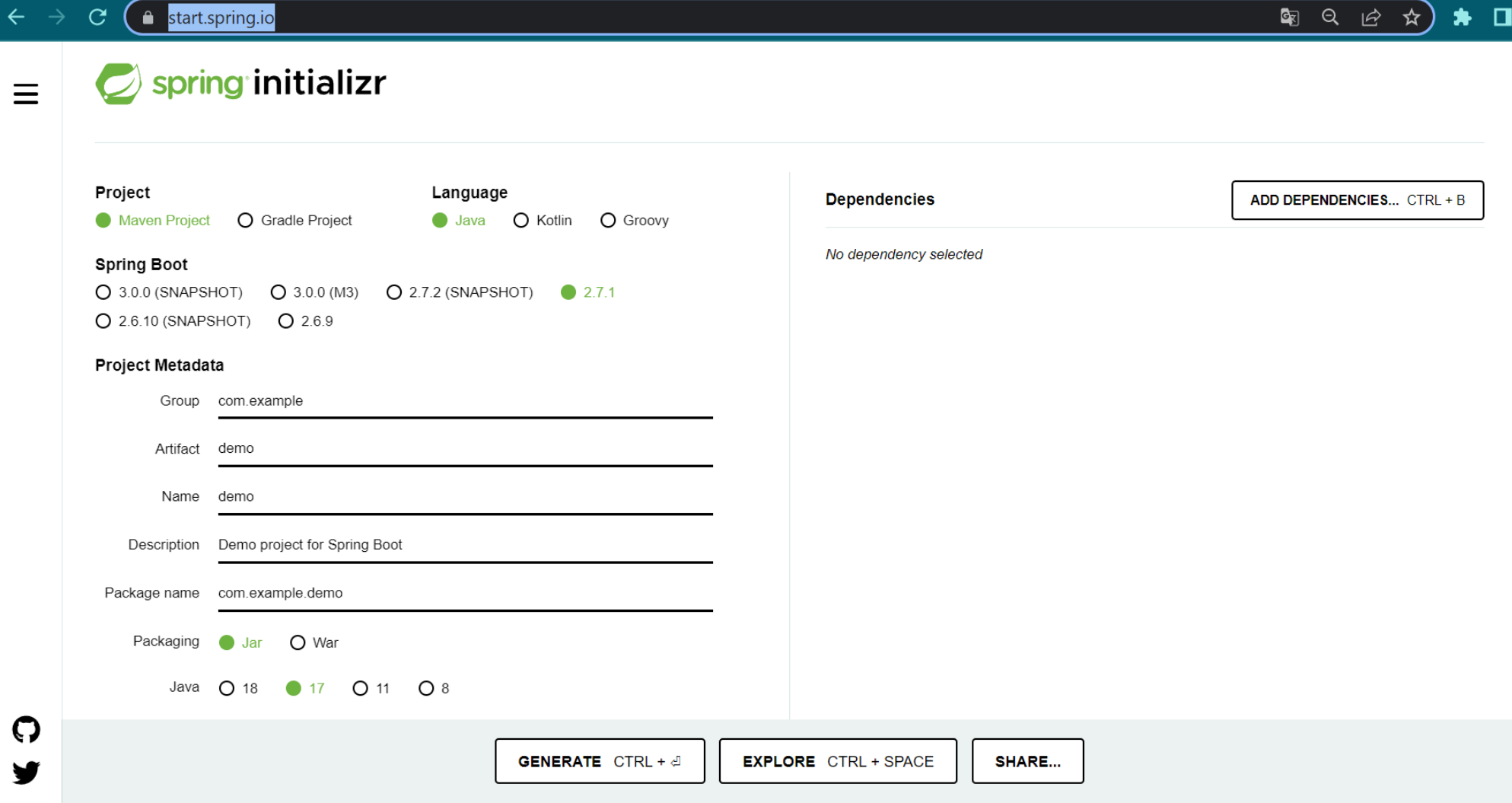
- ☐ Embebed Application Server Tomcat
- ☐ Integración con Herramientas y Tecnologías
- ☐ Herramientas de producción, monitoreo
- ☐ Manejo de configuración
- ☐ DevTools, hot deployment ante cambios
- ☐ No XML, No generación de Código Fuente



Como está compuesto



Creando un-Proyecto Spring Boot Online: <http://start.spring.io>



The screenshot shows the Spring Initializr web application in a browser. The URL bar shows start.spring.io. The page has a header with the Spring logo and a hamburger menu. The main content area is divided into three sections: Project, Spring Boot, and Project Metadata. The Project section has radio buttons for Maven Project (selected), Gradle Project, and Language (Java, Kotlin, Groovy). The Spring Boot section has radio buttons for versions: 3.0.0 (SNAPSHOT), 3.0.0 (M3), 2.7.2 (SNAPSHOT), 2.7.1 (selected), 2.6.10 (SNAPSHOT), and 2.6.9. The Project Metadata section has input fields for Group (com.example), Artifact (demo), Name (demo), Description (Demo project for Spring Boot), and Package name (com.example.demo). There are also radio buttons for Packaging (.Jar selected, War) and Java version (18, 17 selected, 11, 8). On the right, there is a Dependencies section with a button 'ADD DEPENDENCIES... CTRL + B' and the text 'No dependency selected'. At the bottom, there are three buttons: 'GENERATE CTRL + G', 'EXPLORE CTRL + SPACE', and 'SHARE...'. Social media icons for GitHub and Twitter are in the bottom left corner.

