A Minimal Application

For the common case of having one Flask application all you have to do is to create your Flask application, load the configuration of choice and then create the **SQLAlchemy** object by passing it the application.

Once created, that object then contains all the functions and helpers from both **sqlalchemy** and **sqlalchemy.orm**. Furthermore it provides a class called `Model` that is a declarative base which can be used to declare models:

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:////tmp/test.db'
db = SQLAlchemy(app)


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
```

To create the initial database, just import the `db` object from an interactive Python shell and run the **SQLAlchemy.create_all()** method to create the tables and database:

```
>>> from yourapplication import db
>>> db.create_all()
```

Boom, and there is your database. Now to create some users:

```
>>> from yourapplication import User
>>> admin = User(username='admin', email='admin@example.com')
>>> guest = User(username='guest', email='guest@example.com')
```

But they are not yet in the database, so let's make sure they are:

```
>>> db.session.add(admin)
>>> db.session.add(guest)
>>> db.session.commit()
```

Accessing the data in database is easy as a pie:

```
>>> User.query.all()
[<User u'admin'>, <User u'guest'>]
>>> User.query.filter_by(username='admin').first()
<User u'admin'>
```

Note how we never defined a `__init__` method on the `User` class? That's because SQLAlchemy adds an implicit constructor to all model classes which accepts keyword arguments for all its columns and relationships. If you decide to override the constructor for any reason, make sure to keep accepting `**kwargs` and call the super constructor with those `**kwargs` to preserve this behavior:

```
class Foo(db.Model):
    # ...
    def __init__(self, **kwargs):
        super(Foo, self).__init__(**kwargs)
        # do custom stuff
```

Simple Relationships

SQLAlchemy connects to relational databases and what relational databases are really good at are relations. As such, we shall have an example of an application that uses two tables that have a relationship to each other:

```
from datetime import datetime


class Post(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    title = db.Column(db.String(80), nullable=False)
    body = db.Column(db.Text, nullable=False)
    pub_date = db.Column(db.DateTime, nullable=False,
        default=datetime.utcnow)

    category_id = db.Column(db.Integer, db.ForeignKey('category.id'),
        nullable=False)
    category = db.relationship('Category',
        backref=db.backref('posts', lazy=True))

    def __repr__(self):
        return '<Post %r>' % self.title


class Category(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(50), nullable=False)

    def __repr__(self):
        return '<Category %r>' % self.name
```

First let's create some objects:

```
>>> py = Category(name='Python')
>>> Post(title='Hello Python!', body='Python is pretty cool',
category=py)
>>> p = Post(title='Snakes', body='Ssssssss')
>>> py.posts.append(p)
>>> db.session.add(py)
```

As you can see, there is no need to add the `Post` objects to the session. Since the `Category` is part of the session all objects associated with it through relationships will be added too. It does not matter whether **`db.session.add()`** is called before or after creating these objects. The association can also be done on either side of the relationship - so a post can be created with a category or it can be added to the list of posts of the category.

Let's look at the posts. Accessing them will load them from the database since the relationship is lazy-loaded, but you will probably not notice the difference - loading a list is quite fast:

```
>>> py.posts
[<Post 'Hello Python!'>, <Post 'Snakes'>]
```

While lazy-loading a relationship is fast, it can easily become a major bottleneck when you end up triggering extra queries in a loop for more than a few objects. For this case, SQLAlchemy lets you override the loading strategy on the query level. If you wanted a single query to load all categories and their posts, you could do it like this:

```
>>> from sqlalchemy.orm import joinedload
>>> query = Category.query.options(joinedload('posts'))
>>> for category in query:
...     print category, category.posts
<Category u'Python'> [<Post u'Hello Python!'>, <Post u'Snakes'>]
```

If you want to get a query object for that relationship, you can do so using **`with_parent()`**. Let's exclude that post about Snakes for example:

```
>>> Post.query.with_parent(py).filter(Post.title != 'Snakes').all()
[<Post 'Hello Python!'>]
```

Road to Enlightenment

The only things you need to know compared to plain SQLAlchemy are:

1. **`SQLAlchemy`** gives you access to the following things:
   - all the functions and classes from **`sqlalchemy`** and **`sqlalchemy.orm`**
   - a preconfigured scoped session called `session`
   - the **`metadata`**
   - the **`engine`**
   - a **`SQLAlchemy.create_all()`** and **`SQLAlchemy.drop_all()`** methods to create and drop tables according to the models.
   - a **`Model`** baseclass that is a configured declarative base.
2. The **`Model`** declarative base class behaves like a regular Python class but has a `query` attribute attached that

can be used to query the model.
(**`Model`** and **`BaseQuery`**)

3. You have to commit the session, but you don’t have to remove it at the end of the request, Flask-SQLAlchemy does that for you.