The Flask Mega-Tutorial

index of all the articles in the series that have been published to date:

- Part I: Hello, World!
- Part II: Templates
- Part III: Web Forms
- Part IV: Database
- Part V: User Logins
- Part VI: Profile Page And Avatars
- Part VII: Unit Testing
- Part VIII: Followers, Contacts And Friends
- Part IX: Pagination
- Part X: Full Text Search
- Part XI: Email Support
- Part XII: Facelift
- Part XIII: Dates and Times
- Part XIV: I18n and L10n
- Part XV: Ajax
- Part XVI: Debugging, Testing and Profiling
- Part XVII: Deployment on Linux (even on the Raspberry Pi!)
- Part XVIII: Deployment on the Heroku Cloud

Part I: Hello, World!

My background

I'm a software engineer with double digit years of experience developing complex applications in several languages. I first learned Python as part of an effort to create bindings for a C++ library at work.

In addition to Python, I've written web apps in PHP, Ruby, Smalltalk and believe it or not, also in C++. Of all these, the Python/Flask combination is the one that I've found to be the most flexible.

UPDATE: I have written a book titled "Flask Web Development", published in 2014 by O'Reilly Media. The book and the tutorial complement each other, the book presents a more updated usage of Flask and is, in general, more advanced than the tutorial, but some topics are only covered in the tutorial. Visit http://flaskbook.com for more information.

The application

The application I'm going to develop as part of this tutorial is a decently featured microblogging server that I decided to call *microblog*. Pretty creative, I know.

These are some of the topics I will cover as we make progress with this project:

- User management, including managing logins, sessions, user roles, profiles and user avatars.
- Database management, including migration handling.
- Web form support, including field validation.
- Pagination of long lists of items.
- Full text search.
- Email notifications to users.
- HTML templates.
- Support for multiple languages.
- Caching and other performance optimizations.
- Debugging techniques for development and production servers.
- Installation on a production server.

So as you see, I'm going pretty much for the whole thing. I hope this application, when finished, will serve as a sort of template for writing other web applications.

Requirements

If you have a computer that runs Python then you are probably good to go. The tutorial application should run just fine on Windows, OS X and Linux. Unless noted, the code presented in these articles has been tested against Python 2.7 and 3.4.

The tutorial assumes that you are familiar with the terminal window (command prompt for Windows users) and know the basic command line file management functions of your operating system. If you don't, then I recommend that you learn how to create directories, copy files, etc. using the command line before continuing.

Finally, you should be somewhat comfortable writing Python code. Familiarity with Python modules and packages is also recommended.

Installing Flask

Okay, let's get started!

If you haven't yet, go ahead and install Python.

Now we have to install Flask and several extensions that we will be using. My preferred way to do this is to create a virtual environment where everything gets installed, so that your main Python installation is not affected. As an added benefit, you won't need root access to do the installation in this way.

So, open up a terminal window, choose a location where you want your application to live and create a new folder there to contain it. Let's call the application folder `microblog`.

If you are using Python 3.4, then cd into the `microblog` folder and then create a virtual environment with the following command:

```
$ python -m venv flask
```

Note that in some operating systems you may need to use `python3` instead of `python`. The above command creates a private version of your Python interpreter inside a folder named `flask`.

If you are using any other version of Python older than 3.4, then you need to download and install virtualenv.py before you can create a virtual environment. If you are on Mac OS X, then you can install it with the following command:

```
$ sudo easy_install virtualenv
```

On Linux you likely have a package for your distribution. For example, if you use Ubuntu:

```
$ sudo apt-get install python-virtualenv
```

Windows users have the most difficulty in installing virtualenv, so if you want to avoid the trouble then install Python 3.4. If you want to install `virtualenv` on Windows then the easiest way is by installing `pip` first, as explaned in this page. Once `pip is installed, the following command installs`virtualenv`:

```
$ pip install virtualenv
```

We've seen above how to create a virtual environment in Python 3.4. For older versions of Python that have been expanded with `virtualenv`, the command that creates a virtual environment is the following:

```
$ virtualenv flask
```

Regardless of the method you use to create the virtual environment, you will end up with a folder named `flask` that contains a complete Python environment ready to be used for this project.

