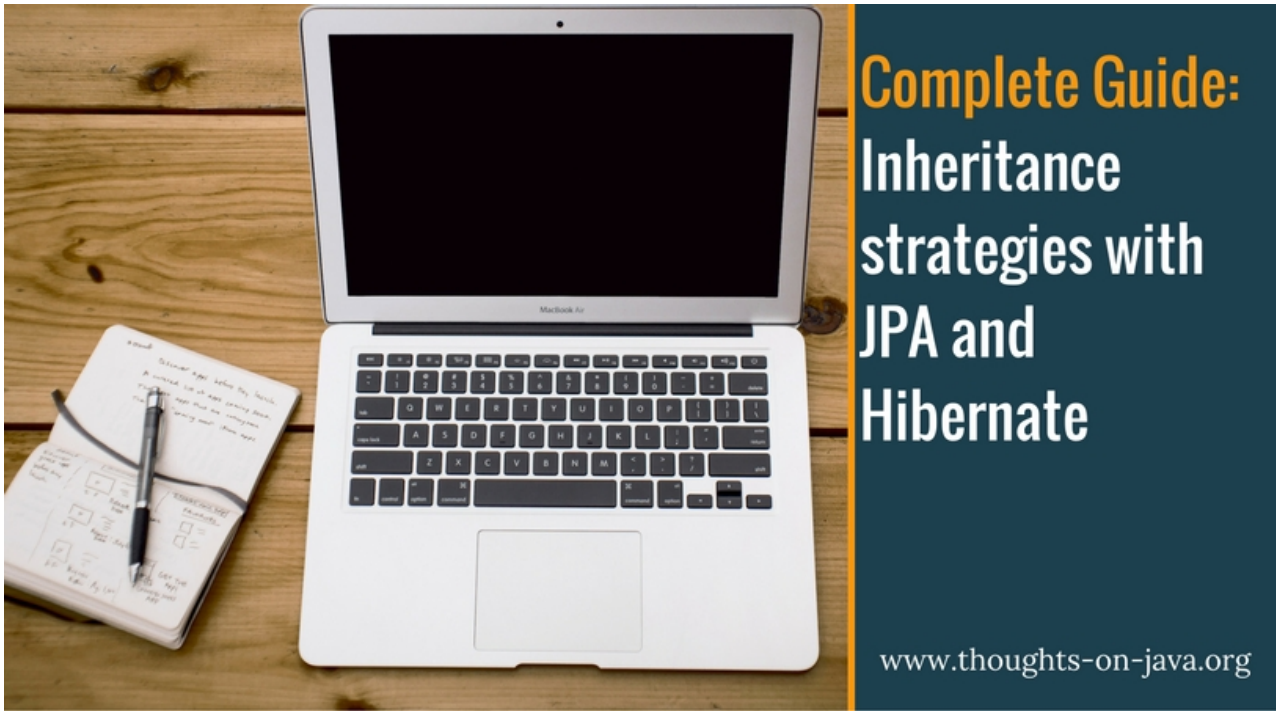
– The Complete Guide

Inheritance Strategies with JPA and Hibernate – The Complete Guide

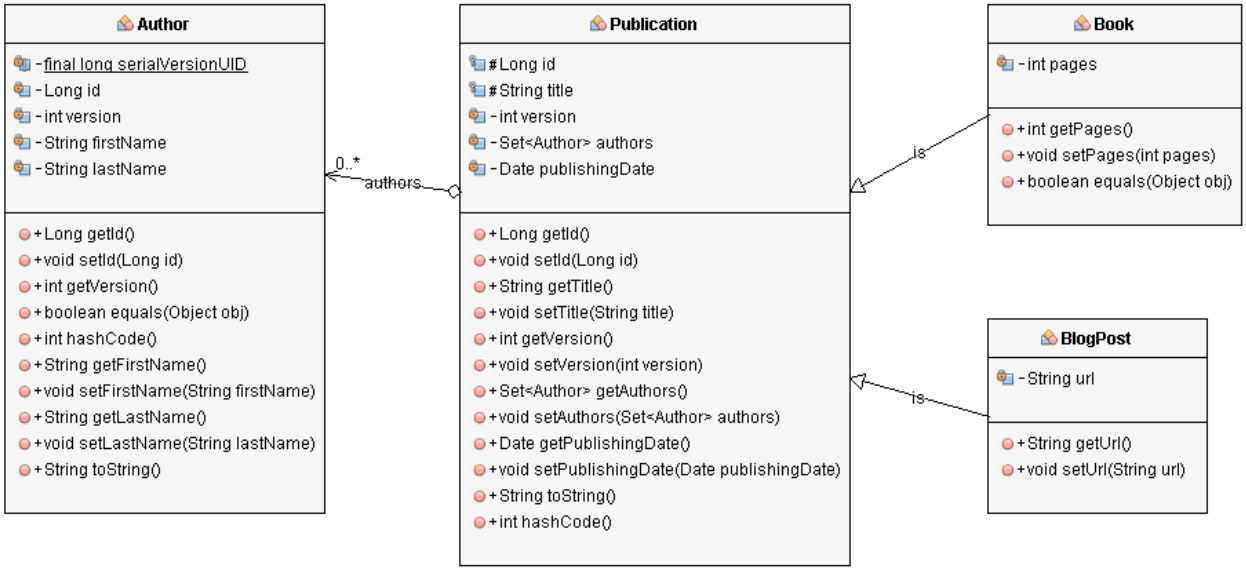
By Thorben Janssen — 31 Comments



Inheritance is one of the key concepts in Java, and it's used in most domain models. That often becomes an issue, if you try to map these models to a relational database. SQL doesn't support this kind of relationship and Hibernate, or any other JPA implementation has to map it to a supported concept.

Domain Model

I will use the same simple domain model in all of the examples to show you the different inheritance strategies. It consists of an author who has written different kinds of publications. A publication can either be a book or a blog post. Both of them share most of their attributes, like the id, a title, and a publishing date. In addition to the shared attributes, the book also stores the number of pages, and the blog post persists its URL.



4 Inheritance Strategies

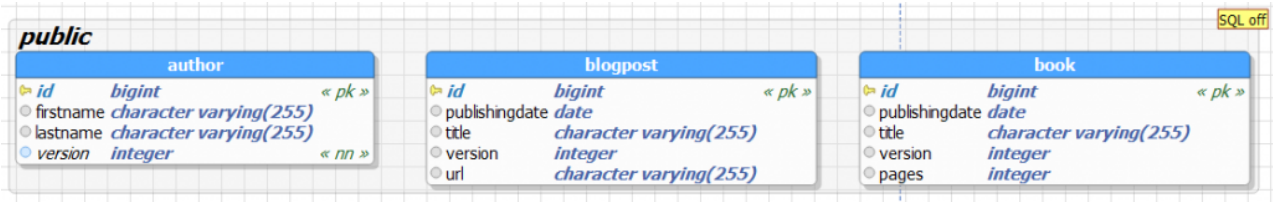
JPA and Hibernate support 4 inheritance strategies which map the domain objects to different table structures.

I've prepared a free cheat sheet for you with the most important information and code snippets you should remember from this post.

DOWNLOAD

Mapped Superclass

The mapped superclass strategy is the simplest approach to mapping an inheritance structure to database tables. It maps each concrete class to its own table.



That allows you to share the attribute definition between multiple entities. But it also has a huge drawback. A mapped superclass is not an entity, and there is no table for it.