Project

☒ Maven Project ☐ Gradle Project

Language

☒ Java ☐ Kotlin ☐ Groovy

Spring Boot

☐ 3.0.0 (SNAPSHOT) ☐ 3.0.0 (M3) ☐ 2.7.2 (SNAPSHOT) ☒ 2.7.1 ☐ 2.6.10 (SNAPSHOT) ☐ 2.6.9

Project Metadata

Group

Artifact

Name

Description

Package name

Packaging ☒ .Jar ☐ War

Java ☐ 18 ☒ 17 ☐ 11 ☐ 8

Dependencies

[ADD DEPENDENCIES... CTRL + B](#)

No dependency selected

GENERATE CTRL + G **EXPLORE CTRL + SPACE** **SHARE...**

II) JPA, Java Persistent API

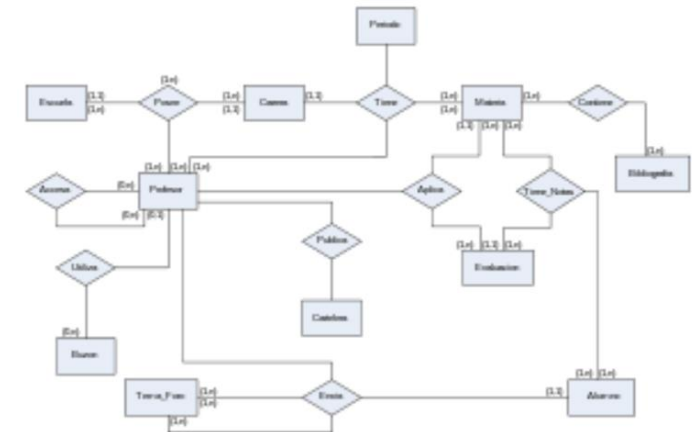
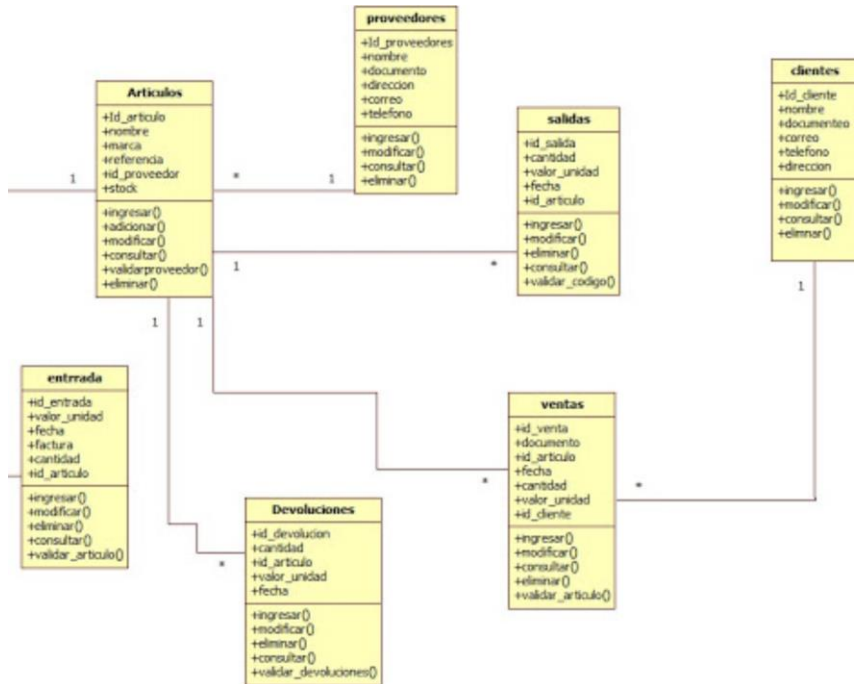
@UPC

Agenda

1. ORM
2. ORM-JPA
3. Persistence Provider, Entity Manager

1. ORM

Object Relational Mapping, es una técnica de programación para convertir datos entre bases de datos relacional y objetos java.



2. ORM - JPA

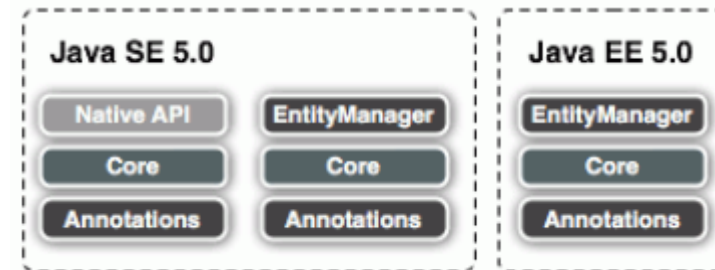
- ❖ El proceso de asignación de objetos Java a tablas de bases de datos y viceversa se llama "**mapeo objeto-relacional**" (ORM).
- ❖ El Java Persistence API (JPA) es un método para ORM. A través de JPA el desarrollador puede asignar, almacenar, actualizar y recuperar datos de bases de datos relacionales a objetos Java y viceversa
- ❖ JPA permite al desarrollador trabajar directamente con los objetos en lugar de sentencias SQL.
- ❖ JPA tiene varias implementaciones disponibles: Eclipse Link, **Hibernate**, TopLink, etc.

2. Entidades-> JPA

Entity beans son deprecados a partir de JEE5

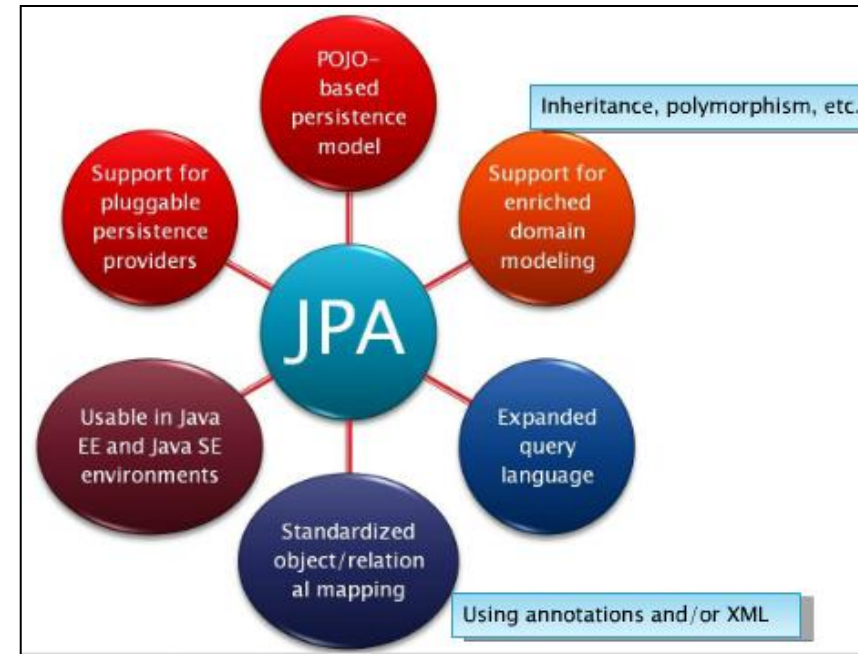
Y ahora hay un nuevo estándar:

- **JPA** – Java Persistence API
- No Application Server
- Provides full O/R mapping
- JPA: Es un estándar ORM e interfaces de manejo de persistencia desde Java EE 5.0 y soportado por la mayoría de vendedores:



2. JPA

- ❑ Basado en POJO
- ❑ No requiere interfaces o subclases
- ❑ Basado en Anotaciones - Opcional DD
- ❑ Soportan modelos de herencia
- ❑ EJBQL extendido, JPAQL



2. Proceso de desarrollo

Top-down: Comenzamos con el desarrollo del modelo de entidades **Java** y el fichero de mapeo, y generamos el script de la **BBDD** a partir de estos dos componentes (hbm2dll).

