

## Project Versions

latest ▼

## Search

Go

## Table Of Contents

### 20. XML Mapping

#### 20.1. Simplified XML Driver

##### 20.1.1. Example

##### 20.1.2. XML-Element Reference

#### 20.2. Defining an Entity

#### 20.3. Defining Fields

#### 20.4. Defining Identity and Generator Strategies

#### 20.5. Defining a Mapped Superclass

#### 20.6. Defining Inheritance Mappings

#### 20.7. Defining Lifecycle Callbacks

#### 20.8. Defining One-To-One

## 20. XML Mapping

The XML mapping driver enables you to provide the ORM metadata in form of XML documents.

The XML driver is backed by an XML Schema document that describes the structure of a mapping document. The most recent version of the XML Schema document is available online at <http://www.doctrine-project.org/schemas/orm/doctrine-mapping.xsd>. In order to point to the latest version of the document of a particular stable release branch, just append the release number, i.e.: doctrine-mapping-2.0.xsd The most convenient way to work with XML mapping files is to use an IDE/editor that can provide code-completion based on such an XML Schema document. The following is an outline of a XML mapping document with the proper xmlns/xsi setup for the latest code in trunk.

```
<doctrine-mapping xmlns="http://doctrine-project.org/schemas/orm/doctrine-mapping"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://doctrine-project.org/schemas/orm/doctrine-mapping.xsd"
  https://raw.githubusercontent.com/doctrine/doctrine2/master/doctrine-mapping.xsd">
  ...
</doctrine-mapping>
```

The XML mapping document of a class is loaded on-demand the first time it is requested and subsequently stored in the metadata cache. In order to work, this requires certain conventions:

- Each entity/mapped superclass must get its own dedicated XML mapping document.
- The name of the mapping document must consist of the fully qualified name of the class where namespace separators are replaced by dots (.). For example an Entity with the fully qualified class-name "MyProject" would require a mapping file "MyProject.Entities.User.dcm.xml" unless the extension is changed.

- All mapping documents should get the extension ".dcm.xml" to identify it as a Doctrine

## 20.9. Defining Many-To-One Associations

## 20.10. Defining One-To-Many Associations

## 20.11. Defining Many-To-Many Associations

## 20.12. Cascade Element

## 20.13. Join Column Element

## 20.14. Defining Order of To-Many Associations

## 20.15. Defining Indexes or Unique Constraints

## 20.16. Derived Entities ID syntax

[Previous topic](#)

[19. Partial Objects](#)

[Next topic](#)

[21. YAML Mapping](#)

[This Page](#)

[Show Source](#)

the file extension easily enough.

```
<?php
$driver->setFileExtension('.xml');
```

It is recommended to put all XML mapping documents in a single folder but you can spread the documents over several folders if you want to. In order to tell the XmlDriver where to look for your mapping documents, supply an array of paths as the first argument of the constructor, like this:

```
<?php
$config = new \Doctrine\ORM\Configuration();
$driver = new \Doctrine\ORM\Mapping\Driver\XmlDriver(array('/path/to/files1'
/path/to/files2'));
$config->setMetadataDriverImpl($driver);
```

Note that Doctrine ORM does not modify any settings for `libxml`, therefore, external XML entities may or may not be enabled or configured correctly. XML mappings are not XXE/XEE attack vectors since they are not related with user input, but it is recommended that you do not use external XML entities in your mapping file to avoid running into unexpected behaviour.

## 20.1. Simplified XML Driver

The Symfony project sponsored a driver that simplifies usage of the XML Driver. The changes between the original driver are:

File Extension is `.orm.xml`

Filenames are shortened, "MyProjectEntitiesUser" will become `User.orm.xml`

You can add a global file and add multiple entities in this file.

Configuration of this client works a little bit different:

```
<?php
$namespaces = array(
    'MyProject\Entities' => '/path/to/files1',
```

```
$driver = new \Doctrine\ORM\Mapping\Driver\SimplifiedXmlDriver($namespaces);
$driver->setGlobalBasename('global'); // global.orm.xml
```

