SQLAlchemy 1.2 Documentation
Release: **1.2.14** **CURRENT RELEASE** | Release Date: November 10, 2018

SQLAlchemy 1.2 Documentation
CURRENT RELEASE
Contents | Index

Search terms: search...

SQLAlchemy ORM

Object Relational Tutorial
Mapper Configuration
Relationship Configuration

Basic Relationship Patterns

A quick walkthrough of the basic relational patterns.

The imports used for each of the following sections is as follows:

```
from sqlalchemy import Table, Column, Integer, ForeignKey
from sqlalchemy.orm import relationship
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()
```

One To Many

A one to many relationship places a foreign key on the child table referencing the parent. `relationship()` is then specified on the parent, as referencing a collection of items represented by the child:

```
class Parent(Base):
    __tablename__ = 'parent'
    id = Column(Integer, primary_key=True)
    children = relationship("Child")

class Child(Base):
    __tablename__ = 'child'
    id = Column(Integer, primary_key=True)
    parent_id = Column(Integer, ForeignKey('parent.id'))
```

To establish a bidirectional relationship in one-to-many, where the "reverse" side is a many to one, specify an additional `relationship()` and connect the two using the `relationship.back_populates` parameter:

```
class Parent(Base):
    __tablename__ = 'parent'
    id = Column(Integer, primary_key=True)
    children = relationship("Child", back_populates="parent")

class Child(Base):
    __tablename__ = 'child'
    id = Column(Integer, primary_key=True)
    parent_id = Column(Integer, ForeignKey('parent.id'))
    parent = relationship("Parent", back_populates="children")
```

`Child` will get a `parent` attribute with many-to-one semantics.

Alternatively, the `backref` option may be used on a single `relationship()` instead of using `back_populates`:

```
class Parent(Base):
    __tablename__ = 'parent'
    id = Column(Integer, primary_key=True)
    children = relationship("Child", backref="parent")
```

Many To One

Many to one places a foreign key in the parent table referencing the child. `relationship()` is declared on the parent, where a new scalar-holding attribute will be created:

```
class Parent(Base):
    __tablename__ = 'parent'
    id = Column(Integer, primary_key=True)
    child_id = Column(Integer, ForeignKey('child.id'))
    child = relationship("Child")

class Child(Base):
    __tablename__ = 'child'
    id = Column(Integer, primary_key=True)
```

Bidirectional behavior is achieved by adding a second `relationship()` and applying the `relationship.back_populates` parameter in both directions:

```
class Parent(Base):
    __tablename__ = 'parent'
    id = Column(Integer, primary_key=True)
    child_id = Column(Integer, ForeignKey('child.id'))
    child = relationship("Child", back_populates="parents")

class Child(Base):
    __tablename__ = 'child'
    id = Column(Integer, primary_key=True)
    parents = relationship("Parent", back_populates="child")
```

Alternatively, the `backref` parameter may be applied to a single `relationship()`, such as `Parent.child`:

```
class Parent(Base):
    __tablename__ = 'parent'
    id = Column(Integer, primary_key=True)
    child_id = Column(Integer, ForeignKey('child.id'))
    child = relationship("Child", backref="parents")
```

One To One

One To One is essentially a bidirectional relationship with a scalar attribute on both sides. To achieve this, the `uselist` flag indicates the placement of a scalar attribute instead of a collection on the "many" side of the relationship. To convert one-to-many into one-to-one:

```
class Parent(Base):
    __tablename__ = 'parent'
    id = Column(Integer, primary_key=True)
    child = relationship("Child", uselist=False, back_populates="par

class Child(Base):
    __tablename__ = 'child'
    id = Column(Integer, primary_key=True)
    parent_id = Column(Integer, ForeignKey('parent.id'))
    parent = relationship("Parent", back_populates="child")
```

Or for many-to-one:

```
class Parent(Base):
    __tablename__ = 'parent'
    id = Column(Integer, primary_key=True)
    child_id = Column(Integer, ForeignKey('child.id'))
    child = relationship("Child", back_populates="parent")

class Child(Base):
    __tablename__ = 'child'
    id = Column(Integer, primary_key=True)
    parent = relationship("Parent", back_populates="child", uselist=
```

As always, the `relationship.backref` and `backref()` functions may be used in lieu of the `relationship.back_populates` approach; to specify `uselist` on a backref, use the `backref()` function:

```
from sqlalchemy.orm import backref

class Parent(Base):
    __tablename__ = 'parent'
    id = Column(Integer, primary_key=True)
    child_id = Column(Integer, ForeignKey('child.id'))
    child = relationship("Child", backref=backref("parent", uselist=
```

