

# Using SQLAlchemy

## A next-generation Python Object Relational mapper

Skill Level: Intermediate

Noah Gift ([noah.gift@giftcs.com](mailto:noah.gift@giftcs.com))

Software Engineer

Giftcs LLC, Cloud Seed Computing LLC

12 Aug 2008

SQLAlchemy is a next-generation Python Object Relational mapper. Learn how to use the new 0.5 API, work with third-party components, and build a basic Web application.

### Introduction

Object Relational Mappers, or ORM's, have developed a substantial amount of buzz in the last few years. Most of this buzz is because ORM's are most often talked about within the confines of Web application frameworks, as they are critical component in a Rapid Development stack. Some Web frameworks like Django and Ruby on Rails have taken the approach of designing a monolithic stack that tightly integrates a homegrown ORM to the framework, and others like Pylons, Turbogears, and Grok have decided on more component-based architecture with swappable third-party components. Each approach has its advantages as a tight integration can allow for a very cohesive experience if the problem maps to the framework, and a component-based architecture allows for maximum flexibility in design. This article is not about Web frameworks, though; it is about SQLAlchemy.

While SQLAlchemy is used in many of the next-generation Python Web frameworks built on top of the WSGI specification, it is developed as a standalone project by Mike Bayer, and a core team of developers. One of the advantages to using a standalone ORM like SQLAlchemy is that it allows a developer to think about the data model first, and lets them decide about how they want to visualize the data later, be it in a command-line tool, or a Web framework, or a GUI framework. This is

a very different approach to development than deciding to first use a Web framework, or GUI framework, and then deciding how to use the data model within the confines of what the frameworks philosophy will allow you to do.

### What is WSGI?

WSGI is a specification that next-generation Python Web frameworks, applications, and servers correspond to. One of the interesting aspects of WSGI is the creation of Python middleware for use in Web applications that are created in Python or any language. Please see the [resources](#) for many links on WSGI and WSGI communities such as Pypefitters.

One of the goals of SQLAlchemy is to provide an enterprise-level persistence pattern that works with a wide range of databases such as SQLite, MySQL, Postgres, Oracle, MS-SQL, SQLServer, and Firebird. SQLAlchemy is in very active development and the most current API revolves around version 0.5. Please see the resources section for links to the official API documentation, tutorials, and a book on SQLAlchemy.

One of the testaments to the success of SQLAlchemy is the rich community that has developed around it. There are several extensions and plugins to SQLAlchemy including: declarative, Migrate, Elixir, SQLSoup, django-sqlalchemy, DBSprockets, FormAlchemy, and z3c.sqlalchemy. In this article we go through a tutorial on the new 0.5 API, examine some of the third-party libraries, and finally look at how it can be used in Pylons.

### Who is Mike Bayer?

Michael Bayer is a NYC-based software contractor with a decade of experience dealing with relational databases of all shapes and sizes. After writing many homegrown database abstraction layers in such languages as C, Java™, and Perl, and finally after several years of practice working with a huge multi-server Oracle system for Major League Baseball, he wrote SQLAlchemy as the 'ultimate toolset' for generating SQL and dealing with databases overall. The goal is to contribute towards a world-class, one-of-a-kind toolset for Python, helping to make Python the universally popular programming platform it deserves to be.

## Installation

This article assumes you will use Python 2.5 or greater, and subversion installed. Python 2.5 includes the SQLite database and as such serves as a great in memory tool for experimenting with SQLAlchemy. If you have Python 2.5 installed, you will only need to install the sqlalchemy 0.5 beta through the use of setuptools. To get the setup tools script, download and run these four commands in your terminal:

```
wget http://peak.telecommunity.com/dist/ez_setup.py
python ez_setup.py
sudo easy_install http://svn.sqlalchemy.org/sqlalchemy/trunk
sudo easy_install ipython
```

The first three lines of code check out the latest version of sqlalchemy and add it as a package to the Python installation on your local system. The final code snippet installs IPython, which is a useful interactive Python interpreter that I will use throughout the article. The first thing to do is to test the version of SQLAlchemy installed. You can test that you have version 0.5.x by issuing this command in the IPython, or regular Python, interpreter.

```
In [1]: import sqlalchemy
In [2]: sqlalchemy.__version__
Out[2]: '0.5.0beta1'
```

