CHAPTER 26

Project 7: Your Own Bulletin Board

Many kinds of software enable you to communicate with other people over the Internet. You've seen a few already (for example, the Usenet groups in Chapter 23 and the chat server in Chapter 24). In this chapter, you will implement another such system: a web-based discussion forum.

What's the Problem?

In this project, you create a simple system for posting and responding to messages via the Web. This has utility in itself, as a discussion forum. One famous example of such a forum is Slashdot (`http://slashdot.org`). The system developed in this chapter is quite simple, but the basic functionality is there, and it should be capable of handling quite a large number of postings.

However, the material covered in this chapter has uses beyond developing stand-alone discussion forums. It could be used to implement a more general system for collaboration, for example, or an issue-tracking system, a blog with commenting functionality, or something completely different. The combination of CGI (or similar technologies) and a solid database (in this case, a SQL database) is quite powerful and versatile.

Tip Even though it's fun and educational to write your own software, in many cases, it's more cost-effective to search for existing software. In the case of discussion forums and the like, chances are that you can find quite a few well-developed systems freely available. Also, most web application frameworks, such as Django, Zope, and TurboGears (mentioned in Chapter 15), have built-in support for this sort of functionality.

Specifically, the final system should meet the following requirements:

- It should display the subjects of all current messages.
- It should support message threading (displaying replies indented under the message they reply to).
- You should be able to view existing messages.
- You should be able to reply to existing messages.

In addition to these functional requirements, it would be nice if the system were reasonably stable, could handle a large number of messages, and avoided such problems as two users writing to the same file at the same time. The desired robustness can be achieved by using a database server of some sort, instead of writing the file-handling code yourself.

Useful Tools

In addition to the CGI stuff from Chapter 15, you'll need a SQL database, as discussed in Chapter 13. You could either use the stand-alone database SQLite, which is used in that chapter, or you could use some other system, such as either of the following two excellent, freely available databases:

- PostgreSQL (`http://www.postgresql.org`)
- MySQL (`http://www.mysql.org`)

In this chapter, I use PostgreSQL for the examples, but the code should work with most SQL databases (including MySQL or SQLite) with few edits.

Before moving on, you should make sure that you have access to a SQL database server (or a stand-alone SQL database, such as SQLite) and check its documentation for instructions on how to manage it.

In addition to the database server itself, you'll need a Python module that can interface with the server (and hide the details from you). Most such modules support the Python DB API, which is discussed in more detail in Chapter 13. In this chapter, I use `psycopg` (`http://initd.org/Software/psycopg`), a robust front end for PostgreSQL. If you're using MySQL, the `MySQLdb` module (`http://sourceforge.net/projects/mysql-python`) is a good choice.

After you have installed your database module, you should be able to import it (for example, with `import psycopg` or `import MySQLdb`) without raising any exceptions.

Preparations

Before your program can start using your database, you must actually create the database. That is done using SQL (see Chapter 13 for some pointers).

The database structure is intimately linked with the problem and can be a bit tricky to change once you've created it and populated it with data (messages). Let's keep it simple.

You'll have only one table, which will contain one row for each message. Each message will have a unique ID (an integer), a subject, a sender (or poster), and some text (the body).

In addition, because you want to be able to display the messages hierarchically (threading), each message should store a reference to the message it is a reply to. The resulting `CREATE TABLE` SQL command is shown in Listing 26-1.

Listing 26-1. *Creating the Database in PostgreSQL*

```
CREATE TABLE messages (
    id          SERIAL PRIMARY KEY,
    subject     TEXT NOT NULL,
    sender      TEXT NOT NULL,
    reply_to    INTEGER REFERENCES messages,
    text        TEXT NOT NULL
);
```

Note that this command uses some PostgreSQL-specific features (`SERIAL`, which ensures that each message automatically receives a unique ID; the `TEXT` data type; and `REFERENCES`, which makes sure that `reply_to` contains a valid message ID). A more MySQL-friendly version is shown in Listing 26-2.

