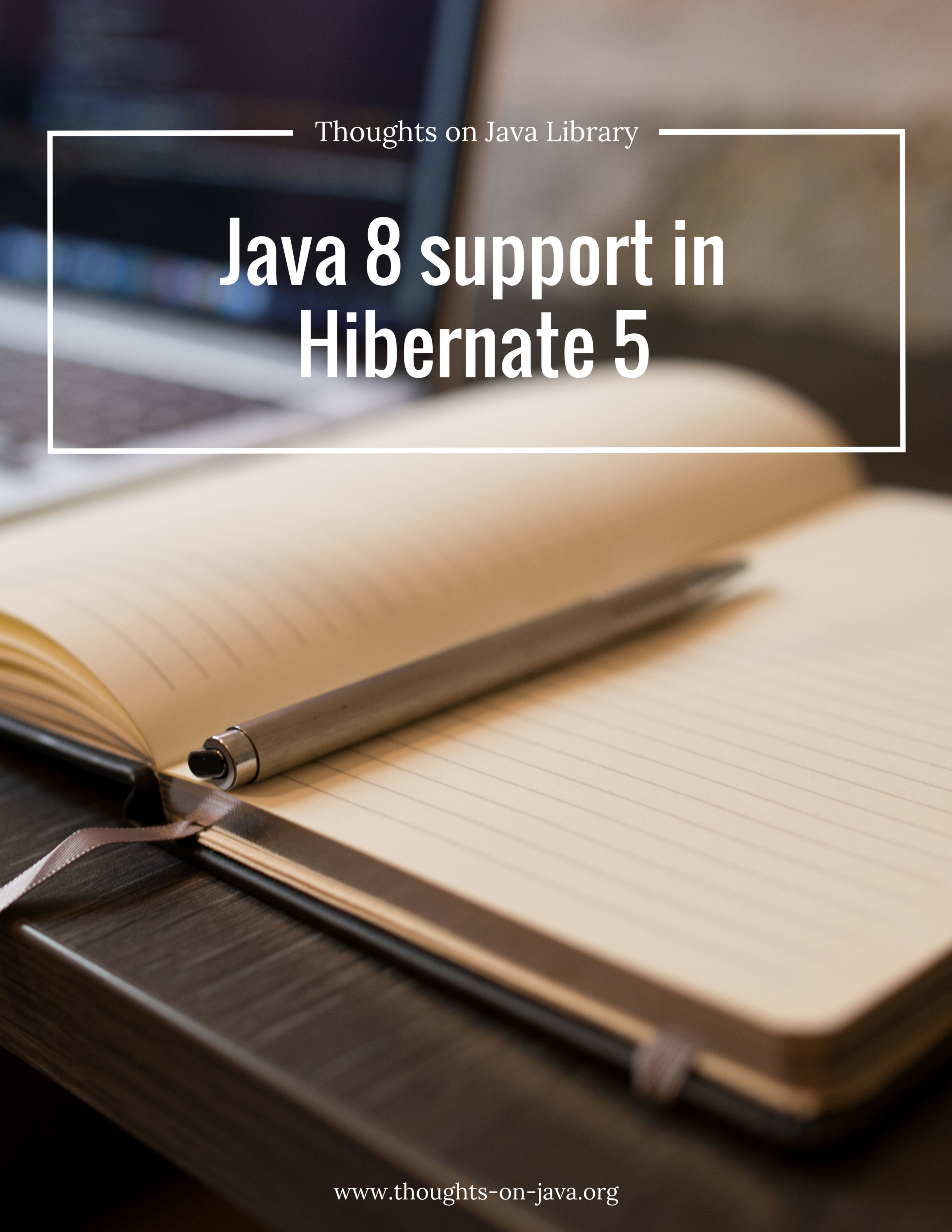


Thoughts on Java Library

Java 8 support in Hibernate 5

www.thoughts-on-java.org

Java 8 Support in Hibernate 5

Thoughts on Java Library

Thorben Janssen

Contents

Foreword 1

How to persist the new Date and Time API 2
- Java 8 support in Hibernate 5 2
- JDBC mappings 3
- Date and Time API classes as entity attributes 3
- Summary 5