Virtual environments can be activated and deactivated, if desired. An activated environment adds the location of its `bin` folder to the system path, so that for example, when you type `python` you get the environment's version and not the system's one. But activating a virtual environment is not necessary, it is equally effective to invoke the interpreter by specifying its pathname.

If you are on Linux, OS X or Cygwin, install flask and extensions by entering the following commands, one after another:

```
$ flask/bin/pip install flask
$ flask/bin/pip install flask-login
$ flask/bin/pip install flask-openid
$ flask/bin/pip install flask-mail
$ flask/bin/pip install flask-sqlalchemy
$ flask/bin/pip install sqlalchemy-migrate
$ flask/bin/pip install flask-whooshalchemy
$ flask/bin/pip install flask-wtf
$ flask/bin/pip install flask-babel
$ flask/bin/pip install guess_language
$ flask/bin/pip install flipflop
$ flask/bin/pip install coverage
```

If you are on Windows the commands are slightly different:

```
$ flask\Scripts\pip install flask
$ flask\Scripts\pip install flask-login
$ flask\Scripts\pip install flask-openid
$ flask\Scripts\pip install flask-mail
$ flask\Scripts\pip install flask-sqlalchemy
$ flask\Scripts\pip install sqlalchemy-migrate
$ flask\Scripts\pip install flask-whooshalchemy
$ flask\Scripts\pip install flask-wtf
$ flask\Scripts\pip install flask-babel
$ flask\Scripts\pip install guess_language
$ flask\Scripts\pip install flipflop
$ flask\Scripts\pip install coverage
```

These commands will download and install all the packages that we will use for our application.

"Hello, World" in Flask

You now have a `flask` sub-folder inside your `microblog` folder that is populated with a Python interpreter and the Flask framework and extensions that we will use for this application. Now it's time to write our first web application!

After you `cd` to the `microblog` folder, let's create the basic folder structure for our application:

```
$ mkdir app
$ mkdir app/static
$ mkdir app/templates
$ mkdir tmp
```

The `app` folder will be where we will put our application package. The `static` sub-folder is where we will store static files like images, javascripts, and cascading style sheets. The `templates` sub-folder is obviously where our templates will go.

Let's start by creating a simple init script for our `app` package (file `app/__init__.py`):

```
from flask import Flask

app = Flask(__name__)
from app import views
```

The script above simply creates the application object (of class `Flask`) and then imports the views module, which we haven't written yet. Do not confuse `app` the variable (which gets assigned the `Flask` instance) with `app` the package (from which we import the `views` module).

If you are wondering why the `import` statement is at the end and not at the beginning of the script as it is always done, the reason is to avoid circular references, because you are going to see that the `views` module needs to import the `app` variable defined in this script. Putting the import at the end avoids the circular import error.

The views are the handlers that respond to requests from web browsers or other clients. In Flask handlers are written as Python functions. Each view function is mapped to one or more request URLs.

Let's write our first view function (file `app/views.py`):

```
from app import app

@app.route('/')
@app.route('/index')
def index():
    return "Hello, World!"
```

This view is actually pretty simple, it just returns a string, to be displayed on the client's web browser. The two `route` decorators above the function create the mappings from URLs `/` and `/index` to this function.

The final step to have a fully working web application is to create a script that starts up the development web server with our application. Let's call this script `run.py`, and put it in the root folder:

```
#!flask/bin/python
from app import app
app.run(debug=True)
```

The script simply imports the `app` variable from our app package and invokes its `run` method to start the server. Remember that the `app` variable holds the `Flask` instance that we created it above.