### 20.1.1. Example

As a quick start, here is a small example document that makes use of several common elements:

```
// Doctrine.Tests\ORM\Mapping\User.dcm.xml
<?xml version="1.0" encoding="UTF-8"?>
<doctrine-mapping xmlns="http://doctrine-project.org/schemas/orm/doctrine-ma
ng"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://doctrine-project.org/schemas/orm/doctrine-ma
ing
                                http://raw.github.com/doctrine/doctrine2/master/do
ine-mapping.xsd">

    <entity name="Doctrine\Tests\ORM\Mapping\User" table="cms_users">

        <indexes>
            <index name="name_idx" columns="name"/>
            <index columns="user_email"/>
        </indexes>

        <unique-constraints>
            <unique-constraint columns="name,user_email" name="search_idx" /
        </unique-constraints>

        <lifecycle-callbacks>
            <lifecycle-callback type="prePersist" method="doStuffOnPrePersist" />
            <lifecycle-callback type="prePersist" method="doOtherStuffOnPrePersist" />
            <lifecycle-callback type="postPersist" method="doStuffOnPostPersist" />
        </lifecycle-callbacks>

        <id name="id" type="integer" column="id">
```

```

<sequence-generator sequence-name="tablename_seq" allocation-size="100" initial-value="1" />
</id>

<field name="name" column="name" type="string" length="50" nullable="true" unique="true" />
<field name="email" column="user_email" type="string" column-definition="CHAR(32) NOT NULL" />

<one-to-one field="address" target-entity="Address" inversed-by="user"
  <cascade><cascade-remove /></cascade>
  <join-column name="address_id" referenced-column-name="id" on-delete="CASCADE" on-update="CASCADE"/>
</one-to-one>

<one-to-many field="phonenumbers" target-entity="Phonenumber" mapped-by="user">
  <cascade>
    <cascade-persist/>
  </cascade>
  <order-by>
    <order-by-field name="number" direction="ASC" />
  </order-by>
</one-to-many>

<many-to-many field="groups" target-entity="Group">
  <cascade>
    <cascade-all/>
  </cascade>
  <join-table name="cms_users_groups">
    <join-columns>
      <join-column name="user_id" referenced-column-name="id" unique="false" />
    </join-columns>
    <inverse-join-columns>
      <join-column name="group_id" referenced-column-name="id" column-definition="INT NULL" />
    </inverse-join-columns>
  </join-table>
</many-to-many>

</entity>

```

```
</doctrine-mapping>
```

---

Be aware that class-names specified in the XML files should be fully qualified.

### 20.1.2. XML-Element Reference

---

The XML-Element reference explains all the tags and attributes that the Doctrine Mapping XSD Schema defines. You should read the Basic-, Association- and Inheritance Mapping chapters to understand what each of this definitions means in detail.

## 20.2. Defining an Entity

---

Each XML Mapping File contains the definition of one entity, specified as the `<entity />` element as a direct child of the `<doctrine-mapping />` element:

```
<doctrine-mapping>
  <entity name="MyProject\User" table="cms_users" repository-class="MyProj
  \UserRepository">
    <!-- definition here -->
  </entity>
</doctrine-mapping>
```

---

Required attributes:

- **name** - The fully qualified class-name of the entity.

Optional attributes:

- **table** - The Table-Name to be used for this entity. Otherwise the Unqualified Class-Name used by default.
- **repository-class** - The fully qualified class-name of an alternative `Doctrine\ORM\EntityRepository` implementation to be used with this entity.
- **inheritance-type** - The type of inheritance, defaults to none. A more detailed description follows in the **Defining Inheritance Mappings** section.

for change-tracking. Entities of this type can be persisted and removed though.

## 20.3. Defining Fields

Each entity class can contain zero to infinite fields that are managed by Doctrine. You can define them using the `<field />` element as a children to the `<entity />` element. The field element is only used for primitive types that are not the ID of the entity. For the ID mapping you have to use the `<id />` element.

```
<entity name="MyProject\User">

    <field name="name" type="string" length="50" />
    <field name="username" type="string" unique="true" />
    <field name="age" type="integer" nullable="true" />
    <field name="isActive" column="is_active" type="boolean" />
    <field name="weight" type="decimal" scale="5" precision="2" />
    <field name="login_count" type="integer" nullable="false">
        <options>
            <option name="comment">The number of times the user has logged i
        /option>
            <option name="default">0</option>
        </options>
    </field>
</entity>
```

Required attributes:

- name - The name of the Property/Field on the given Entity PHP class.

Optional attributes:

- type - The **Doctrine\DBAL\Types\Type** name, defaults to "string"
- column - Name of the column in the database, defaults to the field name.
- length - The length of the given type, for use with strings only.
- unique - Should this field contain a unique value across the table? Defaults to false.
- nullable - Should this field allow NULL as a value? Defaults to false.