Bottom-up: Se parte de un modelo de datos **ER** y mediante la utilidad hbm2hbmxml se generan las clases **java** mapeadas.



JPA
Buddy

3. Persistence Mapping

O/R Mapping Metadata como annotations

- Table mappings: @Table

Column mappings: @Column

Identifier (Primary-Key): @Id

Relationships:

- @ManyToOne, @OneToOne,
- @OneToMany, @ManyToMany

Inheritance: @Inheritance

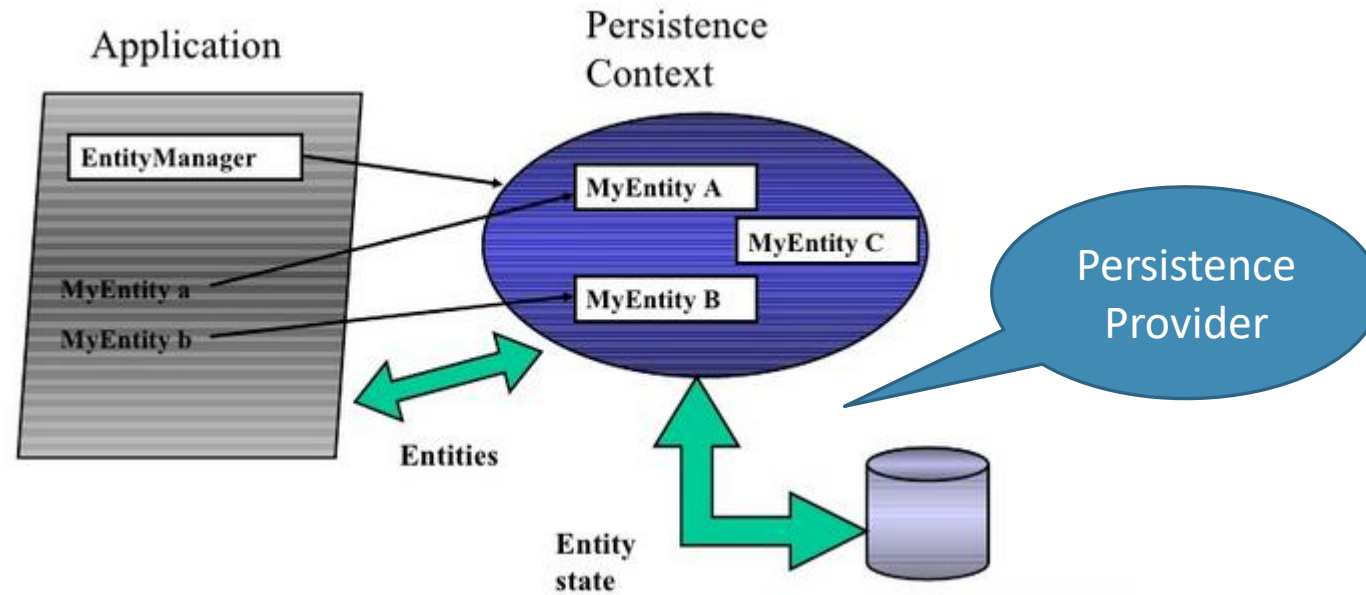
Ejemplo

```
create table EMPLOYEE_TP  
(  
    EMPLOYEE_ID Number NOT NULL AUTO_INCREMENT,  
    EMPLOYEE_NAME varchar(50),  
    SALARY Number,  
    PRIMARY KEY ('EMPLOYEE_ID'),  
);
```

```
@Entity
@Table(name="EMPLOYEE_TP")
public class Employee{
    @Id @GeneratedValue
    @Column(name="EMPLOYEE_ID")
    private int id;
    @Column(name="EMPLOYEE_NAME")
    private String name;
    @Column(name="SALARY")
    private int salary;

    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    .....
}
```

3. Persistence Provider – Entity Manager



JPAQL

CRUD

```
EntityManager em = emf.createEntityManager();  
em.getTransaction().begin();  
AuditedDataPoint adp = new AuditedDataPoint( "Chris" );  
em.persist( adp );  
em.getTransaction().commit();  
adpId = adp.getId();  
em.close();
```

```
public void deleteCounter()  
{  
    EntityManager em = emf.createEntityManager();  
    Counters counters = new Counters();  
    counters = em.find(Counters.class, id3);  
    Assert.assertNotNull(counters);  
    Assert.assertNotNull(counters.getCounter());  
    em.remove(counters);  
  
    EntityManager em1 = emf.createEntityManager();  
    counters = em1.find(Counters.class, id3);  
    Assert.assertNull(counters);  
  
    em.close();  
}
```

```
SELECT OBJECT( e ) FROM Employee AS e
```

```
SELECT e.department  
FROM Employee e
```

```
SELECT p.number  
FROM Employee e JOIN e.phones p  
WHERE e.department.name = 'NA42' AND  
p.type = 'Cell'
```

```
SELECT e  
FROM Employee e  
WHERE e.department.name = 'NA42' AND  
e.address.state IN ('NY','CA')
```

```
SELECT d, COUNT(e), MAX(e.salary), AVG(e.salary)  
FROM Department d JOIN d.employees e  
GROUP BY d  
HAVING COUNT(e) >= 5
```

```
SELECT e  
FROM Employee e  
WHERE e.department.name = 'NA42' AND  
e.address.state IN ('NY','CA')
```

```
SELECT DISTINCT d  
FROM Department d, Employee e  
WHERE d = e.department
```

```
SELECT p.number  
FROM Employee e, Phone p  
WHERE e = p.employee AND  
e.department.name = 'NA42' AND  
p.type = 'Cell'
```

```
SELECT d  
FROM Employee e JOIN e.department d
```

```
SELECT e  
FROM Employee e  
WHERE e.department = :dept AND  
e.salary > :base
```

```
SELECT e  
FROM Employee e  
WHERE e.department = ?1 AND  
e.salary > ?2
```

III) Spring Data

Spring Data

La misión de Spring Data es proporcionar un modelo de programación familiar y coherente basado en Spring para el acceso a los datos.

