Using Python, Django and MySQL in a Database Course

Thomas B. Gendreau
Computer Science Department
University of Wisconsin - La Crosse
La Crosse, WI 54601
gendreau@cs.uwlax.edu

Abstract

Software applications designed with a three-tiered architecture consisting of a web based user interface, a logic layer and a database back end are very common. A natural place to introduce this architecture into the undergraduate curriculum is in a Database Management Systems course. One of the challenges in doing this is to find an easily accessible environment in which students can develop applications. This paper describes such an environment using the programming language Python, a lightweight web server and framework called Django, and a MySQL database. This software is freely available and students can install all three pieces of software on their own machines. Students can quickly develop a simple application and use it as a foundation on which more elaborate features can be built. This paper describes these tools through the development of a simple web application.

1. Introduction

Software applications designed with a three-tiered architecture consisting of a web based user interface, a logic layer and a database back end are very common. A natural place to introduce this architecture into the undergraduate curriculum is in a Database Management Systems course. One of the challenges in doing this is to find an easily accessible environment in which students can develop applications. This paper describes such an environment using the programming language Python[1], a lightweight web server and framework called Django[2][3], and a MySQL[4] database. This software is freely available and students can install all three pieces of software on their own machines. Students can quickly develop a simple application and use it as a foundation on which more elaborate features can be built.

The paper is written as a simple tutorial. We assume the reader is familiar with SQL and that the reader has installed MySQL and Django. In the Linux platform on which most of the examples were developed the software was installed using the commands shown below.

```
> sudo apt-get install mysql-server python-mysqldb

> sudo apt-get install python-django
```

Section 2 gives a brief description of a database used through out the paper. Section 3 discusses MySQL and shows how to implement the example database in MySQL. Section 4 discusses Python. Only a few features of Python are used in this paper. The example used in Section 4 is a stand-alone program. Section 5 discusses Django and gives examples of Python in a web application environment.

2. Example Database

Throughout the following the features of Python, Django and MySQL will be illustrated using a simple database. The database includes three tables; faculty, course and section. The faculty table includes attributes fid and fname. Fid is the primary key of faculty. The course table includes attributes cnum, cname and credits. Cnum is the primary key of course. The section table includes snum, cnum, fid, semester, year, enrollment and capacity. Snum is the primary key of section. Cnum and fid are foreign keys in section. Cnum references course and fid references faculty. This is a very simple database but it is sufficient to illustrate many features of the software discussed in this paper and provides a foundation on which a more realistic student information system can be built. It is a simple example that students can implement in order to become familiar with the basic features of each software tool used in a database course.

3. MySQL

MySQL is a widely available relational database server. It is currently owned by Oracle Corporation. There are free and commercial versions that run on a variety of operating systems including Linux, Windows and Mac OS X. The examples discussed in this paper have been developed on Linux (ubuntu) and Mac OS X platforms. MySQL supports many features SQL features including stored procedures and transactions.

The following is a sequence of commands that creates the example database used in this paper. The sequence assumes the database server is running and the user has logged onto the server and is using the line-oriented interface (for brevity the output is not shown). The create table statements are shown in this example but when we discuss Django models will we see how a model can be used to create the tables.

```
mysql> create database sis;

mysql>use sis;

mysql>create table faculty(fid varchar(9) primary key, fname varchar(40));

mysql>create table course(chum varchar(9) primary key,
        cname varchar(40), credits int);

mysql> create table section(sum varchar(9) primary key,
          cnum varchar(9) references Course(chum), fid varchar(9)
          references Faculty(fid), semester varchar(6), year int, enrollment int,
          capacity int);
```

The first line of code creates a database. A database is a relatively lightweight object in MySQL. A single server can support many databases. In order to query or modify the content of a particular database a user must *use* or *connect* to the particular database. The create table statements are standard SQL statements to create tables.

Data can be inserted using SQL insert statements but MySQL also supports a command to load data from flat files. The following commands load data from files that are located on the same machine as the server (the common situation when this is used in class).

```
mysql> load data local infile 'faculty' into table faculty fields terminated by ':';

mysql> load data local infile 'course' into table course fields terminated by ':';

mysql> load data local infile 'section' into table section fields terminated by ':';
```

In these files the fields are separated by a colon. Users can choose other delimiters. An example line in the file section looks as follows.

```
s001:c001:f001:Fall:2011:10:20
```

MySQL supports SQL select statements and in the following we select all the rows from the course table and then quit MySQL (but the server remains running).

```
mysql> select * from course;

mysql> quit
```

4. Python

Python is a freely available interpreted object-oriented language. Students can quickly develop simple programs in Python but it can also be used to develop commercial scale software products. There are two main versions of Python used today: Python 2.? and Python 3.?. At this time Django only supports version 2 (2.5 or later) Python and the examples discussed in this paper use version 2. To use Python with MySQL the MySQLdb driver must be available, when students install MySQL on their machine they also install the MySQLdb driver. Python can be used to develop stand-alone applications as well as server side code in a web application.

