Declaring Models

Generally Flask-SQLAlchemy behaves like a properly configured declarative base from the `declarative` extension. As such we recommend reading the SQLAlchemy docs for a full reference. However the most common use cases are also documented here.

Things to keep in mind:

- The baseclass for all your models is called `db.Model`. It's stored on the SQLAlchemy instance you have to create. See Quickstart for more details.
- Some parts that are required in SQLAlchemy are optional in Flask-SQLAlchemy. For instance the table name is automatically set for you unless overridden. It's derived from the class name converted to lowercase and with "CamelCase" converted to "camel_case". To override the table name, set the `__tablename__` class attribute.

Simple Example

A very simple example:

```python
class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(80), unique=True, nullable=False)
    email = db.Column(db.String(120), unique=True, nullable=False)

    def __repr__(self):
        return '<User %r>' % self.username
```

Use `Column` to define a column. The name of the column is the name you assign it to. If you want to use a different name in the table you can provide an optional first argument which is a string with the desired column name. Primary keys are marked with `primary_key=True`. Multiple keys can be marked as primary keys in which case they become a compound primary key.

The types of the column are the first argument to `Column`. You can either provide them directly or call them to further specify them (like providing a length). The following types are the most common:

`Integer`	an integer
`String(size)`	a string with a maximum length (optional in some databases, e.g. PostgreSQL)
`Text`	some longer unicode text

v: 2.x

`DateTime`	date and time expressed as Python `datetime` object.
`Float`	stores floating point values
`Boolean`	stores a boolean value
`PickleType`	stores a pickled Python object
`LargeBinary`	stores large arbitrary binary data

One-to-Many Relationships

The most common relationships are one-to-many relationships. Because relationships are declared before they are established you can use strings to refer to classes that are not created yet (for instance if `Person` defines a relationship to `Address` which is declared later in the file).

Relationships are expressed with the **`relationship()`** function. However the foreign key has to be separately declared with the **`ForeignKey`** class:

```python
class Person(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(50), nullable=False)
    addresses = db.relationship('Address', backref='person', lazy=True)

class Address(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    email = db.Column(db.String(120), nullable=False)
    person_id = db.Column(db.Integer, db.ForeignKey('person.id'),
        nullable=False)
```

What does **`db.relationship()`** do? That function returns a new property that can do multiple things. In this case we told it to point to the `Address` class and load multiple of those. How does it know that this will return more than one address? Because SQLAlchemy guesses a useful default from your declaration. If you would want to have a one-to-one relationship you can pass `uselist=False` to **`relationship()`**.

Since a person with no name or an email address with no address associated makes no sense, `nullable=False` tells SQLAlchemy to create the column as `NOT NULL`. This is implied for primary key columns, but it's a good idea to specify it for all other columns to make it clear to other people working on your code that you did actually want a nullable column and did not just forget to add it.

So what do `backref` and `lazy` mean? `backref` is a simple way to also declare a new property on the `Address` class. You can then also use `my_address.person` to get to the person at that address. `lazy` defines when SQLAlchemy will load the data from the database:

v: 2.x

- `'select'` / `True` (which is the default, but explicit is better than implicit) means that SQLAlchemy will load the data as necessary in one go using a standard select statement.

- `'joined'` / `False` tells SQLAlchemy to load the relationship in the same query as the parent using a `JOIN` statement.
- `'subquery'` works like `'joined'` but instead SQLAlchemy will use a subquery.
- `'dynamic'` is special and can be useful if you have many items and always want to apply additional SQL filters to them. Instead of loading the items SQLAlchemy will return another query object which you can further refine before loading the items. Note that this cannot be turned into a different loading strategy when querying so it's often a good idea to avoid using this in favor of `lazy=True`. A query object equivalent to a dynamic `user.addresses` relationship can be created using **`Address.query.with_parent(user)`** while still being able to use lazy or eager loading on the relationship itself as necessary.

How do you define the lazy status for backrefs? By using the **`backref()`** function:

```
class Person(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(50), nullable=False)
    addresses = db.relationship('Address', lazy='select',
        backref=db.backref('person', lazy='joined'))
```

Many-to-Many Relationships

If you want to use many-to-many relationships you will need to define a helper table that is used for the relationship. For this helper table it is strongly recommended to *not* use a model but an actual table:

```
tags = db.Table('tags',
    db.Column('tag_id', db.Integer, db.ForeignKey('tag.id'), primary_key=True)
    db.Column('page_id', db.Integer, db.ForeignKey('page.id'), primary_key=Tru
)

class Page(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    tags = db.relationship('Tag', secondary=tags, lazy='subquery',
        backref=db.backref('pages', lazy=True))

class Tag(db.Model):
    id = db.Column(db.Integer, primary_key=True)
```

Here we configured `Page.tags` to be loaded immediately after loading a Page, but using a separate query. This always results in two queries when retrieving a Page, but when querying for multiple pages you will not get additional queries.

The list of pages for a tag on the other hand is something that's rarely needed. For example, you won't need that list when retrieving the tags for a specific page. Therefore, the backref is lazy-loaded so that accessing it for the first time will trigger a query to get the list of pages for that

v: 2.x

tag. If you need to apply further query options on that list, you could either switch to the `'dynamic'` strategy - with the drawbacks mentioned above - or get a query object using **`Page.query.with_parent(some_tag)`** and then use it exactly as you would with the query object from a dynamic relationship.

v: 2.x