Databases use the data types BLOB (binary large object) and CLOB (character large object) to store large objects, like images and very long texts. JPA and Hibernate provide two kinds of mappings for these types.

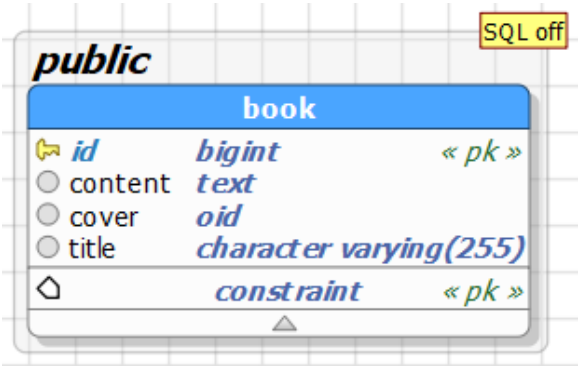
You can choose if you want to:

1. Materialize the LOB and map it to a byte[] or a String. This mapping is defined by the JPA specification and prioritizes ease of use over performance.
2. Use JDBC's LOB locators java.sql.Blob and java.sql.Clob. The LOB locators enable your JDBC driver to optimize for performance, e.g., by streaming the data. This mapping is Hibernate-specific.

The mapping of both approaches looks almost identical. You just need to annotate your entity attribute with a @Lob annotation. The only difference is the type of your entity attribute.

But you will see a huge difference in the way you use the values of these entity attributes.

Let's use both options to map the following Book table.



The columns cover of type oid and content of type text are the important ones for this article. We will map the cover column as BLOB and the content column as a CLOB.

Mapping a LOB to String or byte[]

The materialized mapping to a String or a byte[] is the most intuitive mapping for most Java developers. Entity attributes of these types

are easy to use, and it feels natural to use them in your domain model.

But Hibernate also needs to fetch all data stored in the LOB immediately and map it to a Java object. Depending on the size of your LOB, this can cause severe performance problems. If you, e.g., store large video files in your database, it's often better to use JDBC's LOB locators. I show you how to use them in the next section.

The JPA specification defines this mapping. You can not only use it with Hibernate but also with EclipseLink and OpenJPA.

Define the mapping

Creating materialized mappings is very simple. You just need an attribute of type String or byte[] and annotate it with JPA's @Lob annotation.

```
@Entity
public class Book {

    @Lob
    private String content;

    @Lob
    private byte[] cover;

    ...
}
```

Hibernate can also map nationalized character data types, like NCHAR, NVARCHAR, LONGNVARCHAR, and NCLOB. To define such a mapping, you need to annotate your entity attribute of type String with Hibernate's @Nationalized annotation instead of @Lob.

```
@Entity
public class Book {

    @Nationalized
    private String content;

    ...
}
```

Use the mapping

As I said at the beginning of this article, materialized mappings are straightforward to use. Hibernate fetches all data stored in the LOB when it initializes the entity and maps it to a String or byte[]. You can then use the entity attribute in the same way as any other attribute.

Here are 2 examples that store a new Book entity and fetch an existing Book entity from the database.

```
Book b = new Book();
b.setTitle("Hibernate Tips - More than 70 solutions to common Hibernate problems");
b.setCover(getCover());
b.setContent("This is a veeeery loooong text with almost all the content that you can find in the book ;)");
em.persist(b);
```

```
Book b2 = em.find(Book.class, b.getId());
byte[] cover = b2.getCover();
log.info(b2.getContent());
```

Mapping a LOB to java.sql.Clob or java.sql.Blob

With Hibernate, you can use the same approach to map your LOB to a java.sql.Clob or a java.sql.Blob. These Java types are not as easy to use as a String or byte[]. But they enable your JDBC driver to use LOB-specific optimizations, which might improve the performance of your application. If and what kind of optimizations are used, depends on the JDBC driver and your database.

The mapping is Hibernate-specific and not defined by the JPA specification.