The following program is an example of a stand-alone application that displays information found in a database. The program displays the course number and course name of courses worth 3 credits.

```
import MySQLdb
conn = MySQLdb.connect (host = "localhost",
                        user = "micsuser",
                        passwd = "micspass",
                        db = "sis")
cursor = conn.cursor ()
cursor.execute ("select cnum, cname  from course where credits = 3")
row = cursor.fetchone ()
while row != None:
        print "Course Number:", row[0],  "\tCourse Names", row[1]
        row = cursor.fetchone ()

cursor.close ()
conn.close ()
```

The program first creates a connection to the sis database accessible through a MySQL server running on the local machine. Notice that variables are not declared and the type of value referenced by a variable depends on the last value it was assigned. In this program conn references a connection object, cursor references a cursor object and row references a tuple object.

The call to connect creates a connection object. Given a connection object a cursor

object can be created. One method that can be invoked on a cursor object in execute. The execute method expects a string parameter that should be a legal SQL expression. In this example the SQL expression is hardcoded but it usually will be a string built based on user input. After the SQL expression is executed the program can process the results one row at a time using the fetchone method. The fetchone method returns a tuple. Parts of a tuple can be referenced using a syntax similar to array subscripting. In the above example row[0] refers to the cnum part of the tuple and row[1] refers to the cname part of the tuple. When no more tuples (rows) are available the call to fetchone returns None (which serves a similar purpose to null in Java).

Python uses indentation to show which statements are part of more complex statements. The first line of a statement that can include other statements such as loops, if statements, and function definitions ends with a colon. Statements that are included in the more complex statement are indented at least one space more than the statement they are part of. In this example the body of the while loop includes two statements: the print statement and the second assignment to row.

5. Django

Django is a freely available open source web framework and web server. It is implemented in Python and uses Python code in many of its features. In a production environment it is often used with Apache but it comes with a lightweight, easy to install web server. It is a good choice to use in an introductory course because students can install and develop simple applications quickly.

5.1 A First Simple Web Site

The first step in developing a Django application is to start a project. The following command starts a project called database1. As a result of starting a project a new directory is created called database1. That directory contains a number of files including: manage.py, settings.py and urls.py. Notice these files are examples of Python code and that is why they have an extension of .py. The django-admin program is also a Python program. The line of code was run in ubuntu so the .py extension was not needed but on other operating systems the command would be invoked by saying django-admin.py.

```
>django-admin startproject database1
```

Starting a project does not start the web server. Before we show how to start the Web server we want to edit some other files. The urls.py file contains url patterns that the application is willing to process. The pattern matching uses a simple regular expression format to describe urls that are acceptable and to direct the web server to execute a Python function when a request with a particular url arrives. The following is part of the content of the urls.py file used in our example.

```
from django.conf.urls.defaults import *
from database1.views import get_fname

urlpatterns = patterns('',
        (r'^facultyname/(.{3,9})/$', get_fname),
)
```

The first line imports the standard url functions. The second line imports the get_fname function from the file views.py that it is in the database1 project directory. Before we can use the application we will need to create the file views.py. The urlspattern variable references a Python sequence of patterns. The second pattern indicates that when a url whose second last field is facultyname and whose last field contains between three and nine characters arrives, the function get_fname should be called. Any function referenced in the urls patterns expects a request object as the first parameter. Other parameters can be passed to the function based on the matched patterns. In this case the parentheses around the pattern for the last field indicates that the matched string should be passed to get_fname. This example was tested on web server running on a machine with an IP address of 138.49.196.167 and a port number of 8080. Based on that a user could type the following url into a web browser and the web page will either display the name of the faculty member with faculty id 'f001' or it would display a message indicating that there is no faculty member with an id of 'f001'.

```
http://138.49.196.167:8080/facultyname/f001
```

Before this can happen we must implement the get_fname function in views.py. The following is part of the views.py file used in our example.

```
from django.http import HttpResponse
import MySQLdb

def get_fname(request, fid):
        conn = MySQLdb.connect (host = "localhost",
                        user = "micsuser",
                        passwd = "micspass",
                        db = "sis")
        cursor = conn.cursor ()
        cursor.execute ("select fname  from sis_faculty where  fid = '"+fid+"'")
        if cursor.rowcount == 0:
                html = "<html><body>There is no Faculty member with id  %s.
                                </body></html>"  % fid
        else:
                row = cursor.fetchone()
                html = "<html><body>The Faculty name is %s.</body></html>"
                                % row[0]
        return HttpResponse(html)
```