To start the app you just run this script. On OS X, Linux and Cygwin you have to indicate that this is an executable file before you can run it:

```
$ chmod a+x run.py
```

Then the script can simply be executed as follows:

```
./run.py
```

On Windows the process is a bit different. There is no need to indicate the file is executable. Instead you have to run the script as an argument to the Python interpreter from the virtual environment:

```
$ flask\Scripts\python run.py
```

After the server initializes it will listen on port 5000 waiting for connections. Now open up your web browser and enter the following URL in the address field:

```
http://localhost:5000
```

Alternatively you can use the following URL:

```
http://localhost:5000/index
```

Do you see the route mappings in action? The first URL maps to `/`, while the second maps to `/index`. Both routes are associated with our view function, so they produce the same result. If you enter any other URL you will get an error, since only these two have been defined.

When you are done playing with the server you can just hit Ctrl-C to stop it.

And with this I conclude this first installment of this tutorial.

For those of you that are lazy typists, you can download the code from this tutorial below:

Download microblog-0.1.zip.

Note that you still need to install Flask as indicated above before you can run the application.

What's next

In the next part of the series we will modify our little application to use HTML templates.

I hope to see you in the next chapter.

Miguel

Part II: Templates

Recap

If you followed the previous chapter you should have a fully working, yet very simple web application that has the following file structure:

```
    microblog\
      flask\
        <virtual environment files>
      app\
        static\
        templates\
        __init__.py
        views.py
      tmp\
      run.py
```

To run the application you execute the `run.py` script and then open the `http://localhost:5000` URL on your web browser.

We are picking up exactly from where we left off, so you may want to make sure you have the above application correctly installed and working.

Why we need templates

Let's consider how we can expand our little application.

We want the home page of our microblogging app to have a heading that welcomes the logged in user, that's pretty standard for applications of this kind. Ignore for now the fact that we have no way to log a user in, I'll present a workaround for this issue in a moment.

An easy option to output a nice and big heading would be to change our view function to output HTML, maybe something like this:

```
from app import app

@app.route('/')
@app.route('/index')
def index():
    user = {'nickname': 'Miguel'}  # fake user
    return '''
<html>
  <head>
    <title>Home Page</title>
  </head>
  <body>
    <h1>Hello, ''' + user['nickname'] + '''</h1>
  </body>
</html>
```

```
'''
```

Give the application a try to see how this looks in your browser.

Since we don't have support for users yet I have resorted to using a placeholder user object, sometimes called fake or mock object. This allows us to concentrate on certain aspects of our application that depend on parts of the system that haven't been built yet.

I hope you agree with me that the solution used above to deliver HTML to the browser is very ugly. Consider how complex the code will become if you have to return a large and complex HTML page with lots of dynamic content. And what if you need to change the layout of your web site in a large app that has dozens of views, each returning HTML directly? This is clearly not a scalable option.

Templates to the rescue

If you could keep the logic of your application separate from the layout or presentation of your web pages things would be much better organized, don't you think? You could even hire a web designer to create a killer web site while you code the site's behaviors in Python. Templates help implement this separation.

Let's write our first template (file `app/templates/index.html`):

```
<html>
  <head>
    <title>{{ title }} - microblog</title>
  </head>
  <body>
      <h1>Hello, {{ user.nickname }}!</h1>
  </body>
</html>
```

As you see above, we just wrote a mostly standard HTML page, with the only difference that there are some placeholders for the dynamic content enclosed in `{{ ... }}` sections.

Now let's see how we use this template from our view function (file `app/views.py`):

```
from flask import render_template
from app import app

@app.route('/')
@app.route('/index')
def index():
    user = {'nickname': 'Miguel'}  # fake user
    return render_template('index.html',
                           title='Home',
                           user=user)
```

Try the application at this point to see how the template works. Once you have the rendered page in your browser you may want to view the source HTML and compare it against the original template.

To render the template we had to import a new function from the Flask framework called

`render_template`. This function takes a template filename and a variable list of template arguments and returns the rendered template, with all the arguments replaced.

Under the covers, the `render_template` function invokes the Jinja2 templating engine that is part of the Flask framework. Jinja2 substitutes `{{...}}` blocks with the corresponding values provided as template arguments.