How to use Java 8's Optional with Hibernate 6
- Optional attributes 6
- Load optional entities 8
- Summary 8

How to get query results as a *Stream* with Hibernate 5.2 9
- Advantages of the *stream()* method 9
- How to use the *stream()* method 10
- How to NOT use the *stream()* method 11
- Summary 11

Benefits of *@Repeatable* annotations in Hibernate 5.2 12
- What is *@Repeatable* and why should you like it? 12
- Which Hibernate annotations are repeatable? 13
- How to use *@Repeatable* annotations 13
- Summary 14

Foreword

Hi,

I'm Thorben, the author and founder of thoughts-on-java.org[1]. Thank you for downloading this ebook.

Java 8 was a huge release and brought a lot of improvements to the Java world which most developers don't want to miss anymore. Unfortunately, we have to wait for JPA 2.2 to get standardized support for the new features and APIs. But that doesn't mean that you can't use them.

The development team of Hibernate, the most popular JPA implementation, started to support different Java 8 features in version 5.0 and switched the code base of version 5.2 to Java 8. This allows them to not only support new data types, like the new date and time classes, as entity attributes but also to use Java 8 in their own APIs.

In the following chapters, I will show you:

- how to use classes of the Date and Time API as entity attributes and persist them with the right JDBC data types,
- how to use *Optional* as an entity attribute and as the return type when fetching entities from the database,
- how and why you should get large query results as a *Stream* and
- how to use Hibernate's repeatable annotations to clean up your entities.

Take care,

Thorben

[1] http://www.thoughts-on-java.org

How to persist the new Date and Time API

Do you use Java 8's Date and Time API in your projects? Let's be honest, working with *java.util.Date* is a pain, and I would like to replace it with the new API in all of my projects.

The only problem is that JPA 2.1 does not support it because it was earlier than Java 8. You can, of course, use *LocalDate* or other classes of the Date and Time API as entity attributes, but Hibernate stores them as blobs in the database.

You have two options to persist these classes correctly:

- You can implement a JPA *AttributeConverter* and convert the Java 8 class into one that is supported by Hibernate. I described this in detail in the blog post How to persist LocalDate and LocalDateTime with JPA[2]. This approach does not use any Hibernate-specific APIs and is portable to other JPA implementations. But it is also a little complicated.
- Or you can use the Hibernate-specific Java 8 support which was introduced with Hibernate 5. This approach is not portable to other JPA implementations but much easier to use. I will show you how to do that in this chapter.

Java 8 support in Hibernate 5

Java 8 is one of the big topics for Hibernate 5 and the support for the Date and Time API was one of the first changes.

The Hibernate versions 5.0 and 5.1 are still Java 7 compatible and ship the Java 8 support in a separate jar file called *hibernate-java8.jar*. You just have to add it to the classpath to use it in your application.

[2]http://www.thoughts-on-java.org/persist-localdate-localdatetime-jpa/

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-java8</artifactId>
    <version>5.1.0.Final</version>
</dependency>
```

Hibernate 5.2 is based on Java 8, and the *hibernate-java8.jar* file was removed. Hibernate now ships these classes as part of the *hibernate-core.jar*.

JDBC mappings

Hibernate maps the classes of the Date and Time API to the according JDBC types. The following list gives an overview of the supported classes and their JDBC mapping.

- *java.time.Duration* - BIGINT
- *java.time.Instant* - TIMESTAMP
- *java.time.LocalDateTime* - TIMESTAMP
- *java.time.LocalDate* - DATE
- *java.time.LocalTime* - TIME
- *java.time.OffsetDateTime* - TIMESTAMP
- *java.time.OffsetTime* - TIME
- *java.time.ZonedDateTime* - TIMESTAMP

Date and Time API classes as entity attributes

Hibernate supports the classes of the Date and Time API as BasicTypes. That provides the advantage that you don't have to provide any additional annotations. Not even the *@Temporal* annotation which you currently add to each *java.util.Date* attribute. Hibernate gets all required information from the type of the attribute. You can see an example of an entity with attributes of type *LocalDate*, *LocalDateTime*, and *Duration* in the following code snippet.