## SQLAlchemy 0.5 quick-start guide

The new release of 0.5 brought about a few significant changes to SQLAlchemy. Here is a quick rundown of the changes:

- Declarative extension is the recommended way to start in most cases.
- `session.query()` can accept any combination of class/column expressions.
- `session.query()` is also more or less an ORM-enabled replacement for `select()`.
- Query has some experimental `update()/delete()` methods on it for criterion-based updates/deletes.
- The Session expires itself after `rollback()` and `commit()`; so using the default `sessionmaker()` means you usually don't have to `clear()` or `close()`; Objects synchronize with the current transaction automatically.
- Things go into the Session using `session.add()`, `session.add_all()` (`save/update/save_or_update` are deprecated).

Although the declarative extension has been in SQLAlchemy since 0.4, it has undergone some small changes that make it a powerful shortcut for most SQLAlchemy projects. The new declarative syntax allows for the creation of a table, class, and mapping to the database in one step. Let's take a look at how this new

syntax works for a tool that I wrote to keep track of filesystem changes.

## New SQLAlchemy declarative style

```
#!/usr/bin/env python2.5
#Noah Gift

from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy import Table, Column, Integer, String, MetaData, ForeignKey

Base = declarative_base()
class Filesystem(Base):
    __tablename__ = 'filesystem'

    path = Column(String, primary_key=True)
    name = Column(String)

    def __init__(self, path,name):
        self.path = path
        self.name = name

    def __repr__(self):
        return "<Metadata('%s','%s')>" % (self.path,self.name)
```

With this new declarative style, SQLAlchemy is able to create a table in a database, create a class, and the mapping between the class and the table in one spot. If you are new to SQLAlchemy, you should probably learn this way of setting up the ORM, but it is also good to know there is a more explicit way of controlling each of these steps, as well, if your projects require that level of detail.

If you look at this snippet of code, it is important to point out a few things that might trip up people who are new to either SQLAlchemy or to the new declarative extension. The first thing to point out is that the line

```
Base = declarative_base()
```

creates a class that you then inherit from in the Filesystem Class. If you save and run the code in the article `declarative_style`, and then import it in IPython, you will see this output:

```
In [2]: declarative_style.Filesystem?
Type:      DeclarativeMeta
Base Class: <class 'sqlalchemy.ext.declarative.DeclarativeMeta'>
String Form: <class 'declarative_style.Filesystem'>
```

This DeclarativeMeta type is the magic that allows all of the actions to occur in one simple class definition.

Another thing to point out about this example is that it doesn't actually do anything yet. The actual table will not be created until you run code to create a table, and you

also need to define what database engine SQLAlchemy will use. Those two lines of code look like this:

```
engine = create_engine('sqlite:///meta.db', echo=True)
Base.metadata.create_all(engine)
```

SQLite is an ideal choice for experimentation with SQLAlchemy, and you can choose to either use an in-memory database, in which your line would look like this instead:

```
engine = create_engine('sqlite:///memory:', echo=True)
```

or just create a simple file, as the first example demonstrates. If you choose to create a SQLite file-based database, you can easily start over from scratch without deleting the file by just dropping all of the tables in the database. You can do that by issuing this line of code:

```
Base.metadata.drop_all(engine)
```

At this point, we know enough to create a SQLAlchemy project and control the database from the SQLAlchemy API. The only other major item to tackle before getting into a real-life example is the concept of a session. The SQLAlchemy "official" documentation describes the session as the handle to the database. In practical use, it allows for distinct, transaction-based, connections to occur from a pool of connections that SQLAlchemy has waiting. Inside of a session it is typical to add data to the database, perform queries, or delete data.

In order to create a session, perform these sequential steps:

```
#establish Session type, only need to be done once for all sessions
Session = sessionmaker(bind=engine)
#create record object
create_record = Filesystem("/tmp/foo.txt", "foo.txt")
#make a unique session
session = Session()
#do stuff in session. We are adding a record here
session.add(create_record)
#commit the transaction
session.commit()
```

This is really all that needs to be done to get up and running with SQLAlchemy. While SQLAlchemy has a very sophisticated API that does many complicated things, it is really very simple to get started with. To end this section, I should also point out that when you created the engine in the example above, you did it with `echo=True`.