Control statements in templates

The Jinja2 templates also support control statements, given inside `{%...%}` blocks. Let's add an if statement to our template (file `app/templates/index.html`):

```
<html>
  <head>
    {% if title %}
    <title>{{ title }} - microblog</title>
    {% else %}
    <title>Welcome to microblog</title>
    {% endif %}
  </head>
  <body>
      <h1>Hello, {{ user.nickname }}!</h1>
  </body>
</html>
```

Now our template is a bit smarter. If the view function forgets to define a page title then instead of showing an empty title the template will provide its own title. Feel free to remove the `title` argument in the `render_template` call of our view function to see how the conditional statement works.

Loops in templates

The logged in user in our `microblog` application will probably want to see recent posts from followed users in the home page, so let's see how we can do that.

To begin, we use our handy fake object trick to create some users and some posts to show (file `app/views.py`):

```
def index():
    user = {'nickname': 'Miguel'}  # fake user
    posts = [  # fake array of posts
        {
            'author': {'nickname': 'John'},
            'body': 'Beautiful day in Portland!'
        },
        {
            'author': {'nickname': 'Susan'},
            'body': 'The Avengers movie was so cool!'
        }
    ]
    return render_template("index.html",
                           title='Home',
                           user=user,
```

```
                          posts=posts)
```

To represent user posts we are using a list, where each element has `author` and `body` fields. When we get to implement a real database we will preserve these field names, so we can design and test our template using the fake objects without having to worry about updating it when we move to a database.

On the template side we have to solve a new problem. The list can have any number of elements, it will be up to the view function to decide how many posts need to be presented. The template cannot make any assumptions about the number of posts, so it needs to be prepared to render as many posts as the view sends.

So let's see how we do this using a `for` control structure (file `app/templates/index.html`):

```
<html>
  <head>
    {% if title %}
    <title>{{ title }} - microblog</title>
    {% else %}
    <title>Welcome to microblog</title>
    {% endif %}
  </head>
  <body>
    <h1>Hi, {{ user.nickname }}!</h1>
    {% for post in posts %}
    <div><p>{{ post.author.nickname }} says: <b>{{ post.body }}</b></p></div>
    {% endfor %}
  </body>
</html>
```

Simple, right? Give it a try, and be sure to play with adding more content to the posts array.

Template inheritance

We have one more topic to cover before we close for the day.

Our `microblog` web application will need to have a navigation bar at the top of the page with a few links. Here you will get the link to edit your profile, to login, logout, etc.

We can add a navigation bar to our `index.html` template, but as our application grows we will be needing to implement more pages, and this navigation bar will have to be copied to all of them. Then you will have to keep all these identical copies of the navigation bar in sync, and that could become a lot of work if you have a lot of pages and templates.

Instead, we can use Jinja2's template inheritance feature, which allows us to move the parts of the page layout that are common to all templates and put them in a base template from which all other templates are derived.

So let's define a base template that includes the navigation bar and also the bit of title logic we implemented earlier (file `app/templates/base.html`):

```
<html>
```

```
  <head>
    {% if title %}
    <title>{{ title }} - microblog</title>
    {% else %}
    <title>Welcome to microblog</title>
    {% endif %}
  </head>
  <body>
    <div>Microblog: <a href="/index">Home</a></div>
    <hr>
    {% block content %}{% endblock %}
  </body>
</html>
```

In this template we use the `block` control statement to define the place where the derived templates can insert themselves. Blocks are given a unique name, and their content can be replaced or enhanced in derived templates.

And now what's left is to modify our `index.html` template to *inherit* from `base.html` (file `app/templates/index.html`):

```
{% extends "base.html" %}
{% block content %}
    <h1>Hi, {{ user.nickname }}!</h1>
    {% for post in posts %}
    <div><p>{{ post.author.nickname }} says: <b>{{ post.body }}</b></p></div>
    {% endfor %}
{% endblock %}
```

Since the `base.html` template will now take care of the general page structure we have removed those elements from this one and left only the content part. The `extends` block establishes the inheritance link between the two templates, so that Jinja2 knows that when it needs to render `index.html` it needs to include it inside `base.html`. The two templates have matching `block` statements with name `content`, and this is how Jinja2 knows how to combine the two into one. When we get to write new templates we will also create them as extensions to `base.html`.