Facilita el uso de tecnologías de acceso a datos, bases de datos relacionales y no relacionales, marcos de reducción de mapas y servicios de datos basados en la nube.

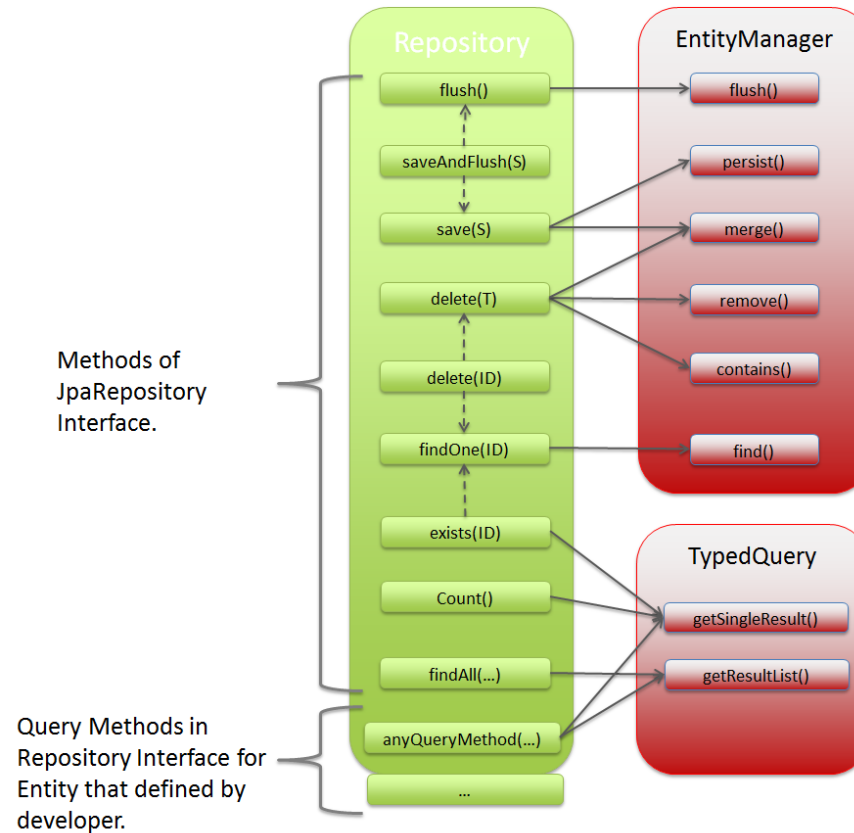
- Potente **repositorio** y abstracciones personalizadas de mapeo de objetos