This is a very handy way to see the actual SQL that is created by SQLAlchemy. It is highly recommended to use this if you are new to SQLAlchemy, as it will take away any perceived magic about what SQLAlchemy does. Now run some of the code you created and see the SQL to create a table.

### Listing 1. SQLAlchemy SQL table-creation output

```
2008-06-22 05:33:46,403 INFO
sqlalchemy.engine.base.Engine.0x..ec PRAGMA
table_info("filesystem")
2008-06-22 05:33:46,404 INFO sqlalchemy.engine.base.Engine.0x..ec {}
2008-06-22 05:33:46,405 INFO sqlalchemy.engine.base.Engine.0x..ec
CREATE TABLE filesystem (
    path VARCHAR NOT NULL,
    name VARCHAR,
    PRIMARY KEY (path)
)
```

### Pylesystem: A realtime filesystem metadata indexer similar to Spotlight or Beagle

Talking abstractly about how you could use a tool is very hard to follow for many people, so I'll show you how to create a metadata tool using SQLAlchemy. The goal of this tool is to monitor the filesystem, create and delete events, and keep these changes in a SQLAlchemy database. If you have ever used Spotlight on OS X, or Beagle on Linux®, then you have used a real-time filesystem indexing tool. To follow along, you need to run the Linux kernel 2.6.13 or higher.

The next example is a bit big, coming in at a little under 100 lines of code. Look at the whole example, run it, and then I will walk through what each section does. In order to run this script, you must perform these actions in the terminal:

1. `wget http://peak.telecommunity.com/dist/ez_setup.py`
2. `sudo python ez_setup.py`
3. `sudo easy_install`  
`"http://git.dbzteam.org/?p=pyinotify.git;a=snapshot;h=HEAD;sf=tgz"`
4. `sudo easy_install http://svn.sqlalchemy.org/sqlalchemy/trunk`

### Listing 2. Filesystem event-monitoring database

```
#!/usr/bin/env python2.5
#Noah Gift 06/21/08
#tweaks by Mike Bayer 06/22/08
import os

from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy import Table, Column, Integer, String, MetaData, ForeignKey
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker
```

```

from sqlalchemy.orm import scoped_session

from pyinotify import *

path = "/tmp"

#SQLAlchemy
engine = create_engine('sqlite:///meta.db', echo=True)
Base = declarative_base()
Session = scoped_session(sessionmaker(bind=engine))

class Filesystem(Base):
    __tablename__ = 'filesystem'

    path = Column(String, primary_key=True)
    name = Column(String)

    def __init__(self, path,name):
        self.path = path
        self.name = name

    def __repr__(self):
        return "<Metadata('%s','%s')>" % (self.path,self.name)

def transactional(fn):
    """add transactional semantics to a method."""

    def transact(self, *args):
        session = Session()
        try:
            fn(self, session, *args)
            session.commit()
        except:
            session.rollback()
            raise
        transact.__name__ = fn.__name__
    return transact

class ProcessDir(ProcessEvent):
    """Performs Actions based on mask values"""

    @transactional
    def process_IN_CREATE(self, session, event):
        print "Creating File and File Record:", event.pathname
        create_record = Filesystem(event.pathname, event.path)
        session.add(create_record)

    @transactional
    def process_IN_DELETE(self, session, event):
        print "Removing:", event.pathname
        delete_record = session.query(Filesystem).\
            filter_by(path=event.pathname).one()
        session.delete(delete_record)

def init_repository():
    #Drop the table, then create again with each run
    Base.metadata.drop_all(engine)
    Base.metadata.create_all(engine)
    session = Session()

    #Initial Directory Walking Addition Brute Force
    for dirpath, dirnames, filenames in os.walk(path):
        for file in filenames:
            fullpath = os.path.join(dirpath, file)
            record = Filesystem(fullpath, file)
            session.add(record)
        session.flush()

