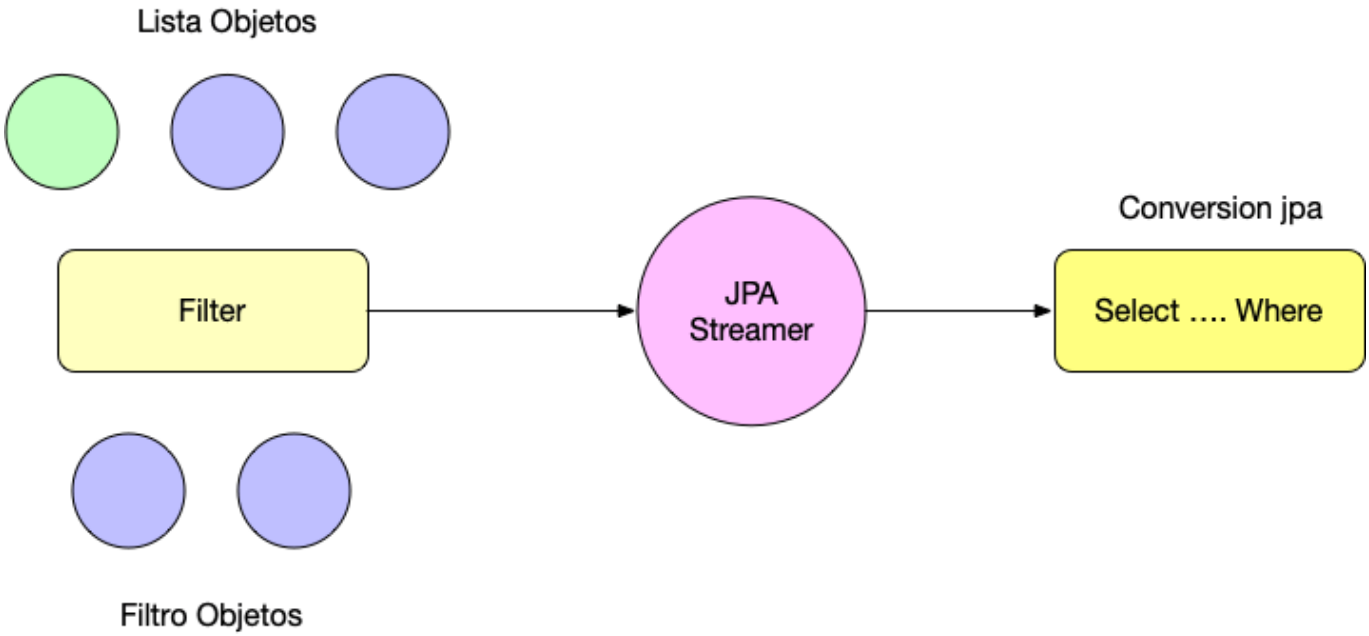
JPA Streamer es una librería que nos permite trabajar con JPA con consultas dinámicas construidas a través de expresiones Lambda y Java Streams .



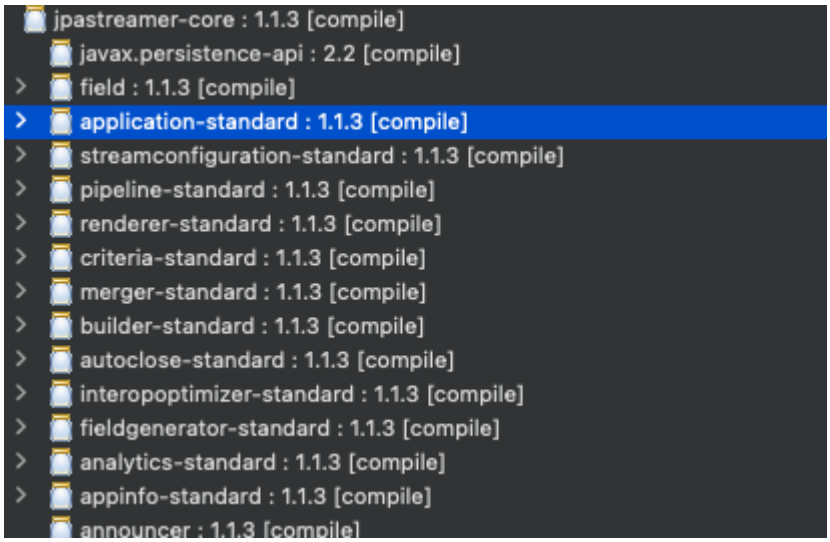

¿Como podemos sacar partido de esta librería?. Bueno lo primero que tendremos que hacer es instalarla.

```
<dependency>
                <groupId>com.mysql</groupId>
                <artifactId>mysql-connector-j</artifactId>
                <version>8.0.32</version>
        </dependency>

        <dependency>
                <groupId>com.speedment.jpastreamer</groupId>
                <artifactId>jpastreamer-core</artifactId>

                <version>1.1.3</version>
        </dependency>
```

Eso hará que automáticamente nos instale todos las dependencias de JPA que el necesita que no son pocas precisamente.