Fuente: <https://spring.io/projects/spring-data>

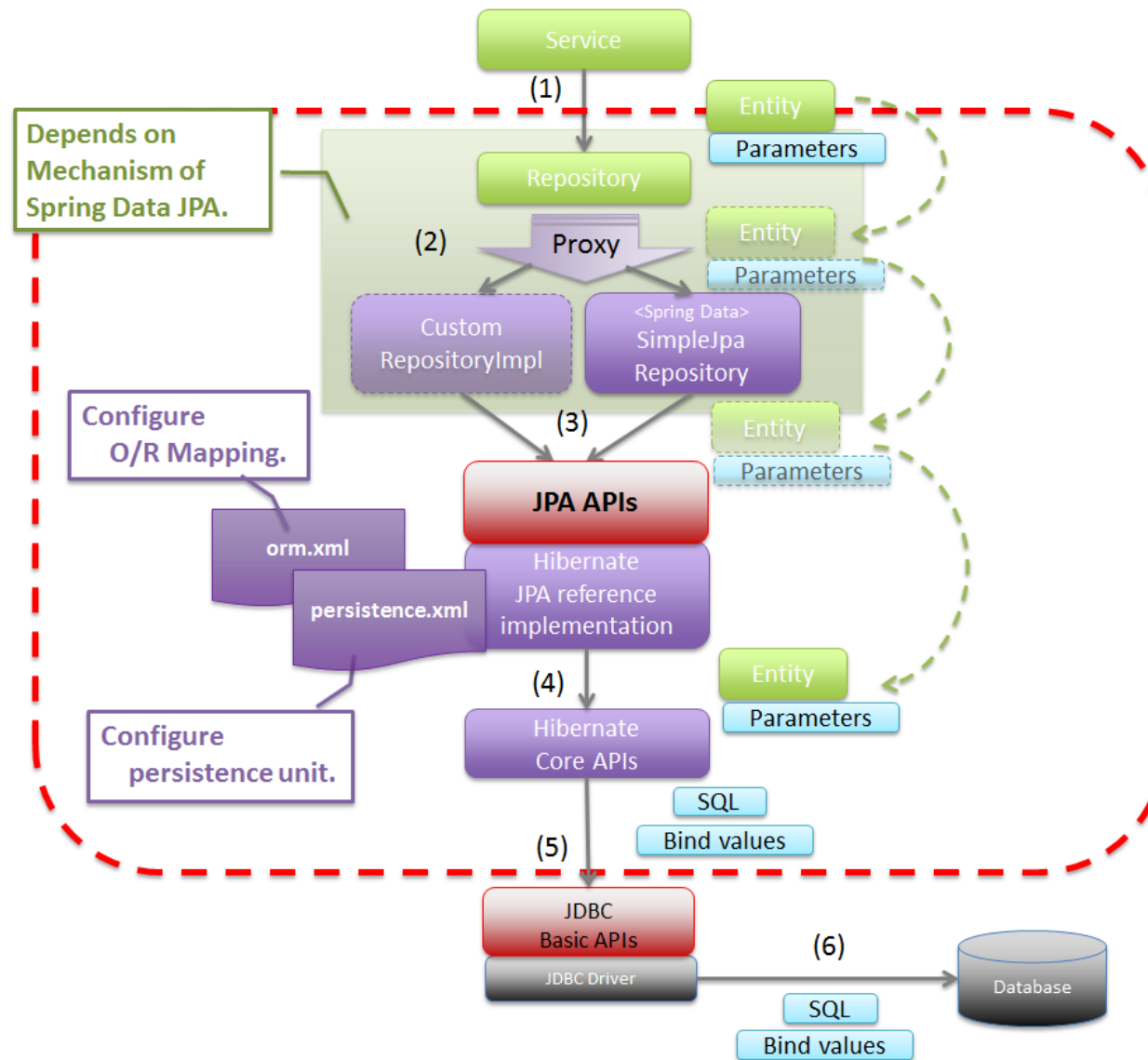
Módulos Principales

- Spring Data JDBC : soporte de repositorio de Spring Data para JDBC.
- Spring Data JPA : soporte de repositorio de Spring Data para JPA.
- Spring Data LDAP : soporte de repositorio de Spring Data para Spring LDAP .
- Spring Data MongoDB y Apache Cassandra : repositorios y compatibilidad con objetos y documentos basados en Spring para MongoDB y Cassandra con alta disponibilidad.
- Spring Data Redis y Apache Geode : fácil configuración y acceso a Redis y Geode desde las aplicaciones de Spring, altamente consistentes y baja latencia.