Final words

If you want to save time, the `microblog` application in its current state is available here:

Download microblog-0.2.zip.

Note that the zip file does not include the flask virtual environment, you will need to create it following the instructions in the first chapter in the series before you can run the application.

If you have any questions or comments feel free to leave them below.

In the next chapter of the series we will be looking at web forms. I hope to see you then.

Miguel

Part III: Web Forms

Recap

In the previous chapter of the series we defined a simple template for the home page and used fake objects as placeholders for things we don't have yet, like users or blog posts.

In this article we are going to fill one of those many holes we still have in our app, we will be looking at how to work with web forms.

Web forms are one of the most basic building blocks in any web application. We will be using forms to allow users to write blog posts, and also for logging in to the application.

To follow this chapter along you need to have the `microblog` app as we left it at the end of the previous chapter. Please make sure the app is installed and running.

Configuration

To handle our web forms we are going to use the Flask-WTF extension, which in turn wraps the WTForms project in a way that integrates nicely with Flask apps.

Many Flask extensions require some amount of configuration, so we are going to setup a configuration file inside our root `microblog` folder so that it is easily accessible if it needs to be edited. Here is what we will start with (file `config.py`):

```
WTF_CSRF_ENABLED = True
SECRET_KEY = 'you-will-never-guess'
```

Pretty simple, it's just two settings that our Flask-WTF extension needs. The `WTF_CSRF_ENABLED` setting activates the cross-site request forgery prevention (note that this setting is enabled by default in current versions of Flask-WTF). In most cases you want to have this option enabled as it makes your app more secure.

The `SECRET_KEY` setting is only needed when CSRF is enabled, and is used to create a cryptographic token that is used to validate a form. When you write your own apps make sure to set the secret key to something that is difficult to guess.

Now that we have our config file we need to tell Flask to read it and use it. We can do this right after the Flask app object is created, as follows (file `app/__init__.py`):

```
from flask import Flask

app = Flask(__name__)
app.config.from_object('config')

from app import views
```

The user login form

Web forms are represented in Flask-WTF as classes, subclassed from base class `Form`. A form subclass simply defines the fields of the form as class variables.

Now we will create a login form that users will use to identify with the system. The login mechanism that we will support in our app is not the standard username/password type, we will have our users login using their OpenID. OpenIDs have the benefit that the authentication is done by the provider of the OpenID, so we don't have to validate passwords, which makes our site more secure to our users.

The OpenID login only requires one string, the so called OpenID. We will also throw a 'remember me' checkbox in the form, so that users can choose to have a cookie installed in their browsers that remembers their login when they come back.

Let's write our first form (file `app/forms.py`):

```
from flask.ext.wtf import Form
from wtforms import StringField, BooleanField
from wtforms.validators import DataRequired

class LoginForm(Form):
    openid = StringField('openid', validators=[DataRequired()])
    remember_me = BooleanField('remember_me', default=False)
```

I believe the class is pretty much self-explanatory. We imported the `Form` class, and the two form field classes that we need, `StringField` and `BooleanField`.

The `DataRequired` import is a validator, a function that can be attached to a field to perform validation on the data submitted by the user. The `DataRequired` validator simply checks that the field is not submitted empty. There are many more validators included with Flask-WTF, we will use some more in the future.

Form templates

We will also need a template that contains the HTML that produces the form. The good news is that the `LoginForm` class that we just created knows how to render form fields as HTML, so we just need to concentrate on the layout. Here is our login template (file `app/templates/login.html`):

```
<!-- extend from base layout -->
{% extends "base.html" %}

{% block content %}
  <h1>Sign In</h1>
  <form action="" method="post" name="login">
      {{ form.hidden_tag() }}
      <p>
          Please enter your OpenID:<br>
          {{ form.openid(size=80) }}<br>
      </p>
      <p>{{ form.remember_me }} Remember Me</p>
      <p><input type="submit" value="Sign In"></p>
```

```
  </form>
{% endblock %}
```

Note that in this template we are reusing the `base.html` template through the `extends` template inheritance statement. We will actually do this with all our templates, to ensure a consistent layout across all pages.