Listing 26-2. *Creating the Database in MySQL*

```
CREATE TABLE messages (
    id          INT NOT NULL AUTO_INCREMENT,
    subject     VARCHAR(100) NOT NULL,
    sender      VARCHAR(15) NOT NULL,
    reply_to    INT,
    text        MEDIUMTEXT NOT NULL,
    PRIMARY KEY(id)
);
```

Finally, for those of you using SQLite, there's a schema in Listing 26-3.

Listing 26-3. *Creating the Database in SQLite*

```
create table messages (
    id          integer primary key autoincrement,
    subject     text not null,
    sender      text not null,
    reply_to    int,
    text        text not null
);
```

I've kept these code snippets simple (a SQL guru would certainly find ways to improve them) because the focus of this chapter is, after all, the Python code. The SQL statements create a new table with the following five fields (columns):

`id`: Used to identify the individual messages. Each message automatically receives a unique ID by the database manager, so you don't need to worry about assigning those from your Python code.

`subject`: A string that contains the subject of the message.

`sender`: A string that contains the sender's name or email address or something like that.

`reply_to`: If the message is a reply to another message, this field contains the `id` of the other message. (Otherwise, the field won't contain anything.)

`text`: A string that contains the body of the message.

When you've created this database and set the permissions on it so that your web server is allowed to read its contents and insert new rows, you're ready to start coding the CGI.

First Implementation

In this project, the first prototype will be very limited. It will be a single script that uses the database functionality so that you can get a feel for how it works. Once you have that pegged, writing the other necessary scripts won't be very hard. In many ways, this is just a short reminder of the material covered in Chapter 13.

The CGI part of the code is very similar to that in Chapter 25. If you haven't read that chapter yet, you might want to take a look at it. You should also be sure to review the section "CGI Security Risks" in Chapter 15.

■ **Caution** In the CGI scripts in this chapter, I've imported and enabled the `cgitb` module. This is very useful to uncover flaws in your code, but you should probably remove the call to `cgitb.enable` before deploying the software—you probably wouldn't want an ordinary user to face a full `cgitb` traceback.

The first thing you need to know is how the Python DB API works. If you haven't read Chapter 13, you probably should at least skim through it now. If you would rather just press on, here is the core functionality again (replace `db` with the name of your database module—for example, `psycopg` or `MySQLdb`):

`conn = db.connect('user=foo dbname=bar')`: Connects to the database named `bar` as user `foo` and assigns the returned connection object to `conn`. (Note that the parameter to connect is a string.)

■ **Caution** In this project, I assume that you have a dedicated machine on which the database and web server run. The given user (`foo`) should be allowed to connect only from that machine to avoid unwanted access. If you have other users on your machine, you should probably protect your database with a password, which may also be supplied in the parameter string to `connect`. To find out more about this, consult the documentation for your database (and your Python database module).

`curs = conn.cursor()`: Gets a *cursor* object from the connection object. The cursor is used to actually execute SQL statements and fetch the results.

`conn.commit()`: Commits the changes caused by the SQL statements since the last commit.

`conn.close()`: Closes the connection.

`curs.execute(sql_string)`: Executes a SQL statement.

`curs.fetchone()`: Fetches one result row as a sequence—for example, a tuple.

`curs.dictfetchone()`: Fetches one result row as a dictionary. (Not part of the standard, and therefore not available in all modules.)

`curs.fetchall()`: Fetches all result rows as a sequence of sequences—for example, a list of tuples.

`curs.dictfetchall()`: Fetches all result rows as a sequence (for example, a list) of dictionaries. (Not part of the standard, and therefore not available in all modules.)

Here is a simple test (assuming `psycopg`)—retrieving all the messages in the database (which is currently empty, so you won't get any):

```
>>> import psycopg
>>> conn = psycopg.connect('user=foo dbname=bar')
>>> curs = conn.cursor()
>>> curs.execute('SELECT * FROM messages')
>>> curs.fetchall()
[]
```

Because you haven't implemented the web interface yet, you must enter messages manually if you want to test the database. You can do that either through an administrative tool (such as `mysql` for MySQL or `psql` for PostgreSQL), or you can use the Python interpreter with your database module.