Many To Many

Many to Many adds an association table between two classes. The association table is indicated by the `secondary` argument to `relationship()`. Usually, the `Table` uses the `MetaData` object associated with the declarative base class, so that the `ForeignKey` directives can locate the remote tables with which to link:

```
association_table = Table('association', Base.metadata,
    Column('left_id', Integer, ForeignKey('left.id')),
    Column('right_id', Integer, ForeignKey('right.id'))
)

class Parent(Base):
    __tablename__ = 'left'
    id = Column(Integer, primary_key=True)
    children = relationship("Child",
                    secondary=association_table)

class Child(Base):
    __tablename__ = 'right'
    id = Column(Integer, primary_key=True)
```

For a bidirectional relationship, both sides of the relationship contain a collection. Specify using `relationship.back_populates`, and for each `relationship()` specify the common association table:

```
association_table = Table('association', Base.metadata,
    Column('left_id', Integer, ForeignKey('left.id')),
    Column('right_id', Integer, ForeignKey('right.id'))
)

class Parent(Base):
    __tablename__ = 'left'
    id = Column(Integer, primary_key=True)
    children = relationship(
        "Child",
        secondary=association_table,
        back_populates="parents")

class Child(Base):
    __tablename__ = 'right'
    id = Column(Integer, primary_key=True)
    parents = relationship(
        "Parent",
        secondary=association_table,
        back_populates="children")
```

When using the `backref` parameter instead of `relationship.back_populates`, the backref will automatically use the same `secondary` argument for the reverse relationship:

```
association_table = Table('association', Base.metadata,
    Column('left_id', Integer, ForeignKey('left.id')),
    Column('right_id', Integer, ForeignKey('right.id'))
)

class Parent(Base):
    __tablename__ = 'left'
    id = Column(Integer, primary_key=True)
    children = relationship("Child",
                    secondary=association_table,
                    backref="parents")

class Child(Base):
```

```
    __tablename__ = 'right'
    id = Column(Integer, primary_key=True)
```

The `secondary` argument of `relationship()` also accepts a callable that returns the ultimate argument, which is evaluated only when mappers are first used. Using this, we can define the `association_table` at a later point, as long as it's available to the callable after all module initialization is complete:

```
class Parent(Base):
    __tablename__ = 'left'
    id = Column(Integer, primary_key=True)
    children = relationship("Child",
                    secondary=lambda: association_table,
                    backref="parents")
```

With the declarative extension in use, the traditional "string name of the table" is accepted as well, matching the name of the table as stored in `Base.metadata.tables`:

```
class Parent(Base):
    __tablename__ = 'left'
    id = Column(Integer, primary_key=True)
    children = relationship("Child",
                    secondary="association",
                    backref="parents")
```

Deleting Rows from the Many to Many Table

A behavior which is unique to the `secondary` argument to `relationship()` is that the `Table` which is specified here is automatically subject to INSERT and DELETE statements, as objects are added or removed from the collection. There is **no need to delete from this table manually**. The act of removing a record from the collection will have the effect of the row being deleted on flush:

```
# row will be deleted from the "secondary" table
# automatically
myparent.children.remove(somechild)
```

A question which often arises is how the row in the "secondary" table can be deleted when the child object is handed directly to `Session.delete()`:

```
session.delete(somechild)
```

There are several possibilities here:

- If there is a `relationship()` from `Parent` to `Child`, but there is **not** a reverse-relationship that links a particular `Child` to each `Parent`, SQLAlchemy will not have any awareness that when deleting this particular `Child` object, it needs to maintain

the “secondary” table that links it to the `Parent`. No delete of the “secondary” table will occur.

- If there is a relationship that links a particular `Child` to each `Parent`, suppose it’s called `Child.parents`, SQLAlchemy by default will load in the `Child.parents` collection to locate all `Parent` objects, and remove each row from the “secondary” table which establishes this link. Note that this relationship does not need to be bidirectional; SQLAlchemy is strictly looking at every `relationship()` associated with the `Child` object being deleted.
- A higher performing option here is to use ON DELETE CASCADE directives with the foreign keys used by the database. Assuming the database supports this feature, the database itself can be made to automatically delete rows in the “secondary” table as referencing rows in “child” are deleted. SQLAlchemy can be instructed to forego actively loading in the `Child.parents` collection in this case using the `passive_deletes` directive on `relationship()`; see Using Passive Deletes for more details on this.

