Project Versions

stable

Search

Go

Table Of Contents

7. Inheritance Mapping
7.1. Mapped Superclasses
7.2. Single Table Inheritance
7.2.1. Design-time considerations
7.2.2. Performance impact
7.2.3. SQL Schema considerations
7.3. Class Table Inheritance
7.3.1. Design-time considerations
7.3.2. Performance impact
7.3.3. SQL Schema considerations
7.4. Overrides
7.4.1. Association Override

7. Inheritance Mapping

7.1. Mapped Superclasses

A mapped superclass is an abstract or concrete class that provides persistent entity state and mapping information for its subclasses, but which is not itself an entity. Typically, the purpose of such a mapped superclass is to define state and mapping information that is common to multiple entity classes.

Mapped superclasses, just as regular, non-mapped classes, can appear in the middle of an otherwise mapped inheritance hierarchy (through Single Table Inheritance or Class Table Inheritance).

> A mapped superclass cannot be an entity, it is not query-able and persistent relationships defined by a mapped superclass must be unidirectional (with an owning side only). This means that One-To-Many associations are not possible on a mapped superclass at all. Furthermore Many-To-Many associations are only possible if the mapped superclass is only used in exactly one entity at the moment. For further support of inheritance, the single or joined table inheritance features have to be used.

Example:

```
<?php
/** @MappedSuperclass */
class MappedSuperclassBase
{
    /** @Column(type="integer") */
    protected $mapped1;
    /** @Column(type="string") */
    protected $mapped2;
    /**
     * @OneToOne(targetEntity="MappedSuperclassRelated1")
     * @JoinColumn(name="related1_id", referencedColumnName="id")
     */
    protected $mappedRelated1;

    // ... more fields and methods
}

/** @Entity */
```

7.5. Query the Type

Previous topic

6. Association Mapping

Next topic

8. Working with Objects

This Page

Show Source

```
    /** @Id @Column(type="integer") */
    private $id;
    /** @Column(type="string") */
    private $name;

    // ... more fields and methods
}
```

The DDL for the corresponding database schema would look something like this (this is for SQLite):

```
CREATE TABLE EntitySubClass (mapped1 INTEGER NOT NULL, mapped2 TEXT NOT NULL, i
d INTEGER NOT NULL, name TEXT NOT NULL, related1_id INTEGER DEFAULT NULL, PRIMA
RY KEY(id))
```

As you can see from this DDL snippet, there is only a single table for the entity subclass. All the mappings from the mapped superclass were inherited to the subclass as if they had been defined on that class directly.

7.2. Single Table Inheritance

Single Table Inheritance is an inheritance mapping strategy where all classes of a hierarchy are mapped to a single database table. In order to distinguish which row represents which type in the hierarchy a so-called discriminator column is used.

Example:

PHP YAML

```
<?php
namespace MyProject\Model;

/**
 * @Entity
 * @InheritanceType("SINGLE_TABLE")
 * @DiscriminatorColumn(name="discr", type="string")
 * @DiscriminatorMap({"person" = "Person", "employee" = "Employee"})
 */
class Person
{
    // ...
}

/**
```

```
class Employee extends Person
{
    // ...
}
```

Things to note:

- The @InheritanceType, @DiscriminatorColumn and @DiscriminatorMap must be specified on the topmost class that is part of the mapped entity hierarchy.
- The @DiscriminatorMap specifies which values of the discriminator column identify a row as being of a certain type. In the case above a value of "person" identifies a row as being of type **`Person`** and "employee" identifies a row as being of type **`Employee`**.
- All entity classes that is part of the mapped entity hierarchy (including the topmost class) should be specified in the @DiscriminatorMap. In the case above Person class included.
- The names of the classes in the discriminator map do not need to be fully qualified if the classes are contained in the same namespace as the entity class on which the discriminator map is applied.
- If no discriminator map is provided, then the map is generated automatically. The automatically generated discriminator map contains the lowercase short name of each class as key.

