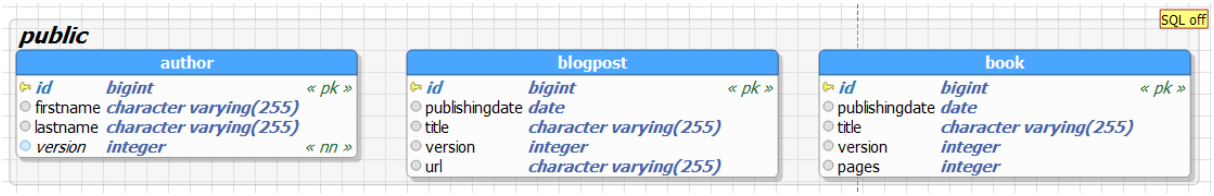
Mapped Superclass

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*.

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.

```
@MappedSuperclass
public abstract class Publication {…}
```

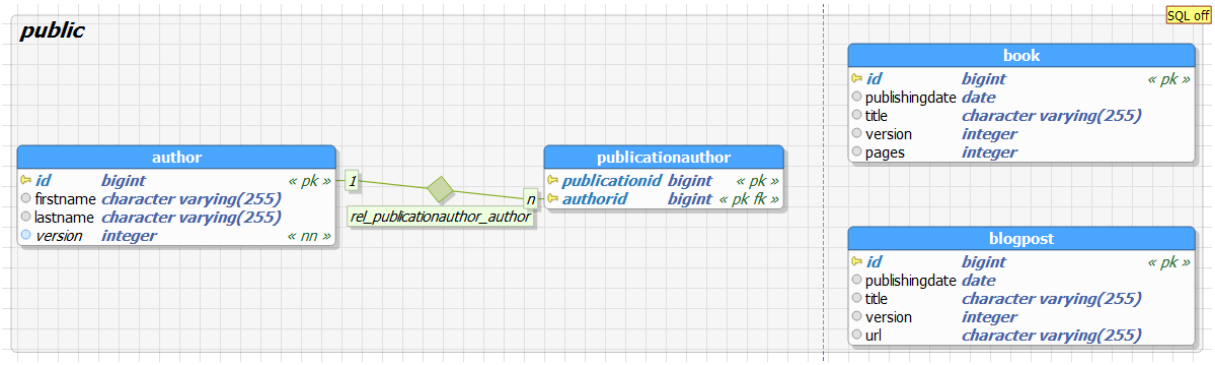
The subclasses Book and BlogPost extend the Publication class and add their specific attributes with their mapping annotations. Both classes are also annotated with @Entity and will be managed by the persistence provider.

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

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

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 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 to the class to define the inheritance strategy. In this case, it's the *InheritanceType.TABLE_PER_CLASS*.

```
@Entity
@Inheritance(strategy =
        InheritanceType.TABLE_PER_CLASS)
public abstract class Publication {…}
```

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.

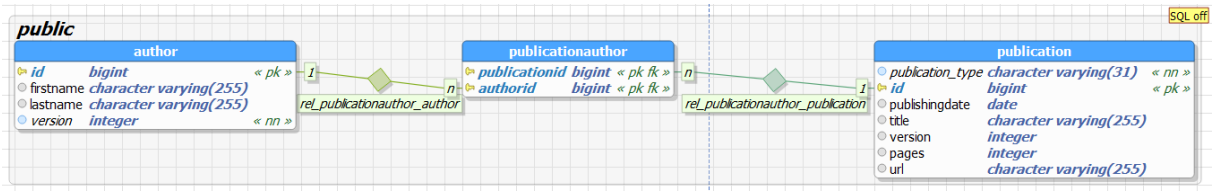
```
@Entity(name = “Book”)
public class Book extends Publication {…}
```

```
@Entity(name = “BlogPost”)
public class BlogPost extends Publication {…}
```

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.

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = “Publication_Type”)
public abstract class Publication {…}
```

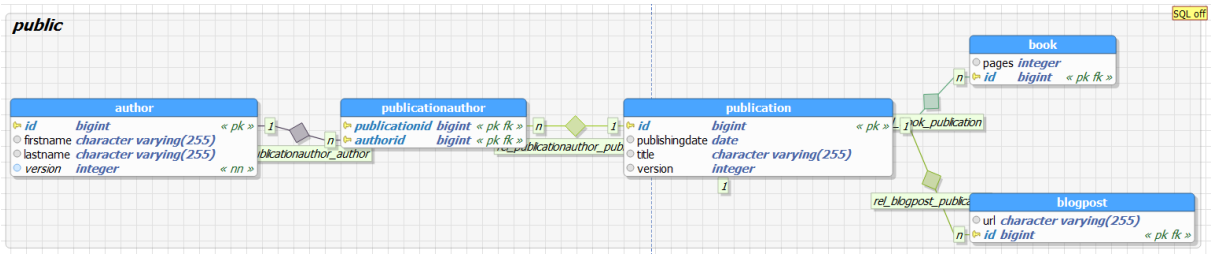
The definition of the subclasses is again similar to the previous examples. But this time, you should also provide a *@DiscriminatorValue* annotation. It specifies the discriminator value for this specific entity class so that your persistence provider can map each database record to a concrete entity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

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

```
@Entity(name = “BlogPost”)
@DiscriminatorValue(“Blog”)
public class BlogPost extends Publication {…}
```

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.



Each query of a subclass requires a join of the 2 tables to select the columns of all entity attributes. That increases the complexity of each query, but it also allows you to use *not null* 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*.

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class Publication {…}
```

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.

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

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

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.