The function returns an HttpResponse object. MySQLdb is imported so the code has access to the database driver in order to connect to the MySQL database called sis. Notice that the body of the function is indented deeper than the first line of the function. The second parameter, fid, contains the string matched by the pattern **.{3,9}** in urls.py. This code is very similar to the earlier example Python code. The function rowcount can be applied to a cursor object and returns the number of rows return from the last call to execute. The table sis_faculty refers to the faculty table. This name will be explained below when we discuss models.

With a simple urls.py file and a views.py file we are ready to start the web server. The following line starts the web server. The server is willing to take requests from any IP address and the server listens on port 8080. If the 0.0.0.0 part is removed then the server will only accept requests from the machine on which the server is running.

```
> python manage.py runserver 0.0.0.0:8080
```

With just these few lines code a student now has a simple working web site.

5.2 Data Models

Part of the Django framework supports the development of a data model for the application through which the database can be created and accessed. In order to use the data model features an application must be created. An application is created within a project and a project can have zero to many applications that are part of it. The following line creates a new application called sis.

```
> python manage.py startapp sis
```

The line should be executed in the database1 directory. It creates a new directory called sis. The files in the sis directory include models.py and views.py. The models.py will be used to define the data model using Python class definitions and data types defined by Django. The following is the data model created for our example.

```
from django.db import models
class faculty(models.Model):
        fid = models.CharField(max_length=9, primary_key=True)
        fname = models.CharField(max_length=40)

        def __unicode__(self):
                return u'%s, %s' % (self.fid, self.fname)

class course(models.Model):
        cnum = models.CharField(max_length=9, primary_key=True)
        cname = models.CharField(max_length=40)
        credits = models.IntegerField()
```

```
        def __unicode__(self):
                return u'%s, %s, %s' % (self.cnum, self.cname, self.credits)

class section(models.Model):
        snum = models.CharField(max_length=9, primary_key=True)
        cnum = models.ForeignKey(course, db_column='cnum')
        fid = models.ForeignKey(faculty, db_column='fid')
        semester = models.CharField(max_length=6)
        year = models.IntegerField()
        enrollment = models.IntegerField()
        capacity = models.IntegerField()

        def __unicode__(self):
                return u'%s, %s' % (self.snum, self.enrollment)
```

The data model includes one class for each entity of our database. Since both relationships in the database are one-to-many (represented by the two foreign key specifications in section), the model will be used to create three tables in the database. It is also possible to specify many-to-many relationships in the model. For each many-to-many relationship in the model an additional table will be created when the model is synchronized with the database. Django requires each class in the model to implement the __unicode__ function that returns the unicode representation of an instance of the class. There is great flexibility is the exact unicode representation.

In order to use the model with a particular database the file settings.py must be modified. The first modification is to the database related variables in the file. Shown below is part of the settings.py file related to database settings. The values have been set work with the MySQL database used in this example..

```
DATABASE_ENGINE = 'mysql'
DATABASE_NAME = 'sis'
DATABASE_USER = 'micsuser'
DATABASE_PASSWORD = 'micspass'
```

These settings require the database server and the web server to be located on the same machine and that the database *sis* has been created before using the model. Other variables can be set in the case where the database server and the web server are located on different machines.

The second part of the settings.py file that must be changed is shown below.

```
INSTALLED_APPS = (
        'database1.sis',
)
```

The variable INSTALLED_APPS is a sequence of installed apps. Python expects the trailing comma in sequences with a single element (and it can be included in any sequence with one or more items). In this example we have only one installed app but remember a project can have multiple apps.

With the settings.py file properly modified we can use the model we created above. Shown below is a command that can be used to validate the model.

```
>python manage.py validate
```

This checks the syntax of the model and also indirectly checks that the settings.py file has be properly modified. It is usually a good idea to do this before synchronizing the model with the database.

