El concepto de JPA Persist es uno de los conceptos más elementales de JPA pero a veces cuesta entender a detalle como funciona a la hora de persistir datos contra la base de datos . Vamos a ver un ejemplo sencillo que se encargue de persistir la información partiendo de la entidad Libro.

```
package com.arquitecturajava.dominio;

import java.util.Date;

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;
	private Date fecha;
	private int precio;
	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 Date getFecha() {
		return fecha;
	}
	public void setFecha(Date fecha) {
		this.fecha = fecha;
	}
	public int getPrecio() {
		return precio;
	}
	public void setPrecio(int precio) {
		this.precio = precio;
	}
	public Libro(String isbn, String titulo, String autor, Date
fecha, int precio) {
		super();
		this.isbn = isbn;
		this.titulo = titulo;
		this.autor = autor;
		this.fecha = fecha;
		this.precio = precio;
	}
```

```
	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;
	}
```

```
	@Override
	public String toString() {
		return "Libro [isbn=" + isbn + ", titulo=" + titulo +
", autor=" + autor + ", fecha=" + fecha + ", precio="
				+ precio + "]";
	}
}
```

Acabamos de construir la clase Libro es momento de persistirla con un EntityManager . Para ello vamos a construir un programa Main y encargarnos de usar el EntityManager y EntityManagerFactory para salvar la información en la base de datos:

```
package com.arquitecturajava.main;

import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

import com.arquitecturajava.dominio.Libro;

public class Principal2Insertar {

	public static void main(String[] args) {
		// unidad de persistencia
		EntityManagerFactory emf=
Persistence.createEntityManagerFactory("biblioteca");
		EntityManager em= emf.createEntityManager();
		String textoFecha= "1/01/2020";
```

```
                SimpleDateFormat ffecha= new
SimpleDateFormat("d/M/yyyy");
                Date fecha=null;
                try {
                        fecha= ffecha.parse(textoFecha);
                } catch (ParseException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }
                Libro l1= new Libro ("2A","JPA","Gema",fecha,3);
                em.getTransaction().begin();
                em.persist(l1);
                em.getTransaction().commit();

        }

}
```

Todo correcto . Mucha gente piensa que cuando construimos el Libro , si modificamos alguna propiedad a posteriori se lanzarán varias consultas SQL con cada uno de los cambios que se produce, esto no es así ya que el Libro mantiene su estado en memoria y el EntityManager sabe que propiedades han cambiado y por lo tanto que hay que persistir . Imaginemonos que modificamos un poco el código y cambiamos una propiedad de valor (el titulo del libro).

```
                em.getTransaction().begin();
                em.persist(l1);
                l1.setTitulo("JPA2");
                em.merge(l1);
                em.getTransaction().commit();
```

En este caso al cambiar el valor del titulo del Libro lactualizamos el estado del objeto y

lanzamos una SQL adicional que se encargue de una vez insertado el objeto en la base de datos de actualizarlo.

```
Hibernate: insert into libros (autor, fecha, precio, titulo, isbn)
values (?, ?, ?, ?, ?)
Hibernate: update libros set autor=?, fecha=?, precio=?, titulo=?
where isbn=?
```

¿Que pasaría en el caso de que volvamos a cambiar el estado del objeto a su valor inicial? .

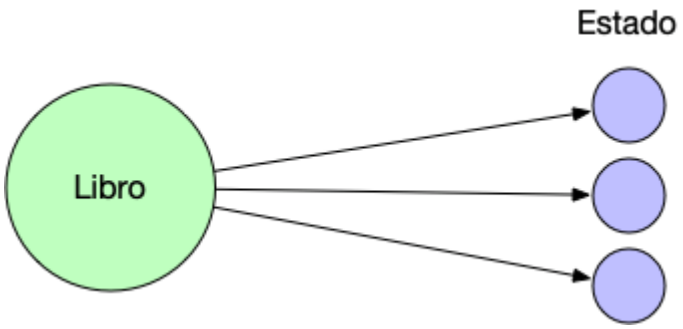
```
em.getTransaction().begin();
em.persist(l1);
l1.setTitulo("JPA2");
em.merge(l1);
l1.setTitulo("JPA");
em.merge(l1);
em.getTransaction().commit();
```

En este caso el framework de Persistencia que mantiene el estado del objeto en memoria es capaz de darse cuenta de que realmente no hay cambios a nivel del estado inicial del objeto y por lo tanto únicamente lanza una consulta de inserción.