There are a few interesting differences between a regular HTML form and our template. This template expects a form object instantiated from the form class we just defined stored in a template argument named `form`. We will take care of sending this template argument to the template next, when we write the view function that renders this template.

The `form.hidden_tag()` template argument will get replaced with a hidden field that implements the CSRF prevention that we enabled in the configuration. This field needs to be in all your forms if you have CSRF enabled. The good news is that Flask-WTF handles it for us, we just need to make sure it is included in the form.

The actual fields of our form are rendered by the field objects, we just need to refer to a `{{form.field_name}}` template argument in the place where each field should be inserted. Some fields can take arguments. In our case, we are asking the text field to generate our `openid` field with a width of 80 characters.

Since we have not defined the submit button in the form class we have to define it as a regular field. The submit field does not carry any data so it doesn't need to be defined in the form class.

Form views

The final step before we can see our form is to code a view function that renders the template.

This is actually quite simple since we just need to pass a form object to the template. Here is our new view function (file `app/views.py`):

```
from flask import render_template, flash, redirect
from app import app
from .forms import LoginForm

# index view function suppressed for brevity

@app.route('/login', methods=['GET', 'POST'])
def login():
    form = LoginForm()
    return render_template('login.html',
                           title='Sign In',
                           form=form)
```

So basically, we have imported our `LoginForm` class, instantiated an object from it, and sent it down to the template. This is all that is required to get form fields rendered.

Let's ignore for now the `flash` and `redirect` imports. We'll use them a bit later.

The only other thing that is new here is the `methods` argument in the route decorator. This tells Flask that this view function accepts GET and POST requests. Without this the view will only accept GET requests. We will want to receive the POST requests, these are the ones that will bring in the form data entered by the user.

At this point you can try the app and see the form in your web browser. After you start the application you will want to open `http://localhost:5000/login` in your web browser, as this is the route we have associated with the login view function.

We have not coded the part that accepts data yet, so pressing the submit button will not have any effect at this time.

Receiving form data

Another area where Flask-WTF makes our job really easy is in the handling of the submitted form data. Here is an updated version of our login view function that validates and stores the form data (file `app/views.py`):

```
@app.route('/login', methods=['GET', 'POST'])
def login():
    form = LoginForm()
    if form.validate_on_submit():
        flash('Login requested for OpenID="%s", remember_me=%s' %
              (form.openid.data, str(form.remember_me.data)))
        return redirect('/index')
    return render_template('login.html',
                           title='Sign In',
                           form=form)
```

The `validate_on_submit` method does all the form processing work. If you call it when the form is being presented to the user (i.e. before the user got a chance to enter data on it) then it will return `False`, so in that case you know that you have to render the template.

When `validate_on_submit` is called as part of a form submission request, it will gather all the data, run all the validators attached to fields, and if everything is all right it will return `True`, indicating that the data is valid and can be processed. This is your indication that this data is safe to incorporate into the application.

If at least one field fails validation then the function will return `False` and that will cause the form to be rendered back to the user, and this will give the user a chance to correct any mistakes. We will see later how to show an error message when validation fails.

When `validate_on_submit` returns True our login view function calls two new functions, imported from Flask. The `flash` function is a quick way to show a message on the next page presented to the user. In this case we will use it for debugging, since we don't have all the infrastructure necessary to log in users yet, we will instead just display a message that shows the submitted data. The `flash` function is also extremely useful on production servers to provide feedback to the user

regarding an action.