```

```

    for record in session.query(Filesystem):
        print "Database Record Number: Path: %s , File: %s " \
              % (record.path, record.name)

    session.commit()

if __name__ == "__main__":

    init_repository()

    #Pyionotify
    wm = WatchManager()
    mask = IN_DELETE | IN_CREATE
    notifier = ThreadedNotifier(wm, ProcessDir())
    notifier.start()

    wdd = wm.add_watch(path, mask, rec=True)

```

To see the results of this script in action, you must have two terminal windows open. In the first window, run the pylesystem.py script. You will see a bunch of output that looks something like this (Please note that the following version is partially suppressed for space):

```

2008-06-22 07:18:08,707 INFO
sqlalchemy.engine.base.Engine.0x..ec ['/tmp/ba.txt', 'ba.txt']
2008-06-22 07:18:08,710 INFO
sqlalchemy.engine.base.Engine.0x..ec COMMIT
2008-06-22 07:18:08,715 INFO
sqlalchemy.engine.base.Engine.0x..ec BEGIN
2008-06-22 07:18:08,716 INFO
sqlalchemy.engine.base.Engine.0x..ec SELECT filesystem.path
AS filesystem_path, filesystem.name AS filesystem_name
FROM filesystem
2008-06-22 07:18:08,716 INFO sqlalchemy.engine.base.Engine.0x..ec []
Database Record Number: Path: /tmp/ba.txt , File: ba.txt

```

This first script runs a multi-threaded file system event-monitoring engine that writes all create and delete changes to the /tmp directory into the sqlalchemy database. Just a note: because it is multi-threaded, you will have to type **Control + \** to stop the threaded application when you finish the tutorial.

Now that it is running, you can create events in the second terminal window, and the newly created files or deleted files will be added or subtracted to the database in real time. If you simply "touch" a file in /tmp, such as `touch foobar.txt`, you will see this output in the first window:

```

Creating File and File Record: /tmp/foobar.txt
2008-06-22 08:02:19,468 INFO
sqlalchemy.engine.base.Engine.0x..4c BEGIN
2008-06-22 08:02:19,471 INFO
sqlalchemy.engine.base.Engine.0x..4c INSERT INTO filesystem (path, name) VALUES (?, ?)
2008-06-22 08:02:19,472 INFO
sqlalchemy.engine.base.Engine.0x..4c ['/tmp/foobar.txt', '/tmp']
2008-06-22 08:02:19,473 INFO

```

```
sqlalchemy.engine.base.Engine.0x..4c COMMIT
```

Remember earlier you enabled the SQL to echo? Because of this, you can see the SQL statements as the code adds this new entry to the filesystem. If you now delete that file, you can see the delete happen, as well. Here is what it looks like when you type in a `rm` statement, `rm foobar.txt`:

```
Removing: /tmp/foobar.txt
2008-06-22 08:06:01,727 INFO
sqlalchemy.engine.base.Engine.0x..4c BEGIN
2008-06-22 08:06:01,733 INFO
sqlalchemy.engine.base.Engine.0x..4c SELECT filesystem.path
AS filesystem_path, filesystem.name AS filesystem_name
FROM filesystem
WHERE filesystem.path = ?
LIMIT 2 OFFSET 0
2008-06-22 08:06:01,733 INFO
sqlalchemy.engine.base.Engine.0x..4c ['/tmp/foobar.txt']
2008-06-22 08:06:01,736 INFO
sqlalchemy.engine.base.Engine.0x..4c DELETE FROM filesystem WHERE filesystem.path = ?
2008-06-22 08:06:01,736 INFO
sqlalchemy.engine.base.Engine.0x..4c [u'/tmp/foobar.txt']
2008-06-22 08:06:01,737 INFO
sqlalchemy.engine.base.Engine.0x..4c COMMIT
```

In the `Filesystem` class, you added a transactional method that you will use a decorator to handle the semantics of the commits to the database for filesystem events. The actual `Filesystem` I/O monitoring in `Pyinotify` is done by the `ProcessDir` class, which inherits and overrides methods from `ProcessEvents`. If you notice the respective methods, `process_IN_CREATE` and `process_IN_DELETE` each have the fancy transactional decorator attached to them. All they do then is take the create or delete events and make the changes to the database.