Here is a useful piece of code you can use for testing purposes:

```
#!/usr/bin/env python
# addmessage.py
```

```
import psycopg
conn = psycopg.connect('user=foo dbname=bar')
curs = conn.cursor()

reply_to = raw_input('Reply to: ')
subject = raw_input('Subject: ')
sender = raw_input('Sender: ')
text = raw_input('Text: ')

if reply_to:
    query = """
    INSERT INTO messages(reply_to, sender, subject, text)
    VALUES(%s, '%s', '%s', '%s')""" % (reply_to, sender, subject, text)
else:
    query = """
    INSERT INTO messages(sender, subject, text)
    VALUES('%s', '%s', '%s')""" % (sender, subject, text)

curs.execute(query)
conn.commit()
```

Note that this code is a bit crude. It doesn't keep track of IDs for you (you'll have to make sure that what you enter as `reply_to`, if anything, is a valid ID), and it doesn't deal properly with text containing single quotes (this can be problematic because single quotes are used as string delimiters in SQL). These issues will be dealt with in the final system, of course.

Try to add a few messages and examine the database at the interactive Python prompt. If everything seems okay, it's time to write a CGI script that accesses the database.

Now that you have the database-handling code figured out and some ready-made CGI code you can pinch from Chapter 25, writing a script for viewing the message subjects (a simple version of the "main page" of the forum) shouldn't be too hard. You must do the standard CGI setup (in this case, mainly printing the `Content-type` string), do the standard database setup (get a connection and a cursor), execute a simple SQL `select` command to get all the messages, and then retrieve the resulting rows with `curs.fetchall` or `curs.dictfetchall`.

Listing 26-4 shows a script that does these things. The only really new stuff in the listing is the formatting code, which is used to get the threaded look where replies are displayed below and to the right of the messages they are replies to.

It basically works like this:

- For each message, get the `reply_to` field. If it is `None` (not a reply), add the message to the list of top-level messages. Otherwise, append the message to the list of children kept in `children[parent_id]`.
- For each top-level message, call `format`. The `format` function prints the subject of the message. Also, if the message has any children, it opens a `blockquote` element (HTML), calls `format` (recursively) for each child, and ends the `blockquote` element.

If you open the script in your web browser (see Chapter 15 for information about how to run CGI scripts), you should see a threaded view of all the messages you've added (or their subjects, anyway).

For an idea of what the bulletin board looks like, see Figure 26-1 later in this chapter.

■ **Note** If you're using SQLite, you can't use `dictfetchall`, as in Listing 26-4. The line `rows = curs.dictfetchall()` can be replaced with the following snippet:

```
names = [d[0] for d in curs.description]
rows = [dict(zip(names, row)) for row in curs.fetchall()]
```

Listing 26-4. *The Main Bulletin Board (simple_main.cgi)*

```
#!/usr/bin/python

print 'Content-type: text/html\n'

import cgitb; cgitb.enable()

import psycopg
conn = psycopg.connect('dbname=foo user=bar')
curs = conn.cursor()

print """
<html>
  <head>
    <title>The FooBar Bulletin Board</title>
  </head>
  <body>
    <h1>The FooBar Bulletin Board</h1>
    """


curs.execute('SELECT * FROM messages')
rows = curs.dictfetchall()

toplevel = []
children = {}

for row in rows:
    parent_id = row['reply_to']
    if parent_id is None:
        toplevel.append(row)
    else:
        children.setdefault(parent_id,[]).append(row)
```

```
def format(row):
    print row['subject']
    try: kids = children[row['id']]
    except KeyError: pass
    else:
        print '<blockquote>'
        for kid in kids:
            format(kid)
        print '</blockquote>'

print '<p>'

for row in toplevel:
    format(row)

print """
    </p>
  </body>
</html>
"""
```

■ **Note** If, for some reason, you can't get the program to work, it may be that you haven't set up your database properly. Consult the documentation for your database to see what is needed in order to let a given user connect and to modify the database. You may, for example, need to list the IP address of the connecting machine explicitly.