Define the mapping

As you can see in the following code snippet, the mapping to JDBC's LOB locators java.sql.Clob and java.sql.Blob is almost identical to the previous example. The only 2 differences are:

1. The cover attribute is now of type Blob.
2. The content attribute is of type Clob.

```
@Entity
public class Book {

    @Lob
    private Clob content;

    @Lob
    private Blob cover;

    ...
}
```

And Hibernate also enables you to map the nationalized character data types NCHAR, NVARCHAR, LONGNVARCHAR, and NCLOB to a java.sql.Clob.

```
@Entity
public class Book {

    @Nationalized
    private Clob content;

    ...
}
```

Use the mapping

The types java.sql.Clob and java.sql.Blob provide more flexibility to the JDBC driver, but they are not as easy to use as a byte[] or a String.

You need to use Hibernate's BlobProxy and ClobProxy classes to create a Blob or Clob. As you can see in the code, that's a rather small inconvenience.

```
Book b = new Book();
b.setTitle("Hibernate Tips - More than 70 solutions to common Hibernate problems");
b.setCover(BlobProxy.generateProxy(getCover()));
b.setContent(ClobProxy.generateProxy("This is a veeeery loooong text with almost all the content that you can find in the book ;)"));
em.persist(b);
```

To create a Blob object, you can call the generateProxy method of the BlobProxy with a byte[] or an InputStream. And you can call the generateProxy method of the ClobProxy with a String or a Reader. That makes both proxies very comfortable to use.

Reading a Blob or a Clob is also not too complicated but requires a little more work than using a byte[] or a String. The java.sql.Blob interface provides you with multiple methods to get an InputStream or a byte[] of the BLOB value. And the java.sql.Clob interface defines various ways to get a Reader or a String of the CLOB value.

```
Book b2 = em.find(Book.class, b.getId());
Reader charStream = b2.getContent().getCharacterStream();
InputStream binaryStream = b2.getCover().getBinaryStream();
```

Lazy loading for LOBs

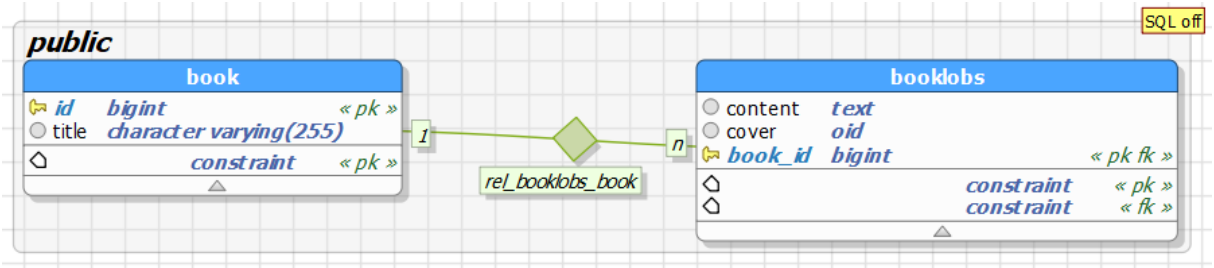
When we are talking about LOBs, we also need to talk about lazy loading. In most cases, LOBs require too much memory to fetch them eagerly every time you fetch the entity. It would be better only to fetch the LOB if you need it in your business code.

As I explained in a previous article, JPA defines lazy fetching for basic attributes as a hint. That means that your persistence provider can decide if it follows that hint or fetches the value eagerly.