```
@Entity
public class MyEntity {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "id", updatable = false, nullable = false)
    private Long id;

    @Column
    private LocalDate date;

    @Column
    private LocalDateTime dateTime;

    @Column
    private Duration duration;

    ...
}
```

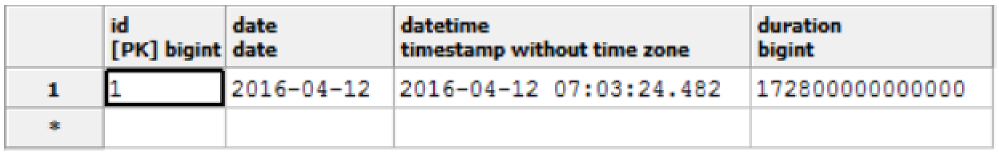
You can then use these attributes in the same way as any other attributes in your Java code.

```
EntityManager em = emf.createEntityManager();
em.getTransaction().begin();

MyEntity e = new MyEntity();
e.setDate(LocalDate.now());
e.setDateTime(LocalDateTime.now());
e.setDuration(Duration.ofDays(2));

em.persist(e);
```

And as you can see in the following screenshot, Hibernate persists them with the right JDBC data type instead of the *blob* it uses without Java 8 support.



	id [PK] bigint	date date	datetime timestamp without time zone	duration bigint
1	1	2016-04-12	2016-04-12 07:03:24.482	172800000000000
*				

Summary

We need to wait for JPA 2.2 to get standardized support for the Date and Time API. Until then you have to handle the type conversion yourself[3], or you can use the proprietary Java 8 support added in Hibernate 5.

Hibernate 5.0 and 5.1 ship the Java 8 support in an additional jar file (*hibernate-java8.jar*) which you need to add to your classpath. Hibernate 5.2 integrated the Java 8 support into the core module and removed the *hibernate-java8.jar* file.

Hibernate handles the classes of the Date and Time API as BasicTypes. That makes them even easier to use than the old *java.util.Date* because you don't have to add any additional annotations.

[3]http://www.thoughts-on-java.org/persist-localdate-localdatetime-jpa/

How to use Java 8’s Optional with Hibernate

Java 8 introduced *Optional<T>* as a container object which may contain null values. It’s often used to indicate to a caller that the object might be null and that it needs to be handled to avoid *NullPointerExceptions*.

Sounds pretty useful, right?

So why not use them in your persistence layer for optional entity attributes or when loading entities that may or may not exist?

Until the release of Hibernate 5.2, the reason was pretty simple: It wasn’t supported. And you still have to wait for JPA 2.2 if you don’t want to rely on proprietary features. But that’s a different topic.

The Hibernate team started to use Java 8 *Optional<T>* in their query APIs in Hibernate 5.2. You can use it to indicate optional attributes and query results which might not return a result.

Optional attributes

Using *Optional<T>* for optional entity attributes is probably the most obvious use case. But there is still no direct support for it in Hibernate 5.2. It requires a small trick which also works with older Hibernate versions.

Let’s say, you’re storing books in a database. Some of them are already published, and others are still in progress. In this case, you have a *Book* entity with an optional *publishingDate* attribute that might be *null*.

With previous Java versions, the *getPublishingDate()* method would just return *null*. The caller would need to know about it and handle it. With Java 8, you can return an *Optional* to indicate the potential *null* value and to avoid *NullPointerExceptions*.

But if you just change the type of the *publishingDate* attribute from *LocalDate* to *Optional<LocalDate>*, Hibernate isn’t able to determine the type of the attribute and throws a *MappingException*.