The flashed messages will not appear automatically in our page, our templates need to display the messages in a way that works for the site layout. We will add these messages to the base template, so that all our templates inherit this functionality. This is the updated base template (file `app/templates/base.html`):

```
<html>
  <head>
    {% if title %}
    <title>{{ title }} - microblog</title>
    {% else %}
    <title>microblog</title>
    {% endif %}
  </head>
  <body>
    <div>Microblog: <a href="/index">Home</a></div>
    <hr>
    {% with messages = get_flashed_messages() %}
      {% if messages %}
        <ul>
        {% for message in messages %}
            <li>{{ message }} </li>
        {% endfor %}
        </ul>
      {% endif %}
    {% endwith %}
    {% block content %}{% endblock %}
  </body>
</html>
```

The technique to display the flashed message is hopefully self-explanatory. One interesting property of flash messages is that once they are requested through the `get_flashed_messages` function they are removed from the message list, so these messages appear in the first page requested by the user after the `flash` function is called, and then they disappear.

The other new function we used in our login view is `redirect`. This function tells the client web browser to navigate to a different page instead of the one requested. In our view function we use it to redirect to the index page we developed in previous chapters. Note that flashed messages will display even if a view function ends in a redirect.

This is a great time to start the app and test how the form works. Make sure you try submitting the form with the openid field empty, to see how the `DataRequired` validator halts the submission process.

Improving field validation

With the app in its current state, forms that are submitted with invalid data will not be accepted. Instead, the form will be presented back to the user to correct. This is exactly what we want.

What we are missing is an indication to the user of what is wrong with the form. Luckily, Flask-WTF also makes this an easy task.

When a field fails validation Flask-WTF adds a descriptive error message to the form object. These messages are available to the template, so we just need to add a bit of logic that renders them.

Here is our login template with field validation messages (file `app/templates/login.html`):

```
<!-- extend base layout -->
{% extends "base.html" %}

{% block content %}
  <h1>Sign In</h1>
  <form action="" method="post" name="login">
      {{ form.hidden_tag() }}
      <p>
          Please enter your OpenID:<br>
          {{ form.openid(size=80) }}<br>
          {% for error in form.openid.errors %}
            <span style="color: red;">[{{ error }}]</span>
          {% endfor %}<br>
      </p>
      <p>{{ form.remember_me }} Remember Me</p>
      <p><input type="submit" value="Sign In"></p>
  </form>
{% endblock %}
```

The only change we've made is to add a for loop that renders any messages added by the validators below the `openid` field. As a general rule, any fields that have validators attached will have errors added under `form.field_name.errors`. In our case we use `form.openid.errors`. We display these messages in a red style to call the user's attention.

Dealing with OpenIDs

In practice, we will find that a lot of people don't even know that they already have a few OpenIDs. It isn't that well known that a number of major service providers on the Internet support OpenID authentication for their members. For example, if you have an account with Google, you have an OpenID with them. Likewise with Yahoo, AOL, Flickr and many other providers. (Update: Google is shutting down their OpenID service on April 15 2015).

To make it easier for users to login to our site with one of these commonly available OpenIDs, we will add links to a short list of them, so that the user does not have to type the OpenID by hand.

We will start by defining the list of OpenID providers that we want to present. We can do this in our config file (file `config.py`):

```
WTF_CSRF_ENABLED = True
SECRET_KEY = 'you-will-never-guess'

OPENID_PROVIDERS = [
    {'name': 'Google', 'url': 'https://www.google.com/accounts/o8/id'},
    {'name': 'Yahoo', 'url': 'https://me.yahoo.com'},
    {'name': 'AOL', 'url': 'http://openid.aol.com/<username>'},
    {'name': 'Flickr', 'url': 'http://www.flickr.com/<username>'},
    {'name': 'MyOpenID', 'url': 'https://www.myopenid.com'}]
```

Now let's see how we use this array in our login view function:

```
@app.route('/login', methods=['GET', 'POST'])
def login():
    form = LoginForm()
    if form.validate_on_submit():
        flash('Login requested for OpenID="%s", remember_me=%s' %
              (form.openid.data, str(form.remember_me.data)))
        return redirect('/index')
    return render_template('login.html',
                           title='Sign In',
                           form=form,
                           providers=app.config['OPENID_PROVIDERS'])
```

Here we grab the configuration by looking it up in `app.config` with its key. The array is then added to the `render_template` call as a template argument.