- version - Should this field be used for optimistic locking? Only works on fields with type integer or datetime.
- scale - Scale of a decimal type.
- precision - Precision of a decimal type.
- options - Array of additional options:
  - default - The default value to set for the column if no value is supplied.
  - unsigned - Boolean value to determine if the column should be capable of representing only non-negative integers (applies only for integer column and might not be supported by all vendors).
  - fixed - Boolean value to determine if the specified length of a string column should be fixed or varying (applies only for string/binary column and might not be supported by all vendors).
  - comment - The comment of the column in the schema (might not be supported by all vendors).
  - customSchemaOptions - Array of additional schema options which are mostly vendor specific.
- column-definition - Optional alternative SQL representation for this column. This definition begins after the field-name and has to specify the complete column definition. Using this feature will turn this field dirty for Schema-Tool update commands at all times.

For more detailed information on each attribute, please refer to the DBAL **Schemas Representation** documentation.

## 20.4. Defining Identity and Generator Strategies

An entity has to have at least one `<id />` element. For composite keys you can specify more than one id-element, however surrogate keys are recommended for use with Doctrine 2. The Id field allows to define properties of the identifier and allows a subset of the `<field />` element attributes:

```
<id name="id" type="integer" column="user_id" />
</entity>
```

---

Required attributes:

- name - The name of the Property/Field on the given Entity PHP class.
- type - The **Doctrine\DBAL\Types\Type** name, preferably "string" or "integer".

Optional attributes:

- column - Name of the column in the database, defaults to the field name.

Using the simplified definition above Doctrine will use no identifier strategy for this entity. That means you have to manually set the identifier before calling **EntityManager#persist(\$entity)**. This is the so called **ASSIGNED** strategy.

If you want to switch the identifier generation strategy you have to nest a **<generator />** element inside the id-element. This of course only works for surrogate keys. For composite keys you always have to use the **ASSIGNED** strategy.

```
<entity name="MyProject\User">
  <id name="id" type="integer" column="user_id">
    <generator strategy="AUTO" />
  </id>
</entity>
```

---

The following values are allowed for the **<generator />** strategy attribute:

- AUTO - Automatic detection of the identifier strategy based on the preferred solution of database vendor.
- IDENTITY - Use of a IDENTIFY strategy such as Auto-Increment IDs available to Doctrine AFTER the INSERT statement has been executed.
- SEQUENCE - Use of a database sequence to retrieve the entity-ids. This is possible before the INSERT statement is executed.

If you are using the SEQUENCE strategy you can define an additional element to describe the sequence:



```

<id name="id" type="integer" column="user_id">
  <generator strategy="SEQUENCE" />
  <sequence-generator sequence-name="user_seq" allocation-size="5" ini
    tial-value="1" />
</id>
</entity>

```

---

Required attributes for **<sequence-generator />**:

- sequence-name - The name of the sequence

Optional attributes for **<sequence-generator />**:

- allocation-size - By how much steps should the sequence be incremented when a value is retrieved. Defaults to 1
- initial-value - What should the initial value of the sequence be.

#### NOTE

If you want to implement a cross-vendor compatible application you have to specify and additionally define the **<sequence-generator />** element, if Doctrine chooses the sequence strategy for a platform.

## 20.5. Defining a Mapped Superclass

---

Sometimes you want to define a class that multiple entities inherit from, which itself is not an entity however. The chapter on **Inheritance Mapping** describes a Mapped Superclass in detail. You can define it in XML using the **<mapped-superclass />** tag.

```

<doctrine-mapping>
  <mapped-superclass name="MyProject\BaseClass">
    <field name="created" type="datetime" />
    <field name="updated" type="datetime" />
  </mapped-superclass>
</doctrine-mapping>

```

---

- name - Class name of the mapped superclass.

You can nest any number of `<field />` and unidirectional `<many-to-one />` or `<one-to-one />` associations inside a mapped superclass.

## 20.6. Defining Inheritance Mappings

There are currently two inheritance persistence strategies that you can choose from when defining entities that inherit from each other. Single Table inheritance saves the fields of the complete inheritance hierarchy in a single table, joined table inheritance creates a table for each entity combining the fields using join conditions.

You can specify the inheritance type in the `<entity />` element and then use the `<discriminator-column />` and `<discriminator-mapping />` attributes.

```
<entity name="MyProject\Animal" inheritance-type="JOINED">
  <discriminator-column name="discr" type="string" />
  <discriminator-map>
    <discriminator-mapping value="cat" class="MyProject\Cat" />
    <discriminator-mapping value="dog" class="MyProject\Dog" />
    <discriminator-mapping value="mouse" class="MyProject\Mouse" />
  </discriminator-map>
</entity>
```

The allowed values for inheritance-type attribute are **JOINED** or **SINGLE\_TABLE**.

All inheritance related definitions have to be defined on the root entity of the hierarchy.