Second Implementation

The first implementation was quite limited in that it didn't even allow users to post messages. In this section, you expand on the simple system in the first prototype, which contains the basic structure for the final version. Some measures will be added to check the supplied parameters (such as checking whether `reply_to` is really a number and whether the required parameters are really supplied), but you should note that making a system like this robust and user-friendly is a tough task. If you intend to use the system (or, I hope, an improved version of your own), you should be prepared to work quite a bit on these issues.

But before you can even think of improving stability, you need something that works, right? So, where do you begin? How do you structure the system?

A simple way of structuring web programs (using technologies such as CGI) is to have one script per action performed by the user. In the case of this system, that would mean the following scripts:

`main.cgi`: Displays the subjects of all messages (threaded) with links to the articles themselves.

`view.cgi`: Displays a single article and contains a link that will let you reply to it.

`edit.cgi`: Displays a single article in editable form (with text fields and text areas, just as in Chapter 25). Its Submit button is linked to the save script.

`save.cgi`: Receives information about an article (from `edit.cgi`) and saves it by inserting a new row into the database table.

Let's deal with these separately.

Writing the Main Script

The `main.cgi` script is very similar to the `simple_main.cgi` script from the first prototype. The main difference is the addition of links. Each subject will be a link to a given message (to `view.cgi`), and at the bottom of the page, you'll add a link that allows the user to post a new message (a link to `edit.cgi`).

Take a look at the code in Listing 26-5. The line containing the link to each article (part of the `format` function) looks like this:

```
print '<p><a href="view.cgi?id=%(id)i">%(subject)s</a></p>' % row
```

Basically, it creates a link to `view.cgi?id=someid` where `someid` is the `id` of the given row. This syntax (the question mark and `key=val`) is simply a way of passing parameters to a CGI script. That means if users click this link, they are taken to `view.cgi` with the `id` parameter properly set. The "Post message" link is just a link to `edit.cgi`.

Listing 26-5. *The Main Bulletin Board (main.cgi)*

```
#!/usr/bin/python

print 'Content-type: text/html\n'

import cgitb; cgitb.enable()

import psycopg
conn = psycopg.connect('dbname=foo user=bar')
curs = conn.cursor()

print """
<html>
  <head>
    <title>The FooBar Bulletin Board</title>
  </head>
  <body>
    <h1>The FooBar Bulletin Board</h1>
    """

curs.execute('SELECT * FROM messages')
rows = curs.dictfetchall()
```

```
toplevel = []
children = {}

for row in rows:
    parent_id = row['reply_to']
    if parent_id is None:
        toplevel.append(row)
    else:
        children.setdefault(parent_id,[]).append(row)

def format(row):
    print '<p><a href="view.cgi?id=%(id)i">%(subject)s</a></p>' % row
    try: kids = children[row['id']]
    except KeyError: pass
    else:
        print '<blockquote>'
        for kid in kids:
            format(kid)
        print '</blockquote>'

print '<p>'

for row in toplevel:
    format(row)

print """
    </p>
    <hr />
    <p><a href="edit.cgi">Post message</a></p>
  </body>
</html>
"""
```

So, let's see how `view.cgi` handles the `id` parameter.

Writing the View Script

The `view.cgi` script uses the supplied CGI parameter `id` to retrieve a single message from the database. It then formats a simple HTML page with the resulting values. This page also contains a link back to the main page (`main.cgi`) and, perhaps more interestingly, to `edit.cgi`, but this time with the `reply_to` parameter set to `id` to ensure that the new message will be a reply to the current one. See Listing 26-6 for the code of `view.cgi`.

Listing 26-6. *The Message Viewer (view.cgi)*

```
#!/usr/bin/python

print 'Content-type: text/html\n'

import cgitb; cgitb.enable()

import psycopg
conn = psycopg.connect('dbname=foo user=bar')
curs = conn.cursor()

import cgi, sys
form = cgi.FieldStorage()
id = form.getvalue('id')

print """
<html>
  <head>
    <title>View Message</title>
  </head>
  <body>
    <h1>View Message</h1>
    """

try: id = int(id)
except:
    print 'Invalid message ID'
    sys.exit()

curs.execute('SELECT * FROM messages WHERE id = %i' % id)
rows = curs.dictfetchall()

if not rows:
    print 'Unknown message ID'
    sys.exit()

row = rows[0]

print """
    <p><b>Subject:</b> %(subject)s<br />
    <b>Sender:</b> %(sender)s<br />
    <pre>%(text)s</pre>
    </p>
```

```
    <hr />
    <a href='main.cgi'>Back to the main page</a>
    | <a href="edit.cgi?reply_to=%(id)s">Reply</a>
  </body>
</html>
""" % row
```