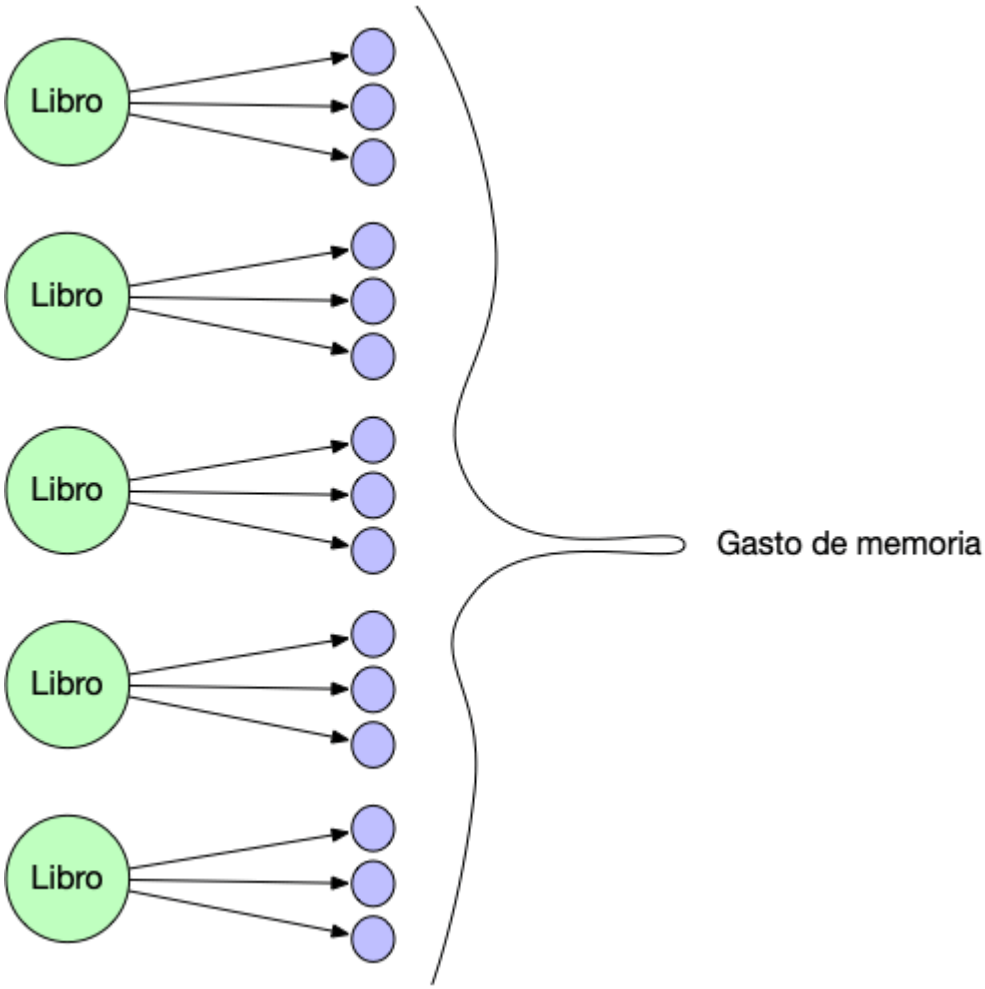
```
Hibernate: insert into libros (autor, fecha, precio, titulo, isbn)
values (?, ?, ?, ?, ?)
```

JPA Persist Flushing y Persistencia (Premium)

Muchas personas me preguntan para que sirve realizar una operación de flush o flushing dentro de JPA . A veces es difícil entender para que sirve este método de JPA . Vamos a intentar aclararlo con el ejemplo que tenemos actualmente . En este caso estamos manejando un Libro . A este Libro podemos realizarle los cambios que queramos y esos cambios serán almacenados en memoria .