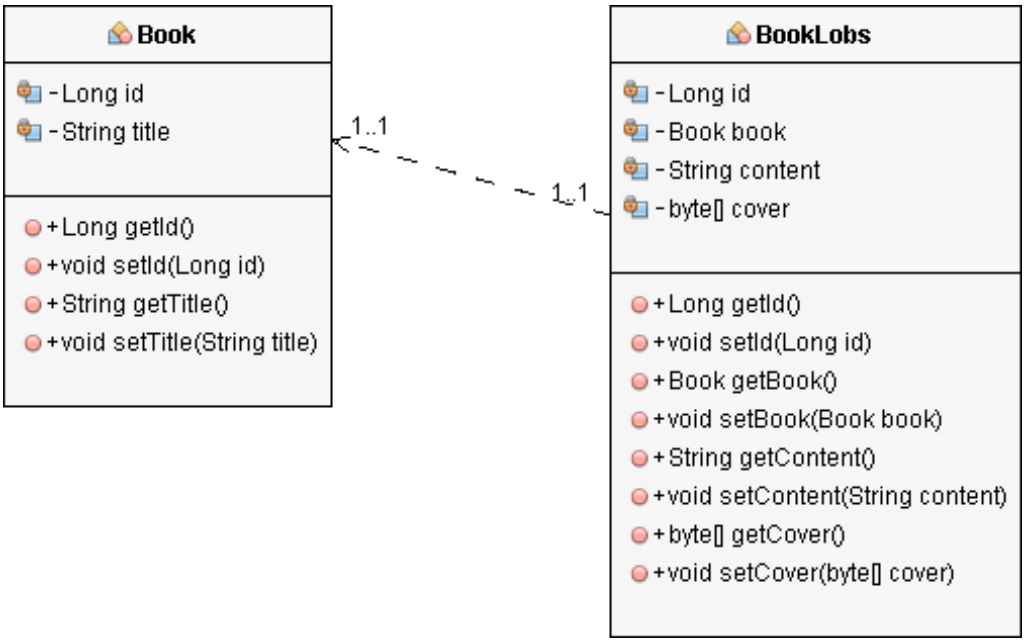
As a result, the support and implementation of this feature depend on your JPA implementation. Hibernate, for example, requires you to activate byte code enhancement. I explain that in more details in my Hibernate Performance Tuning Online Training.

In this article, I want to show and recommend a different approach. It doesn't require any provider-specific features and works with all JPA implementations.

The easiest and best way to load LOBs lazily is to store them in a separate table.



You can then map the LOBs to a separate entity.



That allows you to remove the LOBs from the Book entity and to model a unidirectional one-to-one association with a shared primary key on the BookLobs entity.

Define the mapping

The mapping of the 2 entities is pretty simple.

After removing the 2 LOBs, the Book entity is a simple entity with a generated primary key and a title attribute. As I will show you in the next section, you don't need to model the association to the BookLob entity.

```
@Entity
public class Book {
    @Id
    @GeneratedValue
    private Long id;

    private String title;

    ...
}
```

The BookLob entity models a unidirectional one-to-one association to the Book entity. The @MapsId annotation tells Hibernate to use the primary key value of the associated Book. I explained that in more details in Hibernate Tips: How to Share the Primary Key in a One-to-One Association.

And the materialized mappings of the content and cover attributes are the same as I used in the first example.

```
@Entity
public class BookLobs {

    @Id
    private Long id;

    @OneToOne
    @MapsId
    private Book book;

    @Lob
    private String content;

    @Lob
    private byte[] cover;

    ...
}
```

Use the mapping

The shared primary key and the unidirectional one-to-one associations make using the 2 entities very easy.

To store a new book in your database, you need to instantiate and persist a Book and BookLobs entity. The BookLobs entity uses the primary key value of the associated Book entity. So, you need to make

sure to initialize the association before you persist the BookLobs entity.

```
Book b = new Book();

b.setTitle("Hibernate Tips - More than 70 solutions to common Hibernate problems");

em.persist(b);


BookLobs bLob = new BookLobs();

bLob.setCover(getCover());

bLob.setContent("This is a veeeery loooong text with almost all the content that you can find in the book ;)");

bLob.setBook(b);

em.persist(bLob);
```

And when you want to get the BookLobs entity for a given Book entity, you just need to call the find method on your EntityManager with the id of the Book.

```
Book b2 = em.find(Book.class, b.getId());

BookLobs bLob2 = em.find(BookLobs.class, b2.getId());

byte[] cover = bLob2.getCover();

log.info(bLob2.getContent());
```