CODING

JPAstreamer: Expressing Hibernate/JPA queries with Java streams

Mixing Hibernate/JPA and Java for database actions is neither completely type-safe nor intuitive. The JPAstreamer library offers another solution.

by Per Minborg

January 29, 2021

What if you could keep Hibernate/JPA and at the same time stick to Java and the Java Stream API?

Like many Java developers, I like to stick to a single language. Every time I come across a database project and need to switch from Java to Hibernate Query Language (HQL) or Java Persistence Query Language (JPQL), it is a tedious process because the syntax is wordy and partly unintuitive. Nevertheless, I do appreciate that the Hibernate Java Persistence API (JPA) abstracts away other inconveniences for me.

Now, you might think that it's already possible to obtain streams in the later versions of JPA, and this is perfectly true. However, those streams fall short of enabling the use of streams to express actual database queries. This feature merely relies on a wrapped stream from previous HQL or JPQL queries or users performing complete table scans. To add insult to injury, these stream implementations are often flawed and perform poorly.

To solve these problems, Speedment developed the open source library called JPAstreamer. This article shows how to express JPA, Hibernate, and Spring queries using standard Java streams. All that is necessary is to add a single dependency, and then the application is instantly ready to handle database Java streams.

Advantages

The Java Stream API is efficient and terse, and yet it's intuitive to express application logic with Java streams. JPAstreamer gives you all those advantages for database applications.

To illustrate code efficiency, **Listing 1** shows three examples of code. The first uses a JPA CriteriaBuilder, the second uses Spring Data JPA, and the third uses JPAstreamer code. The code snippets offer pagination of a table containing users (represented below by the class `User`, which is a standard JPA entity).

Listing 1. The same functionality in JPA CriteriaBuilder, Spring Data JPA, and JPAstreamer

JPA CriteriaBuilder:

```
void printPage(EntityManager entityManager, i

    final CriteriaBuilder criteriaBuilder = e
    final CriteriaQuery<User> criteriaQuery =

    final Root<User> root = criteriaQuery.fro
    criteriaQuery.select(root);

    final TypedQuery<User> typedQuery = entit
        .setFirstResult((page - 1) * pageSize
        .setMaxResults(pageSize);

    typedQuery.getResultList()
        .forEach(System.out::println);

}
```

Spring Data JPA:

```
interface UserRepository extends JpaRepositor
    @Query("select user from User user")
    Page<User> findAllPaged(Pageable pageable
}

...

@Autowired
private UserRepository userRepository;

void printPage(int page, int pageSize) {

   userRepository
       .findAllPaged(PageRequest.of((page - 1
       .forEach(System.out::println);
}
```

JPAstreamer:

```
void printPage(int page, int pageSize) {

    jpaStreamer.stream(User.class)
        .skip((page - 1) * pageSize)
        .limit(pageSize)
        .forEach(System.out::println);

}
```

The declarative style offered by the Stream API gives shorter code with improved readability and code metrics. Perhaps more importantly, JPAstreamer provides complete type-safety for all queries, which makes it possible to detect errors early, improve code quality, and save time.

JPAstreamer in a nutshell

Here's an example of a stream that operates on entities from the same `User` table (with attributes including a first and last name):

```
jpaStreamer.stream(User.class)
     .filter(User$.firstName.startsWith("A"))
     .sorted(User$.lastName.reversed())
     .limit(10)
     .forEach(System.out::println);
```

This will print the first 10 users where the first name starts with the letter "A" sorted in reverse order based on their last names. Omitting the details (to be covered shortly), this demonstrates how the desired result is easily described as a pipeline of stream operators.

On the surface, it may look as if the presented stream would require every row in the user table to be materialized into the JVM. However, this is not the case because the stream is actually optimized and rendered by JPAstreamer to JPQL queries similar to this:

```
select
    User
from
    User as User
where
    User.firstName like :param0
order by
    User.lastName desc
```

Thus, the stream queries are as performant as alternative approaches such as JPQL or CriteriaBuilder, but they have the improvement that JPAstreamer constitutes a streamlined and type-safe approach to expressing queries.

JPAstreamer under the hood

Similar to well-known Java libraries such as the JPA static metamodel and Project Lombok, JPAstreamer uses an annotation processor to form a metamodel at compile time. It inspects any classes marked with the standard JPA annotation `@Entity` (see **Listing 2**), and for every entity `Foo.class`, a corresponding `Foo$.class` is generated. The generated classes represent entity attributes as fields that can be used to form predicates.

In the `User` case above, classes can, for example, take the form `User$.firstName.startsWith("A")` that can be interpreted by JPAstreamer's query optimizer.