```
javax.persistence.PersistenceException: [PersistenceUnit: my-persistence-unit] U\
nable to build Hibernate SessionFactory
at org.hibernate.jpa.boot.internal.EntityManagerFactoryBuilderImpl.persistenceEx\
ception(EntityManagerFactoryBuilderImpl.java:951)
...
Caused by: org.hibernate.MappingException: Could not determine type for: java.ut\
il.Optional, at table: Book, for columns: [org.hibernate.mapping.Column(publishi\
ngDate)]
at org.hibernate.mapping.SimpleValue.getType(SimpleValue.java:454)
at org.hibernate.mapping.SimpleValue.isValid(SimpleValue.java:421)
at org.hibernate.mapping.Property.isValid(Property.java:226)
at org.hibernate.mapping.PersistentClass.validate(PersistentClass.java:595)
at org.hibernate.mapping.RootClass.validate(RootClass.java:265)
at org.hibernate.boot.internal.MetadataImpl.validate(MetadataImpl.java:329)
at org.hibernate.boot.internal.SessionFactoryBuilderImpl.build(SessionFactoryBui\
lderImpl.java:489)
at org.hibernate.jpa.boot.internal.EntityManagerFactoryBuilderImpl.build(EntityM\
anagerFactoryBuilderImpl.java:878)
... 28 more
```

To avoid this *Exception*, you have to use field-type access and keep *LocalDate* as the type of the *publishingDate* attribute. Hibernate is then able to determine the data type of the attribute but doesn't return an *Optional*.

And here is the trick: When you use field-type access, you can implement the getter and setter methods in your own way. You can, for example, implement a *getPublishingDate()* method which wraps the *publishingDate* attribute in an *Optional<LocalDate>*.

```
@Entity
public class Book {

    ...

    @Column
    private LocalDate publishingDate;

    ...

    public Optional getPublishingDate() {
        return Optional.ofNullable(publishingDate);
    }
    public void setPublishingDate(LocalDate publishingDate) {
        this.publishingDate = publishingDate;
    }
}
```

Load optional entities

Hibernate 5.2 also introduced the *loadOptional(Serializable id)* method to the *IdentifierLoadAccess* interface which returns an *Optional<T>*. You should use this method to indicate that the result might be empty when you can't be sure that the database contains a record with the provided id.

The *loadOptional(Serializable id)* method is similar to the *load(Serializable id)* method which you already know from older Hibernate versions. It returns the loaded entity or a null value if no entity with the given id exists. The new *loadOptional(Serializable id)* method wraps the entity in an *Optional<T>* and therefore indicates the potential null value.

As you can see in the following code snippet, you can use it in the same way as the existing *load(Serializable id)* method.

```
Session session = em.unwrap(Session.class);
Optional<Book> book = session.byId(Book.class).loadOptional(1L);

if (book.isPresent()) {
  log.info("Found book with id ["+book.get().getId()+"] and title ["+book.get().\
getTitle()+"].");
} else {
  log.info("Book doesn't exist.");
}
```

Summary

In this chapter, I showed you how to use *Optional* in Hibernate 5. The new, Hibernate-specific *loadOptional(Serializable id)* method fetches entities from the database and wraps them into an Optional. It is just a minor improvement compared to the existing *load(Serializable id)* method, but it helps to build cleaner APIs. Unfortunately, you still can't use it as an entity attribute type. But if you use field-type access, you can implement your own getter method which wraps the attribute in an *Optional*.

How to get query results as a *Stream* with Hibernate 5.2

Since version 5.2, Hibernate starts to use Java 8 classes in their proprietary APIs. This brought several changes to the existing APIs, like the small addition to the *Query* interface I want to show you in this chapter. The new *stream()* method allows you to process the query results as a Java 8 *Stream*.

But before we dive into the details, let me quickly explain the benefits of the new stream() method.

Advantages of the *stream()* method

In the beginning, it looks like a small improvement that makes your code a little less clunky. You already can take a *List* of query results and call its *stream()* method to get a *Stream* representation.

```
List<Book> books = session.createQuery("SELECT b FROM Book b", Book.class).list(\
);
books.stream()
    .map(b -> b.getTitle() + " was published on " + b.getPublishingDate())
    .forEach(m -> log.info(m));
```

Sure, this code also gives you a *Stream* with your query result, but it is not the most efficient approach. In this example, Hibernate will get all the selected *Book* entities from the database, store them in memory and put them into a *List*. You then call the *stream()* method and process the results one by one. That approach is OK as long as your result set isn't too big. But if you're working on a huge result set, you better scroll through the records and fetch them in smaller chunks.

You're already familiar with that approach if you've used JDBC result sets or Hibernate's *ScrollableResult*. Scrolling through the records of a result set and processing them as a *Stream* are a great fit. Both approaches process one record after the other, and there's no need to fetch all of them upfront. The Hibernate team, therefore, decided to reuse the existing *scroll()* method and the *ScrollableResult* to implement the new *stream()* method.

How to use the ***stream()*** method

As I wrote in the introduction, the change on the API level is quite small. It's just the *stream*() method on the *Query* interface. But sometimes that's all it takes to adapt an existing API so that you can use it in a modern way. The *stream*() method is part of the *Query* interface and you can, therefore, use it with all kinds of queries and projections.

Let's have a look at a few of them.

Entities

Entities are the most common projection with Hibernate, and you can use them in a *Stream* in any way you like. In the following example, I call the *stream*() method to get the result set as a *Stream*. I then use the *map*() method to create a log message for each *Book* entity and call the *forEach*() method to write each message to the log file. Your business logic will probably be a little more complex.