7.2.1. Design-time considerations

This mapping approach works well when the type hierarchy is fairly simple and stable. Adding a new type to the hierarchy and adding fields to existing supertypes simply involves adding new columns to the table, though in large deployments this may have an adverse impact on the index and column layout inside the database.

7.2.2. Performance impact

This strategy is very efficient for querying across all types in the hierarchy or for specific types. No table joins are required, only a WHERE clause listing the type identifiers. In particular, relationships involving types that employ this mapping strategy are very performant.

There is a general performance consideration with Single Table Inheritance: If the target-entity of a many-to-one or one-to-one association is an STI entity, it is preferable for performance reasons that it be a leaf entity in the inheritance hierarchy, (ie. have no

7.2.3. SQL Schema considerations

For Single-Table-Inheritance to work in scenarios where you are using either a legacy database schema or a self-written database schema you have to make sure that all columns that are not in the root entity but in any of the different sub-entities has to allows null values. Columns that have NOT NULL constraints have to be on the root entity of the single-table inheritance hierarchy.

7.3. Class Table Inheritance

Class Table Inheritance is an inheritance mapping strategy where each class in a hierarchy is mapped to several tables: its own table and the tables of all parent classes. The table of a child class is linked to the table of a parent class through a foreign key constraint. Doctrine 2 implements this strategy through the use of a discriminator column in the topmost table of the hierarchy because this is the easiest way to achieve polymorphic queries with Class Table Inheritance.

Example:

```
<?php
namespace MyProject\Model;

/**
 * @Entity
 * @InheritanceType("JOINED")
 * @DiscriminatorColumn(name="discr", type="string")
 * @DiscriminatorMap({"person" = "Person", "employee" = "Employee"})
 */
class Person
{
    // ...
}

/** @Entity */
class Employee extends Person
{
    // ...
}
```

Things to note:

- The @InheritanceType, @DiscriminatorColumn and @DiscriminatorMap must be specified on

- The @DiscriminatorMap specifies which values of the discriminator column identify a row as being of which type. In the case above a value of "person" identifies a row as being of type **`Person`** and "employee" identifies a row as being of type **`Employee`**.
- The names of the classes in the discriminator map do not need to be fully qualified if the classes are contained in the same namespace as the entity class on which the discriminator map is applied.
- If no discriminator map is provided, then the map is generated automatically. The automatically generated discriminator map contains the lowercase short name of each class as key.

> When you do not use the SchemaTool to generate the required SQL you should know that deleting a class table inheritance makes use of the foreign key property **`ON DELETE CASCADE`** in all database implementations. A failure to implement this yourself will lead to dead rows in the database.

7.3.1. Design-time considerations

Introducing a new type to the hierarchy, at any level, simply involves interjecting a new table into the schema. Subtypes of that type will automatically join with that new type at runtime. Similarly, modifying any entity type in the hierarchy by adding, modifying or removing fields affects only the immediate table mapped to that type. This mapping strategy provides the greatest flexibility at design time, since changes to any type are always limited to that type's dedicated table.

7.3.2. Performance impact

This strategy inherently requires multiple JOIN operations to perform just about any query which can have a negative impact on performance, especially with large tables and/or large hierarchies. When partial objects are allowed, either globally or on the specific query, then querying for any type will not cause the tables of subtypes to be OUTER JOINed which can increase performance but the resulting partial objects will not fully load themselves on access of any subtype fields, so accessing fields of subtypes after such a query is not safe.

There is a general performance consideration with Class Table Inheritance: If the target-entity of a many-to-one or one-to-one association is a CTI entity, it is preferable for performance reasons that it be a leaf entity in the inheritance hierarchy, (ie. have no subclasses). Otherwise Doctrine **CANNOT** create proxy instances of this entity and will **ALWAYS** load the entity eagerly.