The command shown below can be used to generate the create table statements that would be executed when the model is synchronized with the database. This command does not modify the database but allows the user to examine the database modifications that would take place when the model is synchronized with the database. It also shows any new fields that would be created when synchronization occurs. For example, in the way we defined the model above the primary keys for the tables are explicitly defined. Another option is to have the model synchronization process add a primary key field to each table.

```
>python manage.py sqlall sis
```

To synchronize the model with the database execute the command shown below. Executing the command will cause the database to be modified. The output of the command indicates the tables and indexes that are created in the database.

```
>python manage.py syncdb
```

The one difference between the create table statements that are created by this process and the statements shown in Section 3 is that the since the model is part of the sis app the tables will be named sis_faculty, sis_course and sis_section. Rows can be added by creating and saving new instances of faculty, course or section but data can also be added to the tables by directly using MySQL insert or load statements. In testing our example we ran load statements like those shown in Section 3 (with the table names modified appropriately such as to sis_faculty)

5.3 Templates

The function get_fname shown in Section 5.1 includes explicit html code. For simple pages that is fine but most of the time it is better to separate the html code from the Python code. Django allows this be done with the use of templates. In the database1 directory we created a templates directory. This directory will store all the html files used

in the example. In order to tell the project where the html template files can be found there is one more modification that must be made to the settings.py file. That modification is shown below.

```
TEMPLATE_DIRS = (
    # Don't forget to use absolute paths, not relative paths.
    '/home/ubuntu/database1/templates',
)
```

In our example all the html template files are stored in one directory but in general there can be multiple directories used to store the html files. For example each app could have its own directory for html template files. This code also shows use of the pound sign to indicate comments in Python code.

The function get_cname makes use of templates. The function is stored in the same views,py file where get_fname was stored (the complete views.py file is shown in Section 5.4).

```
def get_cname(request, cnum):
    conn = MySQLdb.connect (host = "localhost",
                    user = "root",
                    passwd = "mysqlroot",
                    db = "sis")
    cursor = conn.cursor ()
    cursor.execute ("select cname,credits  from sis_course where
                        cnum = '"+cnum+"'")
    if cursor.rowcount == 0:
        html = "<html><body>There is no Course with number  %s.
                    </body></html>"  % cnum
    else:
        row = cursor.fetchone()
        t = get_template('cname.html')
        c = Context({'cnum' : cnum, 'cname' : row[0],
                                        'credits' : row[1]})
        html = t.render(c)
    return HttpResponse(html)
```

Explicit html code and templates can be used in the same function. In get_cname the error message is produced with explicit html and the results of a successful search are produced with a template. The call get_template retrieves the file cname.html from the templates directory. The contents of cname.html are shown below.

```
<html>
<head>
<title>Course Information</title>
</head>
```

```
<body>
Course Number {{cnum}}<br>
Course Name {{cname}}<br>
Credit Hours {{credits}}<br>
</body>
</html>
```

The values in double braces are variables that can be given values. One way to assign the variables values is through a Context. A Context includes a mapping between variable names and values. The bold face line in the above example creates a new Context object that maps values to the variables cnum, cname, and credits (exactly those variables used in the template). The function render expects a Context object and uses it to create an html string from a template with values replacing variables.

5.4 The Complete Example

The complete example allows users to query the database to find the name of a faculty member given a faculty id, to find the name and credits of a course given a course number, to find the details of a section given a section number and to search for sections where the enrollment is less than the capacity. The files that were created include the urls.py and views.py in the database1 directory; cname.html, section.html, search_form.html and search_result.html in the database1/templates directory; and models.py and views.py in the database1/sis directory. The content of the files are shown below. Readers should be able to copy the content to implement the simple application we have been discussing. Most of the details have been explained by earlier examples. When additional explanation is required it is provide below.

The *urls.py* file contains the following content. The patterns used are similar to the pattern used for the get_fname example. The functions get_section and search are found in the views.py file in the sis app directory. The other functions are stored in the database1 project directory.

```
from django.conf.urls.defaults import *
from database1.views import get_fname, get_cname
from database1.sis.views import get_section, search

urlpatterns = patterns('',
        (r'^facultyname/(.{3,9})/$', get_fname),
        (r'^coursename/(.{3,9})/$', get_cname),
        (r'^sectioninfo/(.{3,9})/$', get_section),
        (r'^searchSection/$', search),
)
```