There is also a method called `initial_repository` that populates the database each time the script is run by destroying and then recreating the tables to the database. At the very bottom of the script, tell the `Pyinotify` code to run indefinitely, and this is ultimately meant to run as a daemon.

## Summary

This article covered some of the new features of `SQLAlchemy`, demonstrated how easy it is to get started, and how very simple the API is to use. You also built an incredibly powerful tool in under 100 lines of Python code, by using the elegance of `SQLAlchemy` and the hard work of an open-source library, `Pyinotify`. This is one of the features of a simple yet powerful ORM. It takes the mind-numbing complexity of dealing with relational databases, and makes it a joy, not a burden. That energy can then be devoted to spending time solving an interesting problem, as `SQLAlchemy` will be the easy part.

If you are interested in learning more about SQLAlchemy, you should read some of the [resources](#) listed at the end of this article. Listed there are an excellent book and a tremendous amount of outstanding online documentation. Finally, when you get the hang of things, you might consider exploring some of the other projects that use SQLAlchemy and extend it. One of the more interesting recent SQLAlchemy-related projects is the Website [reddit.com](#). It was built using the pure WSGI framework Pylons, which incorporates SQLAlchemy as its default ORM. I have included a link to the full source code for reddit. With your newfound knowledge of SQLAlchemy, you should be able to get your own reddit clone up and running quickly, and should be able to dig right into some of database queries. Good luck and have fun!

## Downloads

Description	Name	Size	Download method
Sample SQLAlchemy Code	sqlalchemy_code.zip	5KB	<a href="#">HTTP</a>

[Information about download methods](#)

# Resources

## Learn

- Visit the [SQLAlchemy Home Page](#) for more information.
- Get the latest documentation from [SQLAlchemy 0.5 API Reference](#).
- [Essential SQLAlchemy Book](#) guides Python developer using this code library.
- For an alternate shorthand configurational style, see [Declarative Extension Documentation](#)
- Learn how to create an [SQLAlchemy Session](#)
- See what Wikipedia has to say about [Spotlight](#)
- Learn about the [Beagle](#) search tool.
- [Pylesystem](#) uses SQLAlchemy and pyionotify to create a real time index of the filesystem.
- [Reddit Source Code](#)
- See how you can access database tables with [SQL Soup](#).
- [Elixir](#) lets you create simple Python classes that map directly to relational database tables. (this pattern
- [DBSprockets](#) give you the power to simply generate Web content from available database definitions

## Get products and technologies

- [IBM trial software](#): Build your next development project with software for download directly from developerWorks.

## Discuss

- [Podcasts](#): Tune in and catch up with IBM technical experts.
- [developerWorks blogs](#): Check out developerWorks blogs and get involved in the [developerWorks community](#).
- Participate in the AIX and UNIX forums:
  - [AIX Forum](#)
  - [AIX forum for developers](#)
  - [Cluster Systems Management](#)
  - [IBM Support Assistant Forum](#)

- [Performance Tools Forum](#)
- [Virtualization Forum](#)
- [More AIX and UNIX forums](#)

## About the author

### Noah Gift

Noah Gift is the co-author of "Python For Unix and Linux" by O'Reilly. He is an author, speaker, consultant, and community leader, writing for publications such as IBM Developerworks, Red Hat Magazine, O'Reilly, and MacTech. His consulting company's website is [www.giftcs.com](http://www.giftcs.com), and his personal website is [www.noahgift.com](http://www.noahgift.com). Noah is also the current organizer for [www.pyatl.org](http://www.pyatl.org), which is the Python User Group for Atlanta, GA.

He has a Master's degree in CIS from Cal State Los Angeles, B.S. in Nutritional Science from Cal Poly San Luis Obispo, is an Apple and LPI certified SysAdmin, and has worked at companies such as, Caltech, Disney Feature Animation, Sony Imageworks, and Turner Studios. In his free time he enjoys spending time with his wife Leah, and their son Liam, playing the piano, and exercising religiously.