Spring Data Repository

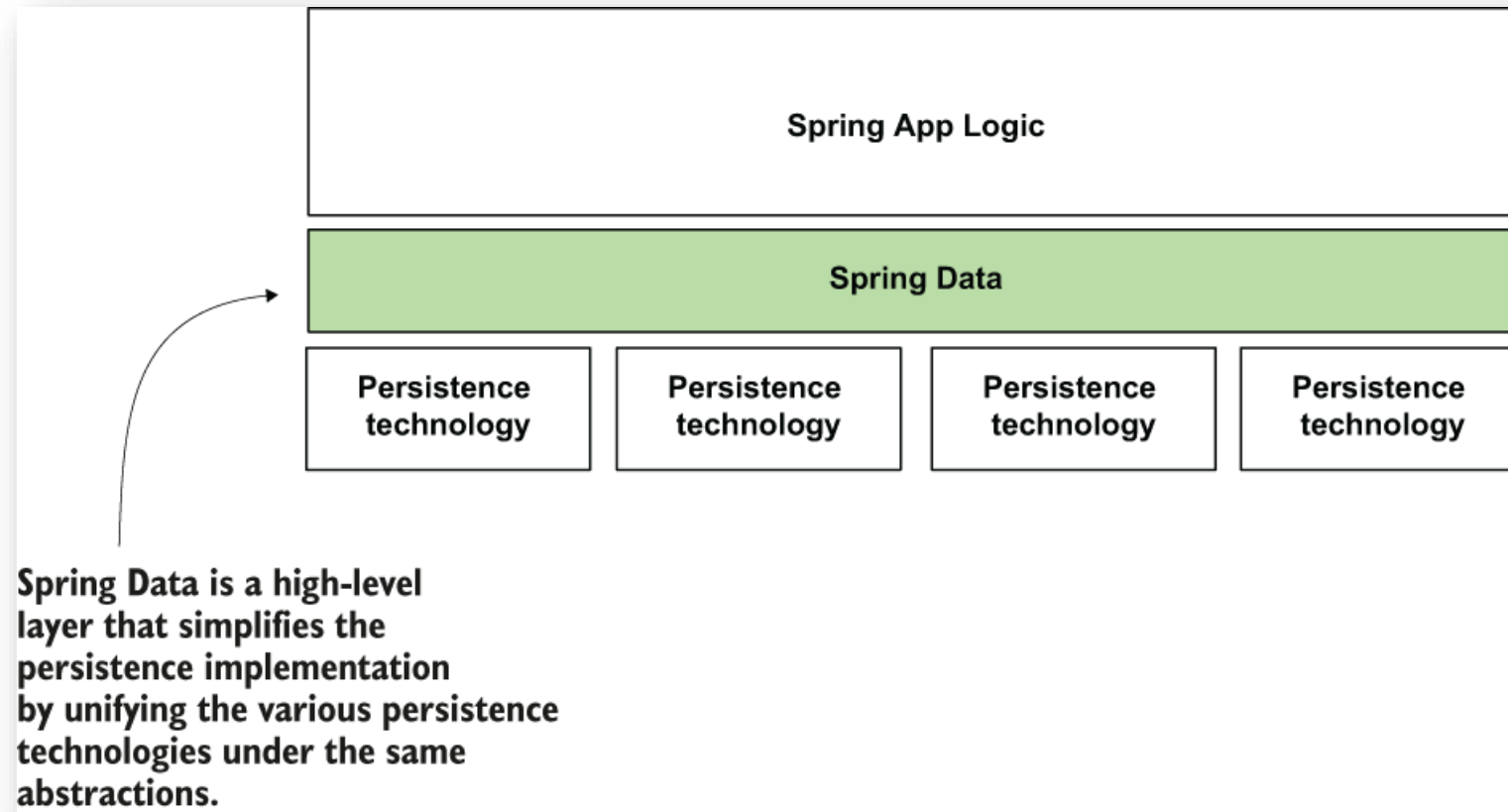


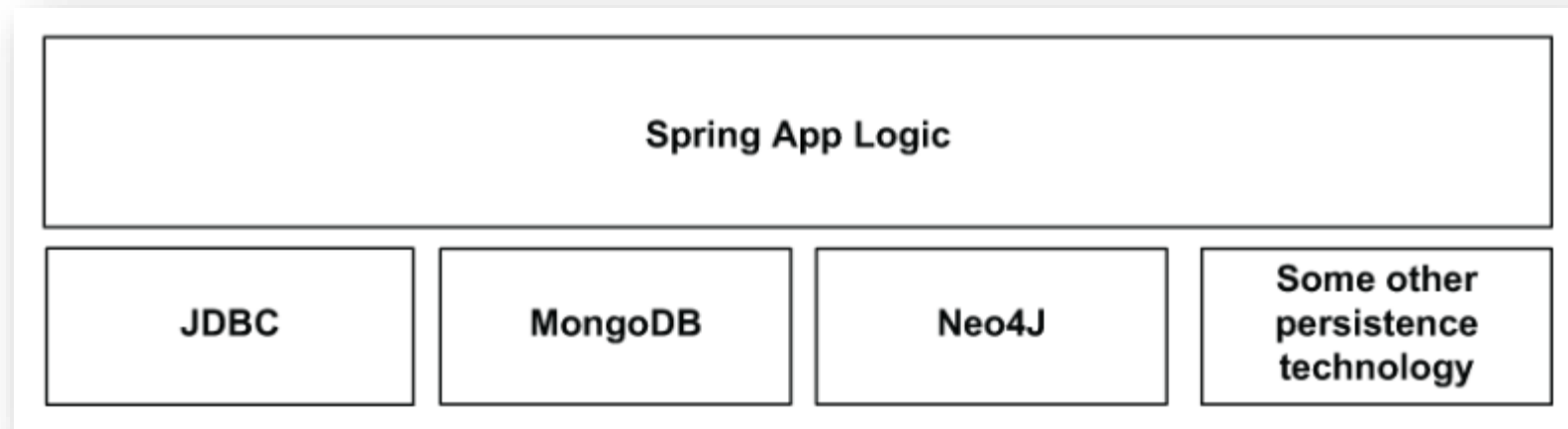
Fuente: https://2.bp.blogspot.com/-1JFCaxkaRWA/XC44BMftRSI/AAAAAAAAAFUk/zrg3KOWg2OEzshRn1QJszH-JaYWv2fcwCK4BGAYYCw/s1600/dataaccess_jpa_api-mapping.png



Fuente: https://2.bp.blogspot.com/-EebmrWpAqO4/W9vzgl61vtI/AAAAAAAAEKw/Kb5YL-e8Ja8ENTWx445hvovrwTlpo3iGAClCBGAs/s1600/dataaccess_jpa_basic_flow.png

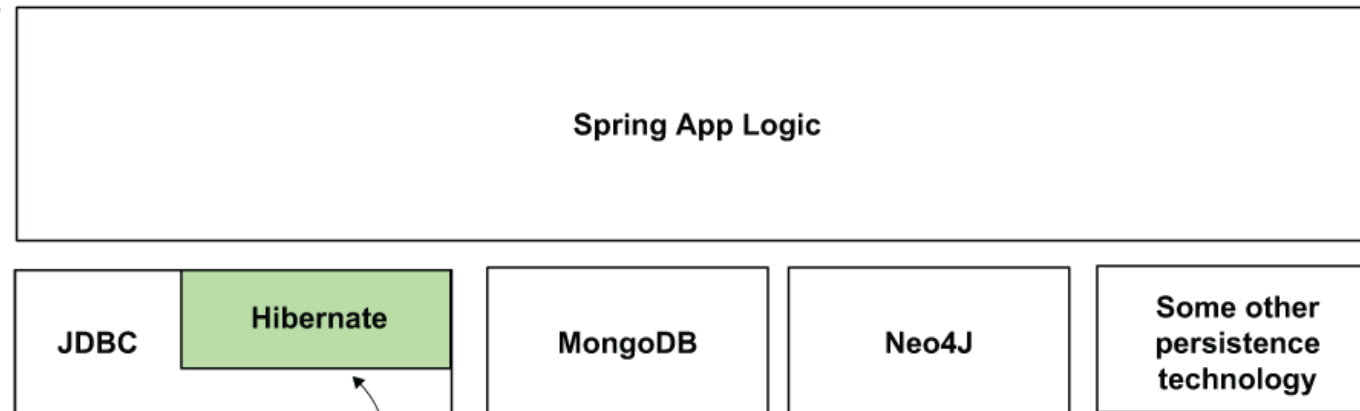
Spring Data es un proyecto del ecosistema Spring que simplifica el desarrollo de la capa de persistencia al proporcionar implementaciones de acuerdo con la tecnología de persistencia que utilizamos.



