Hibernate:

Inheritance Mapping:

To address this, the JPA specification provides several strategies:

* *MappedSuperclass* – the parent classes, can't be entities
* Single Table – The entities from different classes with a common ancestor are placed in a single table.
* Joined Table – Each class has its table, and querying a subclass entity requires joining the tables.
* Table per Class – All the properties of a class are in its table, so no join is required.

Each strategy results in a different database structure.

**Entity inheritance means that we can use polymorphic queries for retrieving all the subclass entities when querying for a superclass.**

Since Hibernate is a JPA implementation, it contains all of the above as well as a few Hibernate-specific features related to inheritance.

*Mapped Supper class:*

Using the *MappedSuperclass* strategy, inheritance is only evident in the class but not the entity model.

Let's start by creating a *Person* class that will represent a parent class:

@MappedSuperclass  
public class **Person** {

@Id  
 private **long** personId;  
 private String name;

// constructor, getters, setters  
}

**Notice that this class no longer has an *@Entity* annotation**, as it won't be persisted in the database by itself.

Next, let's add an *Employee* subclass:

@Entity  
public class **MyEmployee** extends **Person** {  
 private String company;  
 // constructor, getters, setters   
}

In the database, this will correspond to one *MyEmployee* table with three columns for the declared and inherited fields of the subclass.

If we're using this strategy, ancestors cannot contain associations with other entities.

Single Table:

**The Single Table strategy creates one table for each class hierarchy.** JPA also chooses this strategy by default if we don't specify one explicitly.

We can define the strategy we want to use by adding the *@Inheritance* annotation to the superclass:

@Entity  
@Inheritance(strategy = InheritanceType.SINGLE\_TABLE)  
public class **MyProduct** {  
 @Id  
 private **long** productId;  
 private String name;

// constructor, getters, setters  
}

The identifier of the entities is also defined in the superclass.

Then we can add the subclass entities:

@Entity  
public class **Book** extends **MyProduct** {  
 private String author;  
}

@Entity  
public class **Pen** extends **MyProduct** {  
 private String color;  
}

Joined table:

**Using this strategy, each class in the hierarchy is mapped to its table.** The only column that repeatedly appears in all the tables is the identifier, which will be used for joining them when needed.

Let's create a superclass that uses this strategy:

@Entity  
@Inheritance(strategy = InheritanceType.JOINED)  
public class **Animal** {  
 @Id  
 private **long** animalId;  
 private String species;

// constructor, getters, setters   
}

Then we can simply define a subclass:

@Entity  
public class **Pet** extends **Animal** {  
 private String name;

// constructor, getters, setters  
}

Both tables will have an *animalId* identifier column.

The primary key of the *Pet* entity also has a foreign key constraint to the primary key of its parent entity.

To customize this column, we can add the *@PrimaryKeyJoinColumn* annotation:

@Entity  
@PrimaryKeyJoinColumn(name = "petId")  
public class **Pet** extends **Animal** {  
 // ...  
}

**The disadvantage of this inheritance mapping method is that retrieving entities requires joins between tables**, which can result in lower performance for large numbers of records.

The number of joins is higher when querying the parent class because it will join with every single related child — so performance is more likely to be affected the higher up the hierarchy we want to retrieve records.

Table per Class:

**The Table per Class strategy maps each entity to its table, which contains all the properties of the entity, including the ones inherited.**

The resulting schema is similar to the one using @MappedSuperclass. But Table per Class will indeed define entities for parent classes, allowing associations and polymorphic queries as a result.

To use this strategy, we only need to add the *@Inheritance* annotation to the base class:

@Entity  
@Inheritance(strategy = InheritanceType.TABLE\_PER\_CLASS)  
public class **Vehicle** {  
 @Id  
 private **long** vehicleId;

private String manufacturer;

// standard constructor, getters, setters  
}

Then we can create the subclasses in the standard way.

This is not that different from merely mapping each entity without inheritance. The distinction is clear when querying the base class, which will return all the subclass records as well by using a *UNION* statement in the background.

**The use of *UNION* can also lead to inferior performance when choosing this strategy.** Another issue is that we can no longer use identity key generation.

Polymorphic Queries

As mentioned, querying a base class will retrieve all the subclass entities as well.

Let's see this behavior in action with a JUnit test:

@Test  
public void **givenSubclasses\_whenQuerySuperclass\_thenOk**() {  
 **Book** book = new **Book**(1, "1984", "George Orwell");  
 session.save(book);  
 **Pen** pen = new **Pen**(2, "my pen", "blue");  
 session.save(pen);

assertThat(session.createQuery("from MyProduct")  
 .getResultList()).hasSize(2);  
}

In this example, we've created two *Book* and *Pen* objects and then queried their superclass *MyProduct* to verify that we'll retrieve two objects.

Hibernate can also query interfaces or base classes that are not entities but are extended or implemented by entity classes.

Let's see a JUnit test using our *@MappedSuperclass* example:

@Test  
public void **givenSubclasses\_whenQueryMappedSuperclass\_thenOk**() {  
 **MyEmployee** emp = new **MyEmployee**(1, "john", "baeldung");  
 session.save(emp);

assertThat(session.createQuery(  
 "from com.baeldung.hibernate.pojo.inheritance.Person")  
 .getResultList())  
 .hasSize(1);  
}

Note that this also works for any superclass or interface, whether it's a *@MappedSuperclass* or not. The difference from a usual HQL query is that we have to use the fully qualified name since they are not Hibernate-managed entities.