Writing the Edit Script

The `edit.cgi` script actually performs a dual function: it is used to edit new messages and also to edit replies. The difference isn't all that great: if a `reply_to` is supplied in the CGI request, it is kept in a *hidden input* in the edit form. Also, the subject is set to `"Re: parentsubject"` by default (unless the subject already begins with `"Re:"`—you don't want to keep adding those). Here is the code snippet that takes care of these details:

```
subject = ''
if reply_to is not None:
    print '<input type="hidden" name="reply_to" value="%s"/>' % reply_to
    curs.execute('SELECT subject FROM messages WHERE id = %s' % reply_to)
    subject = curs.fetchone()[0]
    if not subject.startswith('Re: '):
        subject = 'Re: ' + subject
```

■ **Note** Hidden inputs are used to temporarily store information in a web form. They don't show up to the user as text areas and the like do, but their value is still passed to the CGI script that is the action of the form. That way, the script that generates the form can pass information to the script that will eventually process the same form.

Listing 26-7 shows the source code for the `edit.cgi` script.

Listing 26-7. *The Message Editor (edit.cgi)*

```
#!/usr/bin/python

print 'Content-type: text/html\n'

import cgitb; cgitb.enable()


import psycopg
conn = psycopg.connect('dbname=foo user=bar')
curs = conn.cursor()
```

```
import cgi, sys
form = cgi.FieldStorage()
reply_to = form.getvalue('reply_to')

print """
<html>
  <head>
    <title>Compose Message</title>
  </head>
  <body>
    <h1>Compose Message</h1>

    <form action='save.cgi' method='POST'>
    """

subject = ''
if reply_to is not None:
    print '<input type="hidden" name="reply_to" value="%s"/>' % reply_to
    curs.execute('SELECT subject FROM messages WHERE id = %s' % reply_to)
    subject = curs.fetchone()[0]
    if not subject.startswith('Re: '):
        subject = 'Re: ' + subject

print """
    <b>Subject:</b><br />
    <input type='text' size='40' name='subject' value='%s' /><br />
    <b>Sender:</b><br />
    <input type='text' size='40' name='sender' /><br />
    <b>Message:</b><br />
    <textarea name='text' cols='40' rows='20'></textarea><br />
    <input type='submit' value='Save'/>
    </form>
    <hr />
    <a href='main.cgi'>Back to the main page</a>'
  </body>
</html>
""" % subject
```

Writing the Save Script

Now let's move on to the final script. The `save.cgi` script will receive information about a message (from the form generated by `edit.cgi`) and will store it in the database. That means using a SQL `INSERT` command, and because the database has been modified, `conn.commit` must be called so the changes aren't lost when the script terminates.

Listing 26-8 shows the source code for the `save.cgi` script.

Listing 26-8. *The Save Script (save.cgi)*

```
#!/usr/bin/python

print 'Content-type: text/html\n'

import cgitb; cgitb.enable()

def quote(string):
    if string:
        return string.replace("'", "\\'")
    else:
        return string

import psycopg
conn = psycopg.connect('dbname=foo user=bar')
curs = conn.cursor()

import cgi, sys
form = cgi.FieldStorage()

sender = quote(form.getvalue('sender'))
subject = quote(form.getvalue('subject'))
text = quote(form.getvalue('text'))
reply_to = form.getvalue('reply_to')

if not (sender and subject and text):
    print 'Please supply sender, subject, and text'
    sys.exit()

if reply_to is not None:
    query = """
    INSERT INTO messages(reply_to, sender, subject, text)
    VALUES(%i, '%s', '%s', '%s')""" % (int(reply_to), sender, subject, text)
else:
    query = """
    INSERT INTO messages(sender, subject, text)
    VALUES('%s', '%s', '%s')""" % (sender, subject, text)

curs.execute(query)
conn.commit()


print """
<html>
```

```
  <head>
    <title>Message Saved</title>
  </head>
  <body>
    <h1>Message Saved</h1>
    <hr />
    <a href='main.cgi'>Back to the main page</a>
  </body>
</html>s
"""
```