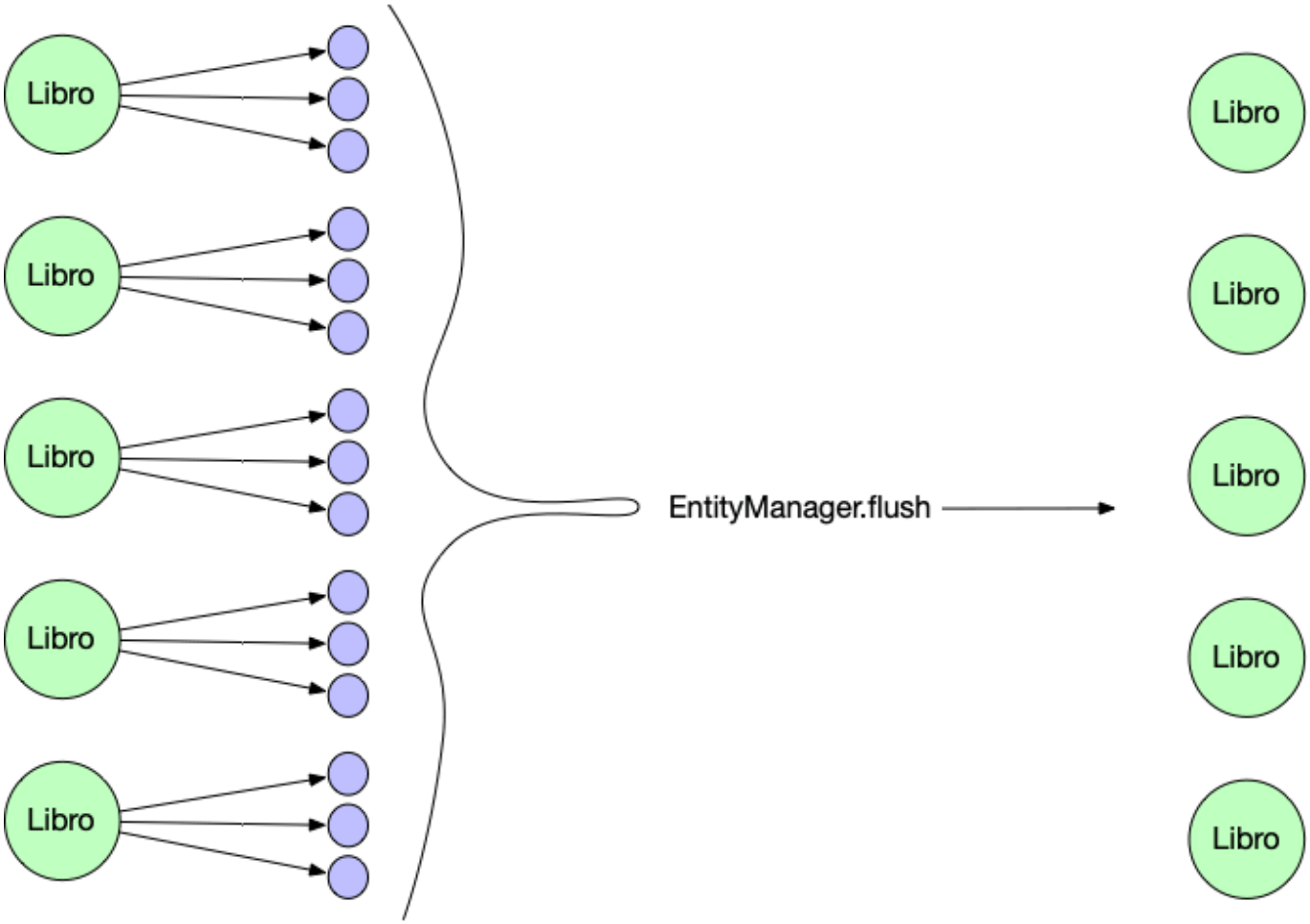


Hasta aquí no hay ningún problema pero si tuvieramos que realizar operaciones de modificaciones de estado con un conjunto elevado de Libros nos podríamos encontrar con que los consumos de memoria se incrementan .



Para solventar este problema es para lo que existe el método flush de JPA ya que lanza las

consultas de actualización a la base de datos y descarga los objetos que mantienen el estado de la memoria del programa permitiendo afinar mejor el rendimiento de la aplicación en casos exigentes.



Por lo tanto cuando nosotros queramos ,podemos usar el método flush y descargar el estado de los objetos y sus cambios de la memoria a nivel de EntityManager de esta forma en operaciones pesadas podemos decidir en que momento vamos actualizando la base de datos. Vamos a modificar nuestro código:

```
em.getTransaction().begin();
  em.persist(l1);
  l1.setTitulo("JPA2");
  em.merge(l1);
```

```
System.out.println("termina merge");
System.out.println("empieza a ejecutar la consulta");
em.flush();
System.out.println("flushing");
l1.setTitulo("JPA");
em.merge(l1);
em.getTransaction().commit();
System.out.println("termina de ejecutar la consulta");
```

En este caso en cuando solicitamos realizar un flush las consultas serán lanzadas contra la base de datos y el estado comenzará a gestionarse de cero no se esperará a que la transacción este finalizando.

```
termina merge
empieza a ejecutar la consulta
Hibernate: insert into libros (autor, fecha, precio, titulo, isbn)
values (?, ?, ?, ?, ?)
Hibernate: update libros set autor=?, fecha=?, precio=?, titulo=?
where isbn=?
flushing
Hibernate: update libros set autor=?, fecha=?, precio=?, titulo=?
where isbn=?
termina de ejecutar la consulta
```

Hay en situaciones que manejar flush puede mejorar de forma importante el rendimiento al descargar parte del estado de la memoria.

Otros artículos relacionados

1. Ejemplo de JPA (Java Persistence API)
2. ¿JPA vs Hibernate?

3. JPA Orphan Removal y como usarlo
4. Spring Boot JPA y su configuración