That means that you can't use polymorphic queries that select all *Publication* entities and you also can't define a relationship between an *Author* entity and all *Publications*. You either need to use uni-directional relationship from the *Publication* to the *Author* entity, or you have to define a relationship between an *Author* and each kind of *Publication*. In general, if you need these relationships, you should have a look at the other inheritance strategies. They are most likely a better fit for your use case.

If you just want to share state and mapping information between your entities, the mapped superclass strategy is a good fit and easy to implement. You just have to set up your inheritance structure, annotate the mapping information for all attributes and add the *@MappedSuperclass* annotation to your superclass. Without the *@MappedSuperclass* annotation, Hibernate will ignore the mapping information of your superclass.

```java
@MappedSuperclass
public abstract class Publication {

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

        @Column
        protected String title;

        @Version
        @Column(name = "version")
        private int version;

        @Column
        @Temporal(TemporalType.DATE)
        private Date publishingDate;

        …
}
```

Publication_MappedSuperClass.java hosted with ❤ by **GitHub** **view raw**

The subclasses *Book* and *BlogPost* extend the *Publication* class and add their specific attributes with their mapping annotations. Both classes are also annotated with *@Entity* and will be managed by the persistence provider.

```java
@Entity(name = "Book")
public class Book extends Publication {

        @Column
        private int pages;

        …
}
```

Book_MappedSuperClass.java hosted with ❤ by **GitHub** **view raw**

```java
@Entity(name = "BlogPost")
public class BlogPost extends Publication {
```