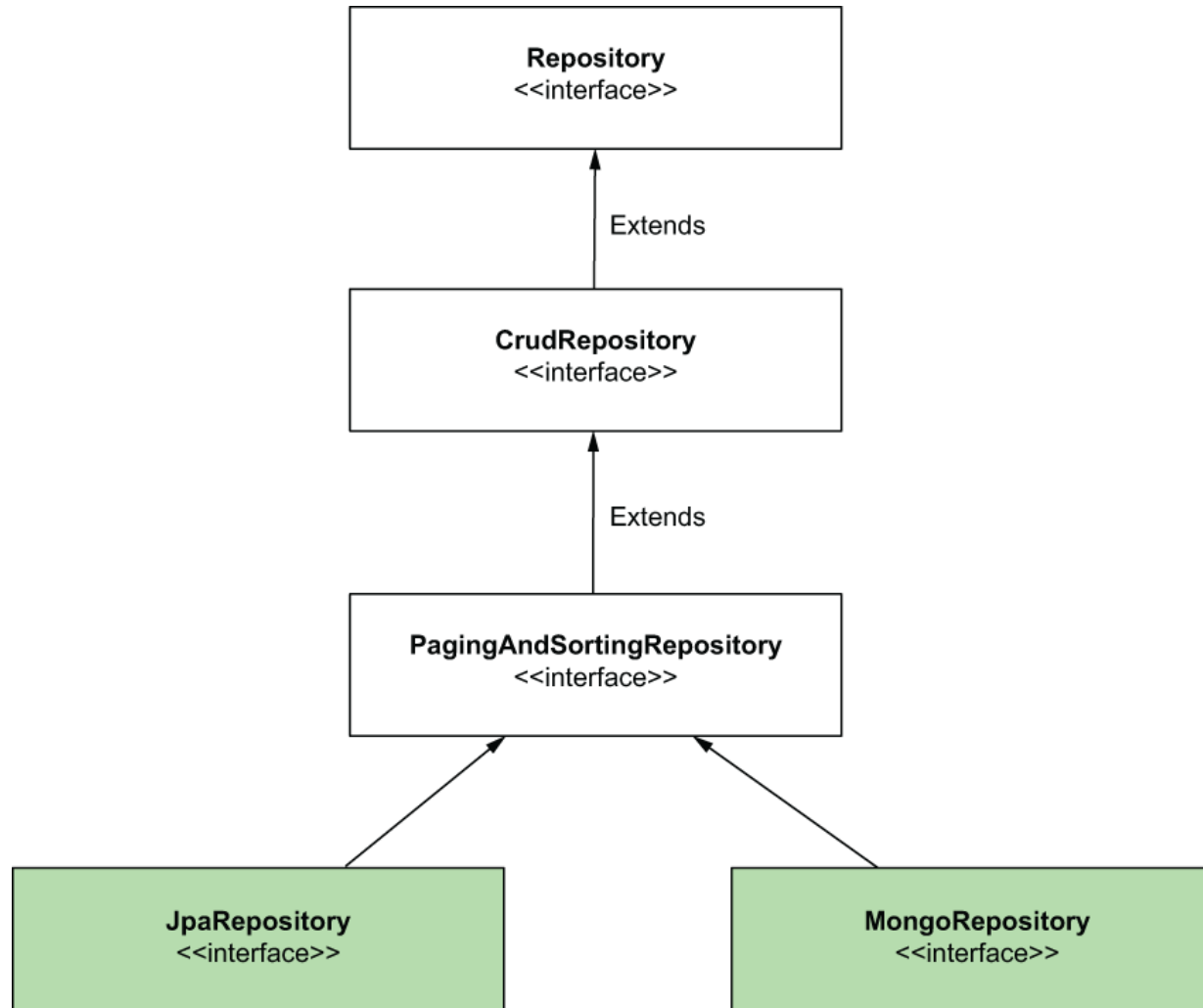


The Spring app can use **JDBC** directly or an **ORM** framework such as **Hibernate**.

Spring Data is a high-level layer that simplifies the persistence implementation by unifying the various technologies under the same abstractions.



Hibernate is an **ORM** persistence framework that relies on **JDBC** and simplifies some aspects of working with persisted data.



```
interface UserRepository extends JpaRepository<User,  
Integer> {}
```

No más DAO

El objetivo del repositorio de Spring Data es reducir significativamente la cantidad de código repetitivo requerido para implementar capas de acceso a datos para varios tipos de persistencia.

5. Spring Data @Query

```
@Table(name = "users")
@Entity
class User {
    @Id
    @GeneratedValue
    private Integer id;

    private String name;
    private Integer age;
    private ZonedDateTime birthDate;
    private Boolean active;

    // standard getters and setters
}

interface UserRepository extends JpaRepository<User, Integer> {}
```

```
List<User> findByName(String name);
```

Y podemos agregar Is o Equals para mejorar la legibilidad:

```
List<User> findByNameIs(String name);
```

```
List<User> findByNameEquals(String name);
```

- Podemos usar palabras clave True y False para agregar condiciones de igualdad para tipos booleanos :

```
List<User> findByActiveTrue();  
List<User> findByActiveFalse();
```

- Podemos encontrar nombres que comiencen con un valor usando StartingWith :

```
List<User> findByNameStartingWith(String prefix);  
Aproximadamente, esto se traduce como "WHERE name LIKE 'value%' ".
```

- Si queremos nombres que terminen con un valor, EndingWith es lo que queremos:

```
List<User> findByNameEndingWith(String suffix);
```

- O podemos encontrar qué nombres contienen un valor con Containing :

```
List<User> findByNameContaining(String infix);  
Tenga en cuenta que todas las condiciones anteriores se denominan expresiones de patrón predefinidas.  
Por lo tanto, no necesitamos agregar el operador % dentro del argumento cuando se llama a estos  
métodos
```

- Esta legibilidad adicional es útil cuando necesitamos expresar la desigualdad:

```
List<User> findByNameIsNot(String name);
```

Esto es bastante más legible que findByNameNot(String) !

- También podemos usar la palabra clave IsNull para agregar criterios IS NULL a la consulta:

```
List<User> findByNameIsNull();
```

```
List<User> findByNameIsNotNull();
```

- *Pero supongamos que estamos haciendo algo más complejo.*

Digamos que necesitamos buscar a los usuarios cuyos nombres comienzan con a , contienen b y terminan con c .

Para eso, podemos agregar nuestro propio LIKE con la palabra clave Like :

```
List<User> findByNameLike(String likePattern);
```

Y luego podemos entregar nuestro patrón LIKE cuando llamamos al método:

```
String likePattern = "a%b%c";
```

```
userRepository.findByNameLike(likePattern);
```

Palabras clave de condición de comparación

Además, podemos usar las palabras clave `LessThan` y `LessThanEqual` para comparar los registros con el valor dado usando los operadores `<` y `<=` :

```
List<User> findByAgeLessThan(Integer age);  
List<User> findByAgeLessThanEqual(Integer age);
```

En la situación opuesta, podemos usar las palabras clave `GreaterThan` y `GreaterThanEqual` :

```
List<User> findByAgeGreaterThan(Integer age);  
List<User> findByAgeGreaterThanEqual(Integer age);
```

O podemos encontrar usuarios que tengan entre dos edades con `Between` :

```
List<User> findByAgeBetween(Integer startAge, Integer endAge);
```

También podemos proporcionar una colección de edades para que coincidan con el uso de `In` :

```
List<User> findByAgeIn(Collection<Integer> ages);
```

Dado que conocemos las fechas de nacimiento de los usuarios, es posible que queramos consultar a los usuarios que nacieron antes o después de una fecha determinada.

Usaríamos Antes y Después para eso:

```
List<User> findByBirthDateAfter(ZonedDateTime birthDate);
```

```
List<User> findByBirthDateBefore(ZonedDateTime birthDate);
```

Expresiones de condiciones múltiples

Podemos combinar tantas expresiones como necesitemos usando las palabras clave And y Or :

```
List<User> findByNameOrBirthDate(String name, ZonedDateTime birthDate);
```

```
List<User> findByNameOrBirthDateAndActive(String name, ZonedDateTime birthDate, Boolean active);
```

El orden de precedencia es Y luego O , al igual que Java.