Listing 2. JPAstreamer generates fields based on existing JPA entities.

```
@Entity
@Table(name = "user", schema = "db-name")
public class User implements Serializable {

  @Id
  @GeneratedValue(strategy = GenerationType.I
  @Column(name = "user_id", nullable = false,
   private Integer userId;

  @Column(name = "first_name", columnDefiniti
private String firstName;

@Column(name = "last_name", columnDefinition
private String lastName;

// … shortened for brevity
```

JPAstreamer does not alter or disturb the existing codebase; it simply extends the API to handle Java stream queries from this point forward. Furthermore, the metamodel is, by default, placed in the `generated sources` subfolder located in the target folder and neither needs to be tested nor checked in with the other source code.

Getting started with JPAstreamer

Let's walk through the easy process of setting up JPAstreamer with your database application. To follow along, your application must use Java 8 (or later) as well as Hibernate or another JPA provider that is responsible for object persistence, such as EclipseLink, OpenJPA, or TopLink.

JPAstreamer is licensed under LGPL; thus it is easy to use in existing Hibernate projects, because Hibernate uses the same license.

Because only snippets of the code are shown here, you should view and run all the examples by downloading the source code from GitHub.

Installation. JPAstreamer is available in the Maven Central Repository, which boils down the installation processes to the simple addition of a single dependency in an existing Maven or Gradle build.

If you are using Maven, add the following lines to your existing Maven project:

```
<dependencies>
    <dependency>
        <groupId>com.speedment.jpastreamer</g
        <artifactId>jpastreamer-core</artifac
        <version>${jpa-streamer-version}</ver
    </dependency>
</dependencies>

<plugins>
    <!-- Needed by some IDEs to mark the gene
    <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>build-helper-maven-plugin
        <version>3.2.0</version>
        <executions>
            <execution>
                <phase>generate-sources</phas
                <goals>
                    <goal>add-source</goal>
                </goals>
                <configuration>
                    <sources>
                        <source>
${project.build.directory}/generated-sources/
     </source>
                    </sources>
                </configuration>
            </execution>
        </executions>
    </plugin>
</plugins>
```

For Gradle, add the following lines to your existing project:

```
repositories {
    mavenCentral()
}

dependencies {
    compile "com.speedment.jpastreamer:jpastr
    annotationProcessor "com.speedment.jpastr
}

/* Needed by some IDEs to mark the generated
sourceSets {
    main {
        java {
            srcDir 'src/main/java'
            srcDir 'target/generated-sources/
```

```
        }
    }
}
```

After the build file is altered, JPAstreamer requires a rebuild of the application to allow the generation of its metamodel. The boilerplates are generated as bytecode from the existing database configuration without user interaction. Interestingly, the Java source code for the bytecode is also available, which is useful for debugging your project.

At this point, the API is now extended, and you can start writing business logic.

Composing queries. With the setup done, I will move on to show how to start composing stream queries. The first step is to obtain an instance of JPAstreamer, for example:

```
JPAStreamer jpaStreamer = JPAStreamer.of("db-name");
```

The string `db-name` is to be replaced with the name of the persistence unit you wish to query. Look it up in your JPA configuration file (often named `persistence.xml`) under the tag `<persistence-unit>`.

Spring users have the option to use dependency injection with the `@Autowired` annotation, for example:

```
@Autowired
Private final JPAStreamer jpaStreamer;
```

The JPAstreamer instance provides access to the method `.stream()`, which accepts a class representing an entity you wish to stream. For example, as I have previously shown, the user table (containing `User` entities) can be streamed by simply typing the following:

```
jpaStreamer.stream(User.class)
```

This returns a stream of all the user rows of type `Stream<User>`. With a stream source at hand, you are free to add any Java stream operations to form a pipeline through which the data will flow (data flowing is a conceptual image rather than an actual description of how the code executes), for example:

```
List<String> users = jpaStreamer.stream(User.
      .filter(User$.age.greaterOrEqual(20))
      .map(u -> u.getFirstName() + " " + u.ge
      .collect(Collectors.toList());
```

This stream collects in a list the names of users who reached the age of 20. As a reminder, `$User` refers to the generated entity that is part of JPAstreamer's metamodel. This entity is used to form predicates and comparators for operations such as

`.filter()` and `.sort()`, which are quickly composed leveraging code completion in modern IDEs.

Here is another example that counts all users from Germany named “Otto” using a combined predicate:

```
long count = jpaStreamer.stream(User.class)
      .filter(User$.country.equal("Germany").
      .count();
```

Appealingly, not a single `User` entity will ever be pulled into the JVM for the stream above. Instead, the entire stream pipeline will be executed by the database in a single `select count(*) where ...` statement, after which the actual count will be directly returned to Java.

Always use predicates. By the way, to allow JPAstreamer to optimize a given stream, always use predicates derived from fields instead of anonymous lambdas. In other words, do this:

```
filter($User.age.greaterOrEqual(20))
```

Don’t do this:

```
filter(u -> u.getAge() >= 20)
```

Using JPAstreamer with Spring

To put the use of JPAstreamer in a broader context, I will show how smoothly it integrates with Spring Boot, allowing very simple creation of a REST API microservice. The demonstrated service utilizes the open source database Sakila that is available for download from Oracle or for use via Docker. This database represents a traditional movie rental store; hence, unsurprisingly it contains entities such as films, actors, and languages. A suitable objective is to create an endpoint that returns a list of films that are available for rent.