If we don't want a subclass to be returned by this type of query, we only need to add the Hibernate *@Polymorphism* annotation to its definition, with type *EXPLICIT*:

@Entity  
@Polymorphism(type = PolymorphismType.EXPLICIT)  
public class **Bag** implements **Item** { ...}

ASSOCIATION mapping

Many to One

One to Many

Many to many

One to One

One to One:

A one-to-one association is similar to many-to-one association with a difference that the column will be set as unique. For example, an address object can be associated with a single employee object.

One to Many:

A One-to-Many mapping can be implemented using a Set java collection that does not contain any duplicate element. We already have seen how to map Set collection in hibernate, so if you already learned Set mapping then you are all set to go with one-to-many mapping.

A Set is mapped with a <set> element in the mapping table and initialized with java.util.HashSet. You can use Set collection in your class when there is no duplicate element required in the collection.

*Many to Many:*

A Many-to-Many mapping can be implemented using a Set java collection that does not contain any duplicate element. We already have seen how to map Set collection in hibernate, so if you already learned Set mapping, then you are all set to go with manyto-many mapping.

A Set is mapped with a <set> element in the mapping table and initialized with java.util.HashSet. You can use Set collection in your class when there is no duplicate element required in the collection.

Bootstrapping of Hibernate API:

Bootstrapping refers to the process of building and initializing a *SessionFactory.*

**To achieve this purpose, we need to have a *ServiceRegistry* that holds the *Services* needed by Hibernate. From this registry, we can build a *Metadata* object that represents the application's domain model and its mapping to the database**.

Let's explore these major objects in greater detail.

**3.1. *Service***

Before we dig into the *ServiceRegistry* concept, we first need to understand what a *Service*is*.* In Hibernate 5.0, a *Service* is a type of functionality represented by the interface with the same name:

org.hibernate.service.Service

By default, Hibernate provides implementations for the most common *Services*, and they are sufficient in most cases. Otherwise, we can build our own *Services* to either modify original Hibernate functionalities or add new ones.

In the next subsection, we'll show how Hibernate makes these *Services* available through a lightweight container called *ServiceRegistry.*

**3.2. *ServiceRegistry***

The first step in building a *SessionFactory* is to create a *ServiceRegistry.* This allows holding various *Services* that provide functionalities needed by Hibernate and is based on the [Java SPI](https://www.baeldung.com/java-spi) functionality.

Technically speaking, we can see the *ServiceRegistry* as a lightweight Dependency Injection tool where beans are only of type *Service.*

There are two types of *ServiceRegistry* and they are hierarchical*.* **The first is the *BootstrapServiceRegistry*, which has no parent and holds these three required services**:

* *ClassLoaderService:* allows Hibernate to interact with the *ClassLoader* of the various runtime environments
* *IntegratorService:* controls the discovery and management of the *Integrator* service allowing third-party applications to integrate with Hibernate
* *StrategySelector:* resolves implementations of various strategy contracts

**To build a *BootstrapServiceRegistry* implementation, we use the *BootstrapServiceRegistryBuilder*factory class,**which allows customizing these three services in a type-safe manner:

**BootstrapServiceRegistry** bootstrapServiceRegistry = new **BootstrapServiceRegistryBuilder**()  
 .applyClassLoader()  
 .applyIntegrator()  
 .applyStrategySelector()  
 .build();

**The second *ServiceRegistry* is the *StandardServiceRegistry*, which builds on the previous *BootstrapServiceRegistry*and holds the three *Services* mentioned above***.* Additionally, it contains various other *Services* needed by Hibernate, listed in the [*StandardServiceInitiators*](https://github.com/hibernate/hibernate-orm/blob/master/hibernate-core/src/main/java/org/hibernate/service/StandardServiceInitiators.java) class.

Like the previous registry, we use the *StandardServiceRegistryBuilder*to create an instance of the *StandardServiceRegistry:*

**StandardServiceRegistryBuilder** standardServiceRegistry =  
 new **StandardServiceRegistryBuilder**();

Under the hood, the *StandardServiceRegistryBuilder* creates and uses an instance of *BootstrapServiceRegistry.* We can also use an overloaded constructor to pass an already created instance:

**BootstrapServiceRegistry** bootstrapServiceRegistry =   
 new **BootstrapServiceRegistryBuilder**().build();  
**StandardServiceRegistryBuilder** standardServiceRegistryBuilder =   
 new **StandardServiceRegistryBuilder**(bootstrapServiceRegistry);

We use this builder to load a configuration from a resource file, such as the default *hibernate.cfg.xml*, and finally, we invoke the *build()* method to get an instance of the *StandardServiceRegistry.*

**StandardServiceRegistry** standardServiceRegistry = standardServiceRegistryBuilder  
 .configure()  
 .build();

**3.3. *Metadata***

Having configured all the *Services* needed by instantiating a *ServiceRegistry*either of type *BootstrapServiceRegistry or StandardServiceRegistry,***we now need to provide the representation of the application's domain model and its database mapping.**

The *MetadataSources* class is responsible for this:

**MetadataSources** metadataSources = new **MetadataSources**(standardServiceRegistry);  
metadataSources.addAnnotatedClass();  
metadataSources.addResource()

Next, we get an instance of *Metadata*, which we'll use in the last step:

**Metadata** metadata = metadataSources.buildMetadata();

**3.4. *SessionFactory***

The last step is to create the *SessionFactory* from the previously created *Metadata:*

**SessionFactory** sessionFactory = metadata.buildSessionFactory();

We can now open a *Session* and start persisting and reading entities:

**Session** session = sessionFactory.openSession();  
**Movie** movie = new **Movie**(100L);  
session.persist(movie);  
session.createQuery("FROM Movie").list();