The *database1/views.py* file is shown below. Both functions have been previously discussed. The additional import lines are required to use templates and Contexts.

```
from django.template.loader import get_template
from django.template import Context
from django.http import HttpResponse
import MySQLdb

def get_fname(request, fid):
        conn = MySQLdb.connect (host = "localhost",
                        user = "micsuser",
                        passwd = "micspass",
                        db = "sis")
        cursor = conn.cursor ()
        cursor.execute ("select fname  from sis_faculty where  fid = '"+fid+"'")
        if cursor.rowcount == 0:
                html = "<html><body>There is no Faculty member with id  %s.
                        </body></html>"  % fid
        else:
                row = cursor.fetchone()
                html = "<html><body>The Faculty name is %s.
                        </body></html>" % row[0]
        return HttpResponse(html)

def get_cname(request, cnum):
        conn = MySQLdb.connect (host = "localhost",
                        user = "micsuser",
                        passwd = "micspass",
                        db = "sis")
        cursor = conn.cursor ()
        cursor.execute ("select cname,credits  from sis_course where
                                cnum = '"+cnum+"'")
        if cursor.rowcount == 0:
                html = "<html><body>There is no Course with number  %s.
                        </body></html>"  % cnum
        else:
                row = cursor.fetchone()
                t = get_template('cname.html')
                c = Context({'cnum' : cnum, 'cname' : row[0], 'credits' : row[1]})
                html = t.render(c)
        return HttpResponse(html)
```

The *database1/sis/views.py* file contains the functions get_section and search. There are a couple of new features in these functions. First instead of using SQL select statements to access the database, the database is accessed through the data model code. The two bold face statements shown below are examples of querying the database through the data model. The statements return QuerySets. The expressions in the filter part of the statement are similar to where clause conditions in a select statement. These filters are

not as flexible as a general select statement but for simple queries they are convenient to use. The first query finds rows in the section table where the section number matches the sid value passed to the function from the url used to access the web site. The second query finds rows in the section table that match the course number passed in from a search form and that have an enrollment less than the capacity.

The second new feature is the use of the render_to_response function. The function combines the work of get_templates and render used in the get_cname function. The parameters are a template and a Context.

```
from django.template.loader import get_template
from django.template import Context
from django.http import HttpResponse
from django.shortcuts import render_to_response
from django.db.models import Q, F
from models import section

def get_section(request, sid):
        result = section.objects.filter(snum=sid)
        if result.count() == 0:
                html = "<html><body>There is no Section with number  %s.
                        </body></html>"  % sid
        else:
                t = get_template('section.html')
                w = result.values()
                c = Context(w[0])
                html = t.render(c)
        return HttpResponse(html)

def search(request):
error = False
if 'cnum' in request.GET:
cid = request.GET['cnum']
if not cid:
        error = True
else:
        sections = section.objects.filter(cnum__exact=cid).filter
                        (enrollment__lt=F('capacity'))
        return render_to_response('search_results.html',
                                                {'sections': sections, 'query': cid})
return render_to_response('search_form.html', {'error': error})
```

The cname.html template was shown earlier. The other templates used in this example are shown below. The search_form.html and search_result.html were created from the examples in chapter seven of [3] with modifications for our faculty-course-section example.

The section.html template is shown below. The variables cnum_id and fid_id refer to the foreign key fields in section. The data model appends the suffix _id to the foreign key fields as they are defined in the data model.

```
<html>
<head>
<title>Section Information</title>
</head>
<body>
Section {{snum}} <br>
Course {{cnum_id}}<br>
Faculty {{fid_id}}<br>
Semester {{semester}}<br>
Year {{year}}<br>
Enrollment {{enrollment}}<br>
Capacity {{capacity}}<br>
</body>
</html>
```

The search_form.html template is shown below. In response to a user entering http://138.49.196.167:8080/searchSection in the browser the web server returns a page with a field in which a course number can be entered and a search button to be clicked.

```
<html>
<head>
   <title>Search Open Sections</title>
</head>
<body>
    {% if error %}
      <p style="color: red;">Please submit a course number.</p>
   {% endif %}
    <form action="" method="get">
       <input type="text" name="cnum">
       <input type="submit" value="Search">
    </form>
</body>
</html>
```

The search_result.html template is shown below. This template is returned in response to the search button being click on the search form.

```
<html>
<head>
   <title>Open Sections</title>
</head>
```

```
<body>
<p>You searched for open sections of: <strong>{{ query }}</strong></p>

{% if sections %}
   <p>Found {{ sections|length }} section{{ sections|pluralize }}.</p>
   <ul>
      {% for section in sections %}
      <li>{{ section.snum}}</li>
      {% endfor %}
   </ul>
{% else %}
   <p>No open sections found.</p>
{% endif %}
</body>
</html>
```

6. Conclusion

A simple example of a three-tiered application is illustrated using MySQL, Python and Django. All the code required to implement the example is provide. Students and instructors can use the example to learn how to develop a simple web application.

7. References

[1] www.python.org
[2] www.djangoproject.com
[3] www.djangobook.com/en/2.0/ (also known as *The Definitive Guide to Django: Web Development Done Right* by Adrian Holovaty and Jacob Kaplan-Moss. Published by Apress)
[4] mysql.com