To achieve this using JPAstreamer and Spring, I first configured a standard Spring application for use with the Sakila database. Next, I added the JPAstreamer dependencies, and then I composed the JPA entities to represent the required tables. Because this service won’t expose all the available properties, I also built a view model to represent only the relevant information.

Because most of this is standard Hibernate, JPA, and Spring, I will highlight only the key code pieces. You can view and run the complete application on GitHub.

Maven configuration. For the Maven configuration, the POM file needs to be set up as a normal Spring project and include both the standard JPAstreamer dependency, as previously shown, and the following Spring extension:

```
<dependency>
<groupId>com.speedment.jpastreamer.integratio
    <artifactId>spring-boot-jpastreamer-autoc
    <version>${jpa-streamer-version}</version
     </dependency>
```

JPA entities and view model. To keep the microservice fairly simple, I added only a small number of entities to work with, namely film, actor, and language, which are all related. I also created a view model exposing the title, description, language, and starring actors of a film. Because this part of the code contains no JPAstreamer-specific parts, they are excluded from this article.

Creating the REST API. With this foundation in place, let's have a look at the REST controller. I will show the final implementation and break it down afterwards:

```
@RestController
public class FilmController {

    private final JPAStreamer jpaStreamer;

    @Autowired
    public FilmController(JPAStreamer jpaStre
        this.jpaStreamer = jpaStreamer;
    }

    @ResponseStatus(code = HttpStatus.OK)
    @GetMapping(value = "/films", produces =
    public Stream<FilmViewModel> films(
        @RequestParam(required = false, defau
        @RequestParam(required = false, defau
    ) {
        return jpaStreamer.stream(Film.class)
            .skip(page * pageSize)
            .limit(pageSize)
            .map(FilmViewModel::from);
    }
}
```

The first lines inject a JPAstreamer instance that can be used to fetch data objects. Next, there's a GET mapping that will list the entries of the film table. Whenever a user executes a GET request on `/films`, the `films()` method will be called. This method handles two optional arguments, `page` and `pageSize`, which you can use to specify the pagination rules. The response is served using JPAstreamer's Stream API to issue a database query.

The expression `.map(FilmViewModel::from)` uses the view model I initially mentioned to simply map each `Film` entry to a condensed format that strips away unwanted columns. This step could optionally be removed to return each `Film` in its entirety.

This demo application is configured to run on port 8080, meaning the endpoint http://localhost:8080/films will produce the following output with the first 10 films (showing only 1 film for brevity):

```
[
    {
        "title":"ACADEMY DINOSAUR",
        "description":"An Epic Drama of a Fem
        "language":"English",
        "actors":[
            "PENELOPE GUINESS",
            "CHRISTIAN GABLE",
            "LUCILLE TRACY",
            "SANDRA PECK",
            "JOHNNY CAGE",
            "MENA TEMPLE",
            "WARREN NOLTE",
            "OPRAH KILMER",
            "ROCK DUKAKIS",
            "MARY KEITEL"
        ]
    },
… 9 additional films ...
]
```

Further, the endpoint http://localhost:8080/films?page=9&pageSize=5 will produce an output with only five films and showing the 10th page. Now wasn't that easy?

Performance considerations

Generally, as database queries get more complex, and the database grows or is under high load, you run into performance issues that need to be mitigated. Below, I will show how JPAstreamer can help you avoid the infamous "N+1 select issue" that often arises when you initialize a lazy association between two entities. Typically, this will result in a staggering number of queries, which drastically decreases the application's performance.

With JPAstreamer, you can avoid this class of problem by providing a stream configuration object, as shown below:

```
StreamConfiguration<User> configuration = Str
     .joining(User$.city);

  jpaStreamer.stream(configuration)
```

This ensures that the resulting query joins cities with users, thus avoiding the N+1 select problem altogether. Any number of columns can be joined to the original entity.

Conclusion

Hibernate and JPA are powerful when it comes to database access, but Hibernate and JPA usage can easily lead to complexity. I have shown how you can integrate the open source library JPAstreamer with Hibernate (or any JPA provider) to compose type-safe and expressive database queries as standard Java streams. Doing this, you will be able to continue using JPA while keeping your codebase clean and maintainable.

As a final note, I would like to emphasize that you can continue to use old JPQL or CriteriaBuilder code in parallel with JPAstreamer. You can elect to convert old code now, later, or never.

Dig deeper

- JPAstreamer homepage
- GitHub repository
- Example source code
- Documentation
- Java Persistence API

Per Minborg

Per Minborg (@PMinborg) is CTO for Speedment Technologies and the lead contributor to JPAstreamer. He is an Oracle Groundbreaker Ambassador, an inventor, a developer, an Oracle Code One alumnus, and a co-author of the publication *Modern Java* with more than 20 years of Java coding experience. Minborg is a frequent contributor to open source projects.

Share this Page