BlogPost_MappedSuperClass.java hosted with ❤ by **GitHub** **view raw**

As I explained at the beginning of this section, you can't use the inheritance structure for polymorphic queries or to define relationships. But you can, of course, query the entites as any other entity.

```
1  List books = em.createQuery("SELECT b FROM Book b", Book.class).getResultList();
```

QueryBooks_MappedSuperClass.java hosted with ❤ by **GitHub** **view raw**

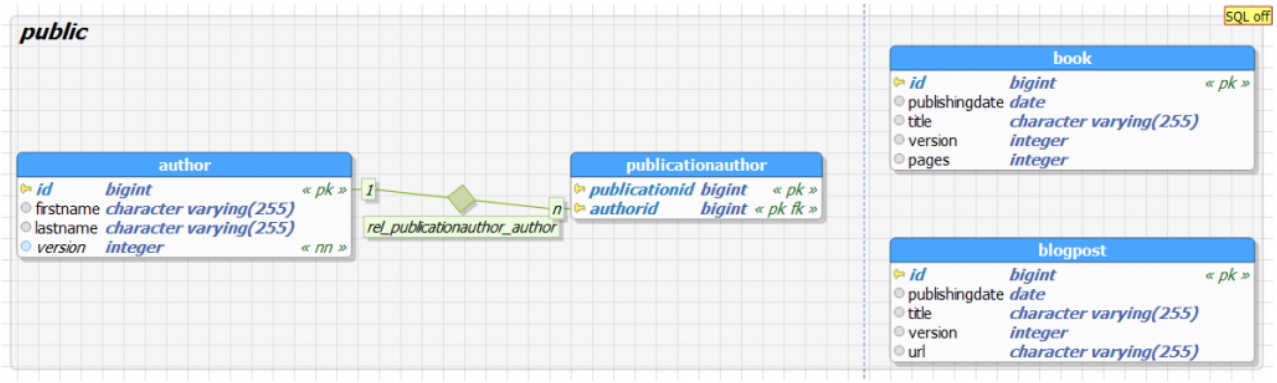
The Book entity and all its attributes are mapped to the book table. This makes the generated query simple and efficient. It just has to select all columns of the book table.

```
1  15:38:36,020 DEBUG [org.hibernate.SQL] - select book0_.id as id1_2_, book0_.publishingDate as publishi2_2_,
```

QueryBooks_MappedSuperClass.log hosted with ❤ by **GitHub** **view raw**

Table per Class

The table per class strategy is similar to the mapped superclass strategy. The main difference is that the superclass is now also an entity. Each of the concrete classes gets still mapped to its own database table. This mapping allows you to use polymorphic queries and to define relationships to the superclass. But the table structure adds a lot of complexity to polymorphic queries, and you should, therefore, avoid them.



The definition of the superclass with the table per class strategy looks similar to any other entity definition. You annotate the class with *@Entity* and add your mapping annotations to the attributes. The only difference is the additional *@Inheritance* annotation which you have to add

```
3  public abstract class Publication {
4
5          @Id
6          @GeneratedValue(strategy = GenerationType.AUTO)
7          @Column(name = “id”, updatable = false, nullable = false)
8          protected Long id;
9
10         @Column
11         protected String title;
12
13         @Version
14         @Column(name = “version”)
15         private int version;
16
17         @ManyToMany
18         @JoinTable(name = “PublicationAuthor”, joinColumns = { @JoinColumn(name = “publicationId”, referen
19         private Set authors = new HashSet();
20
21         @Column
22         @Temporal(TemporalType.DATE)
23         private Date publishingDate;
24
25         …
26 }
```

Publication_TablePerClass.java hosted with ❤ by **GitHub** **view raw**

The definitions of the *Book* and *BlogPost* entities are identical to the previously discussed mapped superclass strategy. You just have to extend the *Publication* class, add the *@Entity* annotation and add the class specific attributes with their mapping annotations.

```
1  @Entity(name = “Book”)
2  public class Book extends Publication {
3
4          @Column
5          private int pages;
6
7          …
8  }
```

Book_TablePerClass.java hosted with ❤ by **GitHub** **view raw**

entity attribute. That makes the query for a specific entity class easy and efficient.

```
1  List books = em.createQuery("SELECT b FROM Book b", Book.class).getResultList();
```