Trying It Out

To test this system, start by opening `main.cgi`. From there, click the Post message link. That should take you to `edit.cgi`. Enter some values in all the fields and click the Save link. That should take you to `save.cgi`, which will display the message Message Saved. Click the Back to the main page link to get back to `main.cgi`. The listing should now include your new message.

To view your message, simply click its subject. You should go to `view.cgi` with the correct ID. From there, try to click the Reply link, which should take you to `edit.cgi` once again, but this time with `reply_to` set (in a hidden input tag) and with a default subject. Once again, enter some text, click Save, and go back to the main page. It should now show your reply, displayed under the original subject. (If it's not showing, try to reload the page.)

The main page is shown in Figure 26-1, the message viewer in Figure 26-2, and the message composer in Figure 26-3.

Figure 26-1. *The main page*

Figure 26-2. *The message viewer*

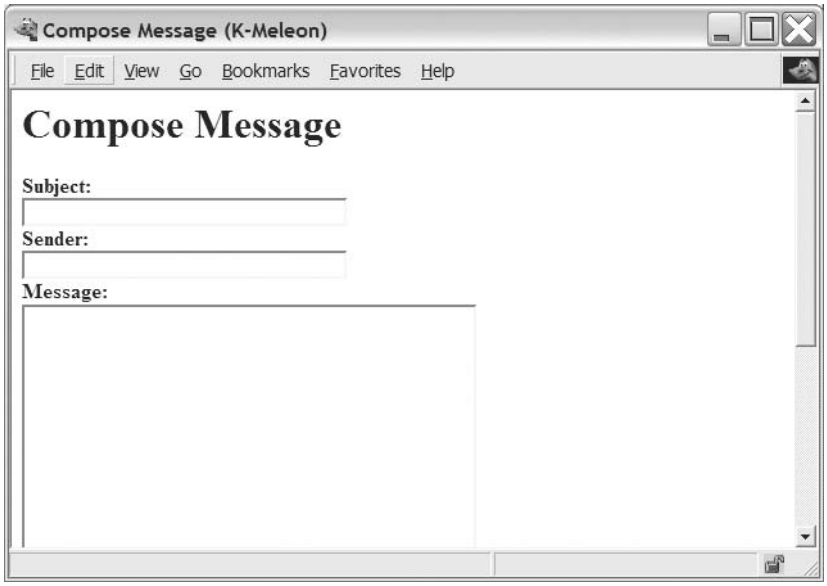

Figure 26-3. *The message composer*

Further Exploration

Now that you have the power to develop huge and powerful web applications with reliable and efficient storage, there are many things you can sink your teeth into:

- How about making a web front end to a database of your favorite Monty Python sketches?
- If you're interested in improving the system in this chapter, you should think about abstraction. How about creating a utility module with a function to print a standard header and another to print a standard footer? That way, you wouldn't need to write the same HTML stuff in each script. Also, it might be useful to add a user database with some password handling or abstract away the code for creating a connection.
- If you would like a storage solution that doesn't require a dedicated server, you could use SQLite (which is used in Chapter 13), or you might want to check out Metakit, a really neat little database package that also lets you store an entire database in a single file (`http://equi4.com/metakit/python.html`).
- Yet another alternative is the Berkeley DB (`http://www.sleepycat.com`), which is quite simple but can handle astonishing amounts of data very efficiently. (The Berkeley DB is accessible, when installed, through the standard library modules `bsddb`, `dbhash`, and `anydbm`.)

What Now?

If you think writing your own discussion forum software is cool, how about writing your own peer-to-peer file sharing program, like BitTorrent (or, at least, its lobotomized half brother)? Well, in the next project, that's exactly what you'll do. And the good news is that it will be easier than most of the network programming you've done so far, thanks to the wonder of remote procedure calls.