7.3.3. SQL Schema considerations

For each entity in the Class-Table Inheritance hierarchy all the mapped fields have to be columns on the table of this entity. Additionally each child table has to have an id column that matches the id column definition on the root table (except for any sequence or auto-increment details). Furthermore each child table has to have a foreign key pointing from the id column to the root table id column and cascading on delete.

7.4. Overrides

Used to override a mapping for an entity field or relationship. May be applied to an entity that extends a mapped superclass to override a relationship or field mapping defined by the mapped superclass.

7.4.1. Association Override

Override a mapping for an entity relationship.

Could be used by an entity that extends a mapped superclass to override a relationship mapping defined by the mapped superclass.

Example:

PHP YAML

```php
<?php
// user mapping
namespace MyProject\Model;
/**
 * @MappedSuperclass
 */
class User
{
    //other fields mapping

    /**
     * @ManyToMany(targetEntity="Group", inversedBy="users")
     * @JoinTable(name="users_groups",
     *  joinColumns={@JoinColumn(name="user_id", referencedColumnName="id")},
     *  inverseJoinColumns={@JoinColumn(name="group_id", referencedColumnName="id")}
     * )
     */
    protected $groups;

    /**
```

```
     */
    protected $address;
}

// admin mapping
namespace MyProject\Model;
/**
 * @Entity
 * @AssociationOverrides({
 *      @AssociationOverride(name="groups",
 *          joinTable=@JoinTable(
 *              name="users_admingroups",
 *              joinColumns=@JoinColumn(name="adminuser_id"),
 *              inverseJoinColumns=@JoinColumn(name="admingroup_id")
 *          )
 *      ),
 *      @AssociationOverride(name="address",
 *          joinColumns=@JoinColumn(
 *              name="adminaddress_id", referencedColumnName="id"
 *          )
 *      )
 * })
 */
class Admin extends User
{
}
```

Things to note:

- The “association override” specifies the overrides base on the property name.
- This feature is available for all kind of associations. (OneToOne, OneToMany, ManyToOne, ManyToMany)
- The association type **CANNOT** be changed.
- The override could redefine the joinTables or joinColumns depending on the association type.

7.4.2. Attribute Override

Override the mapping of a field.

Could be used by an entity that extends a mapped superclass to override a field mapping defined by the mapped superclass.

```php
<?php
// user mapping
namespace MyProject\Model;
/**
 * @MappedSuperclass
 */
class User
{
    /** @Id @GeneratedValue @Column(type="integer", name="user_id", length=150) */
    protected $id;

    /** @Column(name="user_name", nullable=true, unique=false, length=250) */
    protected $name;

    // other fields mapping
}

// guest mapping
namespace MyProject\Model;
/**
 * @Entity
 * @AttributeOverrides({
 *      @AttributeOverride(name="id",
 *          column=@Column(
 *              name     = "guest_id",
 *              type     = "integer",
 *              length   = 140
 *          )
 *      ),
 *      @AttributeOverride(name="name",
 *          column=@Column(
 *              name     = "guest_name",
 *              nullable = false,
 *              unique   = true,
 *              length   = 240
 *          )
 *      )
 * })
 */
class Guest extends User
{
}
```

Things to note:

- The “attribute override” specifies the overrides base on the property name.

a **MappingException**

- The override can redefine all the columns except the type.

7.5. Query the Type

It may happen that the entities of a special type should be queried. Because there is no direct access to the discriminator column, Doctrine provides the **INSTANCE OF** construct.

The following example shows how to use **INSTANCE OF**. There is a three level hierarchy with a base entity **NaturalPerson** which is extended by **Staff** which in turn is extended by **Technician**.

Querying for the staffs without getting any technicians can be achieved by this DQL:

```
<?php
$query = $em->createQuery("SELECT staff FROM MyProject\Model\Staff staff WHERE
staff INSTANCE OF MyProject\Model\Staff");
$staffs = $query->getResult();
```