Podríamos pedir que los usuarios sean ordenados alfabéticamente por su nombre usando OrderBy :

```
List<User> findByNameOrderByName(String name);  
List<User> findByNameOrderByNameAsc(String name);
```

El orden ascendente es la opción de clasificación predeterminada, pero podemos usar Desc en su lugar para ordenarlos a la inversa:

```
List<User> findByNameOrderByNameDesc(String name);
```

```
CREATE PROCEDURE GET_TOTAL_CARS_BY_MODEL(IN model_in VARCHAR(50), OUT  
count_out INT) BEGIN SELECT COUNT(*) into count_out from car WHERE  
model = model_in; END
```

```
@Procedure("GET_TOTAL_CARS_BY_MODEL")  
int getTotalCarsByModel(String model);
```

CrudRepository prove operaciones CRUD genéricas sobre un repositorio
Para un tipo específico de Entidad, idéntico a **JpaRepository**

```
public interface CrudRepository<T, ID extends Serializable>
    extends Repository<T, ID> {

    <S extends T> S save(S entity);

    T findOne(ID primaryKey);

    Iterable<T> findAll();

    Long count();

    void delete(T entity);

    boolean exists(ID primaryKey);

    // ... more functionality omitted.
}
```


Su uso, su creación:

```
interface ArticleRepository extends JpaRepository<Article, Integer> {}
```

Se inyecta en donde lo necesiten:

@Autowired

```
private ArticleRepository articleRepository;
```

@Transactional con CrudRepository

Es por defecto true.

Se puede configurar:

```
public interface ArticleRepository extends CrudRepository<Article, Long> {  
    @Override  
    @Transactional(timeout = 8)  
    Iterable<Article> findAll();  
}
```

```
@Query(" SELECT MAX(eventId) AS eventId FROM Event ")  
Long lastProcessedEvent();
```

```
@Query("SELECT t FROM Tutorial t")  
List<Tutorial> findAll();
```

```
@Query("SELECT t FROM Tutorial t WHERE t.published=true")  
List<Tutorial> findByPublished();
```

Pasos a Seguir

```
@Entity
@Table(name="articles")
public class Article implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    @Column(name="article_id")
    private long articleId;
    @Column(name="title")
    private String title;
    @Column(name="category")
    private String category;
    public long getArticleId() {
        return articleId;
    }
    public void setArticleId(long articleId) {
        this.articleId = articleId;
    }
    public String getTitle() {
        return title;
    }
    public void setTitle(String title) {
        this.title = title;
    }
    public String getCategory() {
        return category;
    }
    public void setCategory(String category) {
        this.category = category;
    }
}
```

Repositorio

```
public interface ArticleRepository extends CrudRepository<Article, Long> { }
```

Ejemplo: Si a entidad tiene la propiedad "title" y sus métodos estándar *getTitle* y *setTitle*, podríamos definir el método ***findByTitle en la interface DAO***; esto deberá generar el correcto query automáticamente:

```
import java.util.List;
import org.springframework.data.repository.CrudRepository;
import com.concretepage.entity.Article;
public interface ArticleRepository extends CrudRepository<Article, Long> {
    List<Article> findByTitle(String title);
    List<Article> findDistinctByCategory(String category);
    List<Article> findByTitleAndCategory(String title, String category);
}
```

Detalle

```
public interface UserRepository extends Repository<User, Long> {  
    List<User> findByEmailAddressAndLastname(String emailAddress, String lastname);  
}
```

Crearía automáticamente:

```
select u from User u where u.emailAddress = ?1 and u.lastname = ?2
```


Ejemplo: Custom Repository @Query

```
public interface ArticleRepository extends CrudRepository<Article, Long> {  
    @Query("SELECT a FROM Article a WHERE a.title=:title and a.category=:category")  
    List<Article> fetchArticles(@Param("title") String title, @Param("category") String category);  
}
```

```
@Service
public class ArticleService {

    @Autowired
    private ArticleRepository articleRepository;

    public Article getArticleById(long articleId) {
        Article obj = articleRepository.findById(articleId).get();
        return obj;
    }

    public List<Article> getAllArticles(){
        List<Article> list = new ArrayList<>();
        articleRepository.findAll().forEach(e -> list.add(e));
        return list;
    }

    public synchronized boolean addArticle(Article article){
        List<Article> list = articleRepository.findByTitleAndCategory(article.getTitle(), article.getCategory());
        if (list.size() > 0) {
            return false;
        } else {
            articleRepository.save(article);
            return true;
        }
    }

    public void updateArticle(Article article) {
        articleRepository.save(article);
    }

    public void deleteArticle(int articleId) {
        articleRepository.delete(getArticleById(articleId));
    }
}
```

Injectando el
repositorio

usando el
repositorio

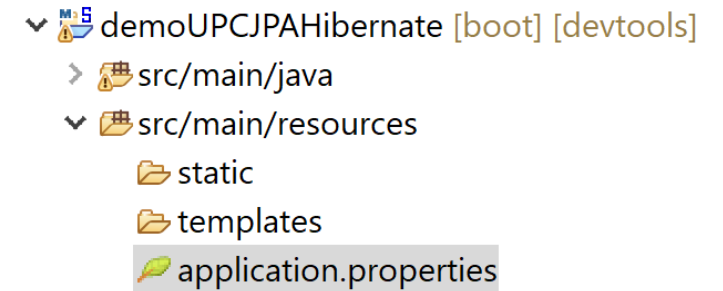
Más custom Queries

```
@Query("SELECT f FROM Foo f WHERE LOWER(f.name) = LOWER(:name)")
Foo retrieveByName(@Param("name") String name);
```

Completo:

```
public interface ProductoRepository extends CrudRepository<Producto, Long>{
    @Query("SELECT a FROM Producto a WHERE a.codigo=:codigo and a.precio=:precio")
    List<Producto> fetchProductos(@Param("codigo") Long codigo, @Param("precio") double precio);
    @Query(value = "select codigo, descripcion ,precio from PRODUCTO_TP where codigo = ?1", nativeQuery = true)
    Producto findProductNative(Long codigo);
}
```

application.properties



```
1 ## Spring DATASOURCE (DataSourceAutoConfiguration & DataSourceProperties)
2 spring.datasource.url = jdbc:mysql://localhost:3306/bank?useSSL=false
3 spring.datasource.username = root
4 spring.datasource.password = root
5
6
7 ## Hibernate Properties
8 # The SQL dialect makes Hibernate generate better SQL for the chosen database
9 spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL5InnoDBDialect
10
11 # Hibernate ddl auto (create, create-drop, validate, update)
12 spring.jpa.hibernate.ddl-auto = update
```