## 20.7. Defining Lifecycle Callbacks

You can define the lifecycle callback methods on your entities using the `<lifecycle-callbacks />` element:

```
<entity name="Doctrine\Tests\ORM\Mapping\User" table="cms_users">
```

```

<lifetime-callbacks>
  <lifetime-callback type="prePersist" method="onPrePersist" />
</lifetime-callbacks>
</entity>

```

---

## 20.8. Defining One-To-One Relations

---

You can define One-To-One Relations/Associations using the `<one-to-one />` element. The required and optional attributes depend on the associations being on the inverse or owning side.

For the inverse side the mapping is as simple as:

```

<entity class="MyProject\User">
  <one-to-one field="address" target-entity="Address" mapped-by="user" />
</entity>

```

---

Required attributes for inverse One-To-One:

- `field` - Name of the property/field on the entity's PHP class.
- `target-entity` - Name of the entity associated entity class. If this is not qualified the namespace of the current class is prepended. **IMPORTANT:** No leading backslash!
- `mapped-by` - Name of the field on the owning side (here Address entity) that contains the owning side association.

For the owning side this mapping would look like:

```

<entity class="MyProject\Address">
  <one-to-one field="user" target-entity="User" inversed-by="address" />
</entity>

```

---

Required attributes for owning One-to-One:

- `field` - Name of the property/field on the entity's PHP class.

Optional attributes for owning One-to-One:

- **inversed-by** - If the association is bidirectional the **inversed-by** attribute has to be spec with the name of the field on the inverse entity that contains the back-reference.
- **orphan-removal** - If true, the inverse side entity is always deleted when the owning sid entity is. Defaults to false.
- **fetch** - Either LAZY or EAGER, defaults to LAZY. This attribute makes only sense on the owning side, the inverse side **ALWAYS** has to use the **FETCH** strategy.

The definition for the owning side relies on a bunch of mapping defaults for the join column names. Without the nested `<join-column />` element Doctrine assumes to foreign key to be called **user\_id** on the Address Entities table. This is because the **MyProject\Address** entity is the owning side of this association, which means it contains the foreign key.

The completed explicitly defined mapping is:

```
<entity class="MyProject\Address">
  <one-to-one field="user" target-entity="User" inversed-by="address">
    <join-column name="user_id" referenced-column-name="id" />
  </one-to-one>
</entity>
```

## 20.9. Defining Many-To-One Associations

The many-to-one association is **ALWAYS** the owning side of any bidirectional association. This simplifies the mapping compared to the one-to-one case. The minimal mapping for this association looks like:

```
<entity class="MyProject\Article">
  <many-to-one field="author" target-entity="User" />
</entity>
```

Required attributes:

- **field** - Name of the property/field on the entity's PHP class.

namespace of the current class is prepended. **IMPORTANT:** No leading backslash!

Optional attributes:

- **inversed-by** - If the association is bidirectional the **inversed-by** attribute has to be spec with the name of the field on the inverse entity that contains the back-reference.
- **orphan-removal** - If true the entity on the inverse side is always deleted when the owni side entity is and it is not connected to any other owning side entity anymore. Defaults false.
- **fetch** - Either LAZY or EAGER, defaults to LAZY.

This definition relies on a bunch of mapping defaults with regards to the naming of the join-column/foreign key. The explicitly defined mapping includes a `<join-column />` tag nested inside the many-to-one association tag:

```
<entity class="MyProject\Article">
  <many-to-one field="author" target-entity="User">
    <join-column name="author_id" referenced-column-name="id" />
  </many-to-one>
</entity>
```

The **join-column** attribute **name** specifies the column name of the foreign key and the **referenced-column-name** attribute specifies the name of the primary key column on the User entity.

## 20.10. Defining One-To-Many Associations

The one-to-many association is **ALWAYS** the inverse side of any association. There exists no such thing as a uni-directional one-to-many association, which means this association only ever exists for bi-directional associations.

```
<entity class="MyProject\User">
  <one-to-many field="phonenumbers" target-entity="Phonenumber" mapped-by=
er" />
</entity>
```

- field - Name of the property/field on the entity's PHP class.
- target-entity - Name of the entity associated entity class. If this is not qualified the namespace of the current class is prepended. **IMPORTANT:** No leading backslash!
- mapped-by - Name of the field on the owning side (here Phonenummer entity) that contains the owning side association.

Optional attributes:

- fetch - Either LAZY, EXTRA\_LAZY or EAGER, defaults to LAZY.
- index-by: Index the collection by a field on the target entity.