Una vez tenemos configurado el proyecto a nivel de Maven para JPA con la librería de JPA Streamer es momento de crearnos una entidad sobre la cual podamos realizar consultas.

```
package com.arquitecturajava.jpa;

import java.util.Objects;

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



@Entity
@Table (name="personas")
public class Persona {

        @Id
```

```
private String nombre;
private String apellidos;
private int edad;
public String getNombre() {
        return nombre;
}
public void setNombre(String nombre) {
        this.nombre = nombre;
}
public String getApellidos() {
        return apellidos;
}
public void setApellidos(String apellidos) {
        this.apellidos = apellidos;
}
public int getEdad() {
        return edad;
}
public void setEdad(int edad) {
        this.edad = edad;
}
public Persona(String nombre, String apellidos, int edad) {
        super();
        this.nombre = nombre;
        this.apellidos = apellidos;
        this.edad = edad;
}
public Persona() {
        super();
}
@Override
```

```
	public int hashCode() {
		return Objects.hash(nombre);
	}
	@Override
	public boolean equals(Object obj) {
		if (this == obj)
			return true;
		if (obj == null)
			return false;
		if (getClass() != obj.getClass())
			return false;
		Persona other = (Persona) obj;
		return Objects.equals(nombre, other.nombre);
	}
}
```

# Usando JPA Streamer

Creada la Entidad usar JPA Streamer es muy sencillo ya que nos puede convertir una consulta que tiene cierta complejidad en JPA en una consulta con un Stream de datos directo.

```
package com.arquitecturajava.jpa;

import com.speedment.jpastreamer.application.JPAStreamer;

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

public class Principal6 {

	public static void main(String[] args) {
```

```
                EntityManagerFactory emf =
Persistence.createEntityManagerFactory("curso");
                JPAStreamer jpaStreamer = JPAStreamer.of(emf);
                jpaStreamer.stream(Persona.class)
            .forEach(System.out::println);
        }
}
```

En este caso hemos usado JPA Streamer para pasarle como parametro un EntityManagerFactory el cual se apoya en el persistence.xml.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1"
    xmlns="http://xmlns.jcp.org/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
    <persistence-unit name="curso" transaction-type="RESOURCE_LOCAL">
<provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
        <class>com.arquitecturajava.jpa.Persona</class>
        <properties>
                <property name= "hibernate.show_sql" value="true" />
                <!-- cuando nos conectamos desde JPA hay  q elegir el
tipo de motor -->
                        <property name="hibernate.dialect"
value="org.hibernate.dialect.MySQLDialect" />
            <property name="javax.persistence.jdbc.driver"
value="com.mysql.jdbc.Driver" />
            <property name="javax.persistence.jdbc.url"
value="jdbc:mysql://localhost/boot" />
            <property name="javax.persistence.jdbc.user" value="root"
/>
```

```
            <property name="javax.persistence.jdbc.password"
value="root" />
        </properties>
    </persistence-unit>
</persistence>
```

Con todo esto conseguido es momento de realizar consulta más basica y seleccionar todas las Personas de la tabla a través de un Stream de datos. SobreEscribimos el método toString de Persona y el resultado será :

```
Running under OpenJDK Runtime Environment 17.0.5+8
Hibernate: select persona0_.nombre as nombre1_0_, persona0_.apellidos
as apellido2_0_, persona0_.edad as edad3_0_ from personas persona0_
Persona [nombre=juan, apellidos=gomez, edad=20]
Persona [nombre=miguel, apellidos=sanchez, edad=30]
```

Como podemos observar se ha impreso el contenido de todas las facturas . De igual manera podemos usar las Entidades que JPAStreamer genera como metadatos al procesar el proyecto de Maven para gestionar filtrados o operativas semejantes con Streams que automaticamente son convertidas a consulstas dinámicas de JPA . Por ejemplo un sencillo filtrado.

```
package com.arquitecturajava.jpa;

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

import com.speedment.jpastreamer.application.JPAStreamer;
import com.arquitecturajava.jpa.Persona$;

public class Principal6 {
```

```
        public static void main(String[] args) {
                EntityManagerFactory emf =
Persistence.createEntityManagerFactory("curso");
                JPAStreamer jpaStreamer = JPAStreamer.of(emf);
jpaStreamer.stream(Persona.class).filter(Persona$.nombre.equal("juan")
)
            .forEach(System.out::println);
        }
}
```

El resultado le veremos pasar por la consola pero con la peculiaridad de que como el Stream se convierte en una consulta dinámica.

```
Running under OpenJDK Runtime Environment 17.0.5+8
Hibernate: select persona0_.nombre as nombre1_0_, persona0_.apellidos
as apellido2_0_, persona0_.edad as edad3_0_ from personas persona0_
where persona0_.nombre=?
Persona [nombre=juan, apellidos=gomez, edad=20]
```

Como podemos observar el uso JPA Streamer convierte un predicate en un filtro de JPA . Esta librería tiene mucho futuro y veremos como evoluciona para simplificar el manejo de Criteria API

## Otros artículos relacionados

- JPA y Jakarta EE (dependencias)
- JPA Persist y buenas prácticas
- JPA Single Table Inheritance
- JPA @OneToOne y relaciones 1 a 1
- JPA
- Spring Boot JPA y su configuración