Note again, these behaviors are *only* relevant to the `secondary` option used with `relationship()`. If dealing with association tables that are mapped explicitly and are *not* present in the `secondary` option of a relevant `relationship()`, cascade rules can be used instead to automatically delete entities in reaction to a related entity being deleted - see Cascades for information on this feature.

Association Object

The association object pattern is a variant on many-to-many: it’s used when your association table contains additional columns beyond those which are foreign keys to the left and right tables. Instead of using the `secondary` argument, you map a new class directly to the association table. The left side of the relationship references the association object via one-to-many, and the association class references the right side via many-to-one. Below we illustrate an association table mapped to the `Association` class which includes a column called `extra_data`, which is a string value that is stored along with each association between `Parent` and `Child`:

```
class Association(Base):
    __tablename__ = 'association'
    left_id = Column(Integer, ForeignKey('left.id'), primary_key=Tru
    right_id = Column(Integer, ForeignKey('right.id'), primary_key=T
    extra_data = Column(String(50))
    child = relationship("Child")

class Parent(Base):
    __tablename__ = 'left'
    id = Column(Integer, primary_key=True)
    children = relationship("Association")
```

```
class Child(Base):
    __tablename__ = 'right'
    id = Column(Integer, primary_key=True)
```

As always, the bidirectional version makes use of `relationship.back_populates` or `relationship.backref`:

```
class Association(Base):
    __tablename__ = 'association'
    left_id = Column(Integer, ForeignKey('left.id'), primary_key=Tru
    right_id = Column(Integer, ForeignKey('right.id'), primary_key=T
    extra_data = Column(String(50))
    child = relationship("Child", back_populates="parents")
    parent = relationship("Parent", back_populates="children")

class Parent(Base):
    __tablename__ = 'left'
    id = Column(Integer, primary_key=True)
    children = relationship("Association", back_populates="parent")

class Child(Base):
    __tablename__ = 'right'
    id = Column(Integer, primary_key=True)
    parents = relationship("Association", back_populates="child")
```

Working with the association pattern in its direct form requires that child objects are associated with an association instance before being appended to the parent; similarly, access from parent to child goes through the association object:

```
# create parent, append a child via association
p = Parent()
a = Association(extra_data="some data")
a.child = Child()
p.children.append(a)

# iterate through child objects via association, including associati
# attributes
for assoc in p.children:
    print(assoc.extra_data)
    print(assoc.child)
```

To enhance the association object pattern such that direct access to the `Association` object is optional, SQLAlchemy provides the Association Proxy extension. This extension allows the configuration of attributes which will access two “hops” with a single access, one “hop” to the associated object, and a second to a target attribute.

> **Warning**
>
> The association object pattern **does not coordinate changes with a separate relationship that maps the**

association table as “secondary”.

Below, changes made to `Parent.children` will not be coordinated with changes made to `Parent.child_associations` or `Child.parent_associations` in Python; while all of these relationships will continue to function normally by themselves, changes on one will not show up in another until the `Session` is expired, which normally occurs automatically after `Session.commit()`:

```
class Association(Base):
    __tablename__ = 'association'

    left_id = Column(Integer, ForeignKey('left.id'),
    right_id = Column(Integer, ForeignKey('right.id')
    extra_data = Column(String(50))

    child = relationship("Child", backref="parent_ass
    parent = relationship("Parent", backref="child_as

class Parent(Base):
    __tablename__ = 'left'
    id = Column(Integer, primary_key=True)

    children = relationship("Child", secondary="assoc

class Child(Base):
    __tablename__ = 'right'
    id = Column(Integer, primary_key=True)
```

Additionally, just as changes to one relationship aren’t reflected in the others automatically, writing the same data to both relationships will cause conflicting INSERT or DELETE statements as well, such as below where we establish the same relationship between a `Parent` and `Child` object twice:

```
p1 = Parent()
c1 = Child()
p1.children.append(c1)

# redundant, will cause a duplicate INSERT on Associa
p1.parent_associations.append(Association(child=c1))
```

It’s fine to use a mapping like the above if you know what you’re doing, though it may be a good idea to apply the `viewonly=True` parameter to the “secondary” relationship to avoid the issue of redundant changes being logged. However, to get a foolproof pattern that allows a simple two-object `Parent->Child` relationship while still using the association object pattern, use the association proxy extension as documented at Association Proxy.

Previous: Relationship Configuration Next: Adjacency List Relationships