QueryBooks_TablePerClass.java hosted with ❤ by **GitHub** **view raw**

```
1  15:56:21,463 DEBUG [org.hibernate.SQL] - select book0_.id as id1_3_, book0_.publishingDate as publishi2_3_,
```

QueryBooks_TablePerClass.log hosted with ❤ by **GitHub** **view raw**

The superclass is now also an entity and you can, therefore, use it to define a relationship between the *Author* and the *Publication* entity. This allows you to call the *getPublications()* method to get all *Publications* written by that *Author*. Hibernate will map each *Publication* to its specific subclass.

```
1  List authors= em.createQuery("SELECT a FROM Author a", Author.class).getResultList();
2  for (Author a : authors) {
3          for (Publication p : a.getPublications()) {
4                  if (p instanceof Book)
5                  log(p.getTitle(), "book");
6                  else
7                  log(p.getTitle(), "blog post");
8          }
9  }
```

PublicationsOfAuthor_TablePerClass.java hosted with ❤ by **GitHub** **view raw**

The Java code looks easy and comfortable to use. But if you have a look at the generated SQL statement, you recognize that the table model makes the required query quite complicated.

```
1  15:57:16,722 DEBUG [org.hibernate.SQL] - select author0_.id as id1_0_, author0_.firstName as firstNam2_0_,
2  15:57:16,765 DEBUG [org.hibernate.SQL] - select publicatio0_.authorId as authorId2_4_0_, publicatio0_.publi
3  Effective Java is a book.
```

PublicationsOfAuthor_TablePerClass.log hosted with ❤ by **GitHub** **view raw**

Hibernate has to join the *author* table with the result of a subselect which uses a union to get all matching records from the *book* and *blogpost* tables. Depending on the amounts of records in both tables, this query might become a performance issue. And it gets even worse if you add more subclasses to the inheritance structure. You should, therefore, try to avoid these kinds of

Single Table

The single table strategy maps all entities of the inheritance structure to the same database table. This approach makes polymorphic queries very efficient and provides the best performance.

But it also has some drawbacks. The attributes of all entities are mapped to the same database table. Each record uses only a subset of the available columns and sets the rest of them to *null*. You can, therefore, not use *not null* constraints on any column that isn't mapped to all entities. That can create data integrity issues, and your database administrator might not be too happy about it.

When you persist all entities in the same table, Hibernate needs a way to determine the entity class each record represents. This is information is stored in a discriminator column which is not an entity attribute. You can either define the column name with a *@DiscriminatorColumn* annotation on the superclass or Hibernate will use *DTYPE* as its default name.

```
1  @Entity
2  @Inheritance(strategy = InheritanceType.SINGLE_TABLE)
3  @DiscriminatorColumn(name = “Publication_Type”)
4  public abstract class Publication {
5
6          @Id
7          @GeneratedValue(strategy = GenerationType.AUTO)
8          @Column(name = “id”, updatable = false, nullable = false)
9          protected Long id;
10
11         @Column
12         protected String title;
13
14         @Version
15         @Column(name = “version”)
16         private int version;
17
18         @ManyToMany
19         @JoinTable(name = “PublicationAuthor”, joinColumns = { @JoinColumn(name = “publicationId”, referen
```

```
26          …
27  }
```

Publication_SingleTable.java hosted with ❤ by **GitHub** **view raw**

The definition of the subclasses is again similar to the previous examples. But this time, you should also provide a *@DiscriminatorValue* annotation. It specifies the discriminator value for this specific entity class so that your persistence provider can map each database record to a concrete entity class.

The *@DiscriminatorValue* annotation is optional if you use Hibernate. If you don't provide a discriminator value, Hibernate will use the simple entity name by default. But this default handling isn't defined by the JPA specification, and you shouldn't rely on it.

```
1  @Entity(name = "Book")
2  @DiscriminatorValue("Book")
3  public class Book extends Publication {
4
5          @Column
6          private int pages;
7
8          …
9  }
```

Book_SingleTable.java hosted with ❤ by **GitHub** **view raw**

As I explained at the beginning of this section, the single table strategy allows easy and efficient data access. All attributes of each entity are stored in one table, and the query doesn't require any join statements. The only thing that Hibernate needs to add to the SQL query to fetch a particular entity class is a comparison of the discriminator value. In this example, it's a simple expression that checks that the column *publication_type* contains the value '*Book*'.