```
Stream<Book> books = session.createQuery("SELECT b FROM Book b", Book.class).str\
eam();
books.map(b -> b.getTitle() + " was published on " + b.getPublishingDate())
    .forEach(m -> log.info(m));
```

Scalar Values

Until now, scalar values were not a very popular projection because it returns a *List* of *Object*[]. You then have to implement a loop to go through all *Object*[]s and cast its elements to their specific types. That gets a lot easier with Streams. The following code snippet shows a simple native SQL query which returns two scalar values.

```
Stream<Object[]> books = session.createNativeQuery("SELECT b.title, b.publishing\
Date FROM book b").stream();
books.map(b -> new BookValue((String)b[0], (Date)b[1]))
    .map(b -> b.getTitle() + " was published on " + b.getPublishingDate())
    .forEach(m -> log.info(m));
```

If you still don't want to write this kind of code, you can still use an *@SqlResultMapping*[4].

[4]http://www.thoughts-on-java.org/2015/04/result-set-mapping-basics.html

POJOs

POJOs or similar projections can be easily created with a constructor expression, as you can see in the following code snippet. Unfortunately, there seems to be a bug (HHH-11029[5]) in Hibernate 5.2.2 so that these projections don't work with *Streams*. Instead of mapping the *BookValues* to *Strings* and writing them to the log file, the following code snippet throws a *ClassCastException*.

```
Stream<BookValue> books = session.createQuery("SELECT new org.thoughts.on.java.m\
odel.BookValue(b.title, b.publishingDate) FROM Book b", BookValue.class).stream(\
);
books.map(b -> b.getTitle() + " was published on " + b.getPublishingDate())
.forEach(m -> log.info(m));
```

How to NOT use the *stream()* method

The *stream*() method provides a comfortable and efficient way to get a *Stream* representation of your query result. I already talked about the advantages of this new method and how you can use it. But one important thing I haven't talked about is how to NOT use it. It is not directly related to the *stream*() method. It's related to a lot of *Stream* API examples I've seen since the release of Java 8.

The *Stream* API provides a set of methods that make it easy to filter elements, to check if they match certain criteria and to aggregate all items. In general, these methods are great but please don't use them to post-process your query results. As long as you can express these operations with SQL (believe me, you can implement almost all of them in SQL), the database can do them a lot better!

Summary

Hibernate 5.2 introduced the *stream*() method to the *Query* interface. It seems like just a small change, but it provides easy access to a *Stream* representation of the result set and allows you to use the existing APIs in a modern way.

As you've seen in the examples, the *stream*() method can be used with all kinds of queries and almost all projections. The only projection that's causing some problems in version 5.2.2 is the projection as a POJO. But I expect that it will be fixed soon.

[5] https://hibernate.atlassian.net/browse/HHH-11029

Benefits of *@Repeatable* annotations in Hibernate 5.2

Hibernate 5.2 introduced several changes based on Java 8 features. In the previous chapters, I showed you how to get query results as a Stream and the support of Date-Time API classes and *Optional*. In this chapter, I will have a look at an improvement to several Hibernate annotations. A lot of them are now *@Repeatable* which makes them much more comfortable to use.

What is *@Repeatable* and why should you like it?

Before I get to the Hibernate annotations, let's have a quick look at *@Repeatable*. The idea of repeatable annotations is very simple, and I've no idea why we had to wait so long for it.

Sometimes, you want to apply the same annotation multiple times. Before Java 8, the only way to do that was to use an additional annotation and provide an array of the annotations you want to apply as a value. The most common example with JPA and Hibernate are the *@NamedQueries* and *@NamedQuery* annotation.