## 20.11. Defining Many-To-Many Associations

From all the associations the many-to-many has the most complex definition. When you rely on the mapping defaults you can omit many definitions and rely on their implicit values.

```
<entity class="MyProject\User">
  <many-to-many field="groups" target-entity="Group" />
</entity>
```

Required attributes:

- field - Name of the property/field on the entity's PHP class.
- target-entity - Name of the entity associated entity class. If this is not qualified the namespace of the current class is prepended. **IMPORTANT:** No leading backslash!

Optional attributes:

- mapped-by - Name of the field on the owning side that contains the owning side association if the defined many-to-many association is on the inverse side.
- inversed-by - If the association is bidirectional the inversed-by attribute has to be specified with the name of the field on the inverse entity that contains the back-reference.
- fetch - Either LAZY, EXTRA\_LAZY or EAGER, defaults to LAZY.

The mapping defaults would lead to a join-table with the name "User\_Group" being created that contains two columns "user\_id" and "group\_id". The explicit definition of this mapping would be:

---

```
<entity class="MyProject\User">
  <many-to-many field="groups" target-entity="Group">
    <join-table name="cms_users_groups">
      <join-columns>
        <join-column name="user_id" referenced-column-name="id"/>
      </join-columns>
      <inverse-join-columns>
        <join-column name="group_id" referenced-column-name="id"/>
      </inverse-join-columns>
    </join-table>
  </many-to-many>
</entity>
```

---

Here both the **<join-columns>** and **<inverse-join-columns>** tags are necessary to tell Doctrine for which side the specified join-columns apply. These are nested inside a **<join-table />** attribute which allows to specify the table name of the many-to-many join-table.

## 20.12. Cascade Element

---

Doctrine allows cascading of several UnitOfWork operations to related entities. You can specify the cascade operations in the **<cascade />** element inside any of the association mapping tags.

---

```
<entity class="MyProject\User">
  <many-to-many field="groups" target-entity="Group">
    <cascade>
      <cascade-all/>
    </cascade>
  </many-to-many>
</entity>
```

---

Besides **<cascade-all />** the following operations can be specified by their respective tags:

- `<cascade-merge />`
- `<cascade-remove />`
- `<cascade-refresh />`

## 20.13. Join Column Element

---

In any explicitly defined association mapping you will need the `<join-column />` tag. It defines how the foreign key and primary key names are called that are used for joining two entities.

Required attributes:

- `name` - The column name of the foreign key.
- `referenced-column-name` - The column name of the associated entities primary key

Optional attributes:

- `unique` - If the join column should contain a UNIQUE constraint. This makes sense for 1 To-Many join-columns only to simulate a one-to-many unidirectional using a join-table.
- `nullable` - should the join column be nullable, defaults to true.
- `on-delete` - Foreign Key Cascade action to perform when entity is deleted, defaults to N ACTION/RESTRICT but can be set to "CASCADE".

## 20.14. Defining Order of To-Many Associations

---

You can require one-to-many or many-to-many associations to be retrieved using an additional **ORDER BY**.

```
<entity class="MyProject\User">
  <many-to-many field="groups" target-entity="Group">
    <order-by>
      <order-by-field name="name" direction="ASC" />
    </order-by>
  </many-to-many>
</entity>
```



```
</entity>
```

---

## 20.15. Defining Indexes or Unique Constraints

---

To define additional indexes or unique constraints on the entities table you can use the `<indexes />` and `<unique-constraints />` elements:

```
<entity name="Doctrine\Tests\ORM\Mapping\User" table="cms_users">

    <indexes>
        <index name="name_idx" columns="name"/>
        <index columns="user_email"/>
    </indexes>

    <unique-constraints>
        <unique-constraint columns="name,user_email" name="search_idx" />
    </unique-constraints>
</entity>
```

---

You have to specify the column and not the entity-class field names in the index and unique constraint definitions.

## 20.16. Derived Entities ID syntax

---

If the primary key of an entity contains a foreign key to another entity we speak of a derived entity relationship. You can define this in XML with the "association-key" attribute in the `<id>` tag.

```
<doctrine-mapping xmlns="http://doctrine-project.org/schemas/orm/doctrine-ma
ng"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://doctrine-project.org/schemas/orm/doctrine-r
ing
                                http://raw.github.com/doctrine/doctrine2/master/doctrine
nning.xsd">
```

```
<entity name="Application\Model\ArticleAttribute">
  <id name="article" association-key="true" />
  <id name="attribute" type="string" />

  <field name="value" type="string" />

  <many-to-one field="article" target-entity="Article" inversed-by="at
butes" />
</entity>

</doctrine-mapping>
```

---