The previously discussed inheritance strategies had their issues with polymorphic queries. They were either not supported or required complex union and join operations. That's not the case if you use the single table strategy. All entities of the inheritance hierarchy are mapped to the same table and can be selected with a simple query. The following code and log snippets show an example for such a query. As you can see in the log messages, Hibernate selects all columns,

Joined

The joined table approach maps each class of the inheritance hierarchy to its own database table. This sounds similar to the table per class strategy. But this time, also the abstract superclass *Publication* gets mapped to a database table. This table contains columns for all shared entity attributes. The tables of the subclasses are much smaller than in the table per class strategy. They hold only the columns specific for the mapped entity class and a primary key with the same value as the record in the table of the superclass.

constraints on subclass attributes and to ensure data integrity. The definition of the superclass *Publication* is similar to the previous examples. The only difference is the value of the inheritance strategy which is *InheritanceType.JOINED*.

The definition of the subclasses doesn't require any additional annotations. They just extend the superclass, provide an *@Entity* annotation and define the mapping of their specific attributes.

As I already explained, the columns mapped by each subclass are stored in 2 different database tables. The *publication* table contains all columns mapped by the superclass *Publication* and the *book* table all columns mapped by the *Book* entity. Hibernate needs to join these 2 tables by their primary keys to select all attributes of the *Book* entity. This is an overhead that makes these queries slightly slower than the simpler queries generated for the single table strategy.

Hibernate has to use a similar approach for polymorphic queries. It has to left join the *publication* table with all tables of the subclasses, to get all *Publication*s of an *Author*.

Choosing a Strategy

Choosing the right inheritance strategy is not an easy task. As so often, you have to decide which advantages you need and which drawback you can accept for your application. Here are a few recommendations:

- If you require the best performance and need to use polymorphic queries and relationships, you should choose the single table strategy. But be aware, that you can't use not null constraints on subclass attributes which increase the risk of data inconsistencies.
- If data consistency is more important than performance and you need polymorphic queries and relationships, the joined strategy is probably your best option.
- If you don't need polymorphic queries or relationships, the table per class strategy is most likely the best fit. It allows you to use constraints to ensure data consistency and provides an option of polymorphic queries. But keep in mind, that polymorphic queries are very complex for this table structure and that you should avoid them.

I've prepared a free cheat sheet for you with the most important information and code snippets you should remember from this post.

DOWNLOAD

Related Posts:

1. **How to persist additional attributes for a relationship with JPA and Hibernate**
2. **6 Hibernate features that I'm missing in JPA**
3. **Mapping Definitions in JPA and Hibernate – Annotations, XML or both?**
4. **Composition vs. Inheritance with JPA and Hibernate**

Become a Thoughts on Java Supporter to claim your member perks and to help me write more articles like this.

Filed Under: Hibernate Advanced, JPA
Tagged With: Mapping

Improve Your Hibernate Skills At An In-Person Workshop

Comments

Simon Martinelli says

Hi Thorben,

Good post.

Reply

Thorben Janssen says

Good point.
Thanks!

Reply

Rafael Ponte says

Great article!

I think the generated SQL code is incorrect in Table per Class strategy. Where's the join?
https://gist.github.com/thjanssen/8b2d2ba33acc690e0a3ea0f4700d45f8#file-querybooks_tableperclass-log

One tip: if possible, try to show formated SQL so that it gets easier to undetstand it.

Reply

Binh Thanh Nguyen says

Thanks, nice explanation!

Reply

Reply

Thorben Janssen says

Thanks, Robert

Reply

Gjorgi says

Thank you for explaining every strategy

Reply

Thorben Janssen says

You're welcome 🙂
And thanks for your comment.

Reply

Ehsan Soleimani says

Very useful blog. thank you

Reply

Reply

Salah Alhaddabi says

Thanks a lot and a very nice comparison that helps a lot when it comes to decide on your strategy

Reply

sujamait@gmail.com says

Thanks easy to understand.

Reply

Thorben Janssen says

Thanks, happy to help

Reply

Samara says

Great Article , Cristal Clear !!

Reply

Reply

damaben says

excellent description, specialy the pros and cons (i have used entity framework strategies (TPH, TPC, TPT and never considered the null-constraint con of TPH (in jpa single table))

Reply

Thorben Janssen says

Yes, that's an often ignored side-effect of that strategy. It doesn't have to be a big issue if you implement and test your application carefully. But you should be aware of it ...

Reply

Sridhar says

Very nice comparison between different strategies, article is to the point and crystal clear, Thank you

Reply

Thorben Janssen says

Christopher Boyer says

Very great writeup! Wouldn't the single table strategy also have a drawback of space? A book is never going to have a url, so every book entity in the publication table is going to have wasted space. That problem will only get worse as more subclasses are added.