```
@Entity
@NamedQueries({
    @NamedQuery(name = "Book.findByTitle", query = "SELECT b FROM Book b WHERE b\
.title = :title"),
    @NamedQuery(name = "Book.findByPublishingDate", query = "SELECT b FROM Book \
b WHERE b.publishingDate = :publishingDate")
})
public class Book {
    ...
}
```

I never liked this approach. The *@NamedQueries* annotation doesn't provide any benefit. The only reason it exists and why you have to add it to your entity is that you want to define multiple *@NamedQuery*. Java 8 finally provides a better solution. You can now declare an annotation as repeatable and apply it multiple times without any wrapper annotations.

Which Hibernate annotations are repeatable?

It’s pretty easy to find all Hibernate annotations that should be repeatable. You just need to have a look at the *org.hibernate.annotations* package and find all annotations that wrap an array of other annotations as their value. The wrapped annotations are obviously the ones that should be repeatable.

I had a look at that package, and it seems like all are now repeatable. You can find a list of all of them and their *JavaDoc* description below. And don’t be surprised about some annotation names. Hibernate provides its own version of a lot of JPA annotations, like *@NamedQuery*, to extend them with Hibernate-specific features.

- *AnyMetaDef* - Used to provide metadata about an Any or ManyToAny mapping.
- *ColumnTransformer* - Custom SQL expression used to read the value from and write a value to a column. Use for direct object loading/saving as well as queries. The write expression must contain exactly one “?” placeholder for the value. For example: read=”decrypt(credit_card_num)” write=”encrypt(?)”
- *FetchProfile* - Define the fetching strategy profile.
- *Filter* - Add filters to an entity or a target entity of a collection.
- *FilterDef* - Filter definition. Defines a name, default condition and parameter types (if any).
- *FilterJoinTable* - Add filters to a join table collection.
- *GenericGenerator* - Generator annotation describing any kind of Hibernate generator in a generic (de-typed) manner.
- *JoinColumnOrFormula* - Allows joins based on column or a formula. One of formula() or column() should be specified, but not both.
- *NamedNativeQuery* - Extends NamedNativeQuery with Hibernate features.
- *NamedQuery* - Extends NamedQuery with Hibernate features.
- *Table* - Complementary information to a table either primary or secondary.
- *Tuplizer* - Define a tuplizer for an entity or a component.
- *TypeDef* - A type definition. Much like Type, but here we can centralize the definition under a name and refer to that name elsewhere. The plural form is TypeDefs.

How to use *@Repeatable* annotations

With JPA and Hibernate versions before 5.2, you were not able to annotate an entity with multiple of the same annotations. If there was the need to do that, e.g. when you wanted to define multiple *@NamedQuery* for an entity, you had to wrap them in another annotation.

```
@Entity
@NamedQueries({
    @NamedQuery(name = "Book.findByTitle", query = "SELECT b FROM Book b WHERE b\
.title = :title"),
    @NamedQuery(name = "Book.findByPublishingDate", query = "SELECT b FROM Book \
b WHERE b.publishingDate = :publishingDate")
})
public class Book implements Serializable {
    ...
}
```

That's no longer required, if you use Hibernate's version of the @*NamedQuery* annotation or any other annotation listed in the previous section. As you can see below, you can now add multiple *org.hibernate.annotations.NamedQuery* annotations directly to the entity.

```
@Entity
@NamedQuery(name = "Hibernate5Book.findByTitle", query = "SELECT b FROM Hibernat\
e5Book b WHERE b.title = :title")
@NamedQuery(name = "Hibernate5Book.findByPublishingDate", query = "SELECT b FROM\
 Hibernate5Book b WHERE b.publishingDate = :publishingDate")
public class Hibernate5Book implements Serializable {
    ...
}
```

Summary

Making all these annotations repeatable is just a small change in the Hibernate code and in the beginning, it might not look like a big deal. But as a regular Hibernate or JPA user, you know that you currently use wrapper annotations at almost all of your entities.

This has become obsolete with Hibernate 5.2, and I like that a lot. The wrapper annotations provide no additional value and reduce the readability of the code.