Reply

Thorben Janssen says

Empty fields don't take up space, but having lots of null values in your table might create performance problems on your database.

Reply

Ehsan says

Great job Thorben. Easy and understandable.
Thanks.

Reply

md7zn4 says

Where is the getPublications() defined? Author Class doesn't have this method.

In that example, the getPublications() method is defined on the *Author* class.

Reply

marco_s says

Very good article Thorben.

I have a question about the single table example:
Assume the Book and Blog Entities have a an additional relationship. E.g Book looks like:

@Entity(name = “Book”)
@DiscriminatorValue(“Book”)
public class Book extends Publication {

@Column
private int pages;
@ManyToOne
private BookPublisher
…
}

If I query the Publication entity via jpql/criteria: How can I join fetch the BookPublisher relationsship? I know how I can join fetch the relationship if I query the Book entity. But how to do it if I query the Publication entity and its subclasses?

Reply

marco_s says

Also noteable with eclipselink the entities in the the joined example require @DiscriminatorValue and @DiscriminatorColumn. This seems to be unspecified in jpa (v2.1).

cheers

Reply

Thorben Janssen says

Hi Marco,

Yes, that's right. All attributes that are shared by all subclasses go into the superclass. You can use these attribute in your queries even if you're operating on the superclass.
You can also cast an entity in your query to a specific subclass using the TREAT function.

Regards,
Thorben

Reply

Philipp Hundelshausen says

I think in the Single Table code the @Table annotation is missing.

Reply

Thorben Janssen says

Reply

Julien says

Thanks, very nice and clear explained!

Reply

Thorben Janssen says

Thanks, Julien!

Reply

Leave a Reply

Logged in as BerndOK@gmail.com. Log out?

Comment

LET'S CONNECT

Thorben Janssen
Independent consultant, trainer and author

SPEAKING AT

25th September 2019
JUG Saxony - Dresden (Germany):
Wer hat was wann geändert? Einfache Auditierung mit Envers (Talk - German)

26th September 2019
JUG Hessen - Kassel (Germany):
Hibernate Tips 'n' Tricks (Talk - German)

21st October 2019
Java Vienna - Vienna (Austria):
tbd (Talk - German)

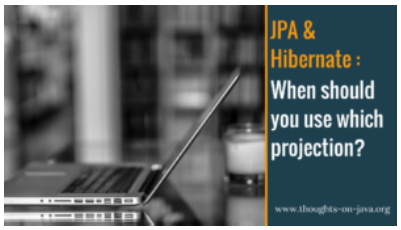
2nd-3rd December 2019
Düsseldorf (Germany):
Advanced Hibernate Workshop (2-day Workshop - English)

4th-6th December 2019
Düsseldorf (Germany):
Hibernate Performance Tuning Workshop (3-day Workshop - English)

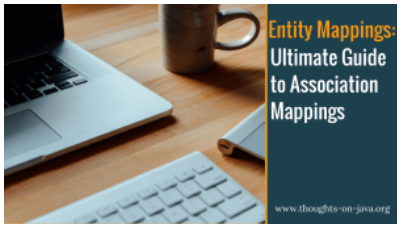
Looking for an on-site training?

14 YouTube Channels You Should Follow in 2019

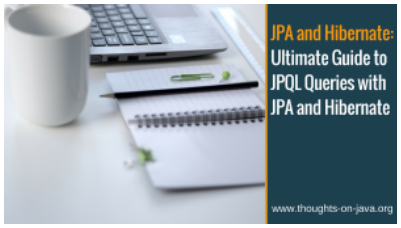
Getting Started With Hibernate



Entities or DTOs – When should you use which projection?



Ultimate Guide – Association Mappings with JPA and Hibernate



Ultimate Guide to JPQL Queries with JPA and Hibernate

Performance Applications

Hibernate Tip: Best Way To Work with Scalar Projections

Using the Optimal Query Approach and Projection for JPA and Hibernate

Implementing the Outbox Pattern with CDC using Debezium

Hibernate Tip: Difference between @JoinColumn and @PrimaryKeyJoinColumn

The Builder Pattern – How to use it with Hibernate

Hibernate Tip: Create an EntityGraph with multiple SubGraphs

Fluent Entities with Hibernate and JPA

Hibernate Tip: Join Unassociated Entities in Criteria Query

Enum Mappings with Hibernate – The Complete Guide