As I'm sure you guessed, we have one more step to be done with this. We now need to specify how we would like to render these provider links in our login template (file `app/templates/login.html`):

```
<!-- extend base layout -->
{% extends "base.html" %}

{% block content %}
<script type="text/javascript">
function set_openid(openid, pr)
{
    u = openid.search('<username>')
    if (u != -1) {
        // openid requires username
        user = prompt('Enter your ' + pr + ' username:')
        openid = openid.substr(0, u) + user
    }
    form = document.forms['login'];
    form.elements['openid'].value = openid
}
</script>
<h1>Sign In</h1>
<form action="" method="post" name="login">
    {{ form.hidden_tag() }}
    <p>
        Please enter your OpenID, or select one of the providers below:<br>
        {{ form.openid(size=80) }}
        {% for error in form.openid.errors %}
          <span style="color: red;">[{{error}}]</span>
        {% endfor %}<br>
        |{% for pr in providers %}
          <a href="javascript:set_openid('{{ pr.url }}',
'{{ pr.name }}');">{{ pr.name }}</a> |
        {% endfor %}
    </p>
    <p>{{ form.remember_me }} Remember Me</p>
    <p><input type="submit" value="Sign In"></p>
</form>
{% endblock %}
```

The template got somewhat long with this change. Some OpenIDs include the user's username, so for those we have to have a bit of javascript magic that prompts the user for the username and then composes the OpenID with it. When the user clicks on an OpenID provider link and (optionally) enters the username, the OpenID for that provider is inserted in the text field.

Below is a screenshot of our login screen after clicking the Google OpenID link:

Final Words

While we have made a lot of progress with our login form, we haven't actually done anything to login users into our system, all we've done so far had to do with the GUI aspects of the login process. This is because before we can do real logins we need to have a database where we can record our users.

In the next chapter we will get our database up and running, and shortly after we will complete our login system, so stay tuned for the follow up articles.

The `microblog` application in its current state is available for download here:

Download microblog-0.3.zip.

Remember that the Flask virtual environment is not included in the zip file. For instructions on how to set it up see the first chapter of the series.

Feel free to leave comments or questions below. I hope to see you in the next chapter.

Miguel

Part IV: Database

Recap

In the previous chapter of the series we created our login form, complete with submission and validation.

In this article we are going to create our database and set it up so that we can record our users in it.

To follow this chapter along you need to have the `microblog` app as we left it at the end of the previous chapter. Please make sure the app is installed and running.

Running Python scripts from the command line

In this chapter we are going to write a few scripts that simplify the management of our database. Before we get into that let's review how a Python script is executed on the command line.

If you are on Linux or OS X, then scripts have to be given executable permission, like this:

```
$ chmod a+x script.py
```

The script has a shebang line, which points to the interpreter that should be used. A script that has been given executable permission and has a shebang line can be executed simply like this:

```
./script.py <arguments>
```

On Windows, however, this does not work, and instead you have to provide the script as an argument to the chosen Python interpreter:

```
$ flask\Scripts\python script.py <arguments>
```

To avoid having to type the path to the Python interpreter you can add your `microblog/flask/Scripts` directory to the system path, making sure it appears before your regular Python interpreter. This can be temporarily achieved by activating the virtual environment with the following command:

```
$ flask\Scripts\activate
```

From now on, in this tutorial the Linux/OS X syntax will be used for brevity. If you are on Windows remember to convert the syntax appropriately.

Databases in Flask

We will use the Flask-SQLAlchemy extension to manage our application. This extension provides a wrapper for the SQLAlchemy project, which is an Object Relational Mapper or ORM.

ORMs allow database applications to work with objects instead of tables and SQL. The operations performed on the objects are translated into database commands transparently by the ORM. Knowing SQL can be very helpful when working with ORMs, but we will not be learning SQL in this tutorial, we will let Flask-SQLAlchemy speak SQL for us.

Migrations

Most database tutorials I've seen cover creation and use of a database, but do not adequately address the problem of updating a database as the application grows. Typically you end up having to delete the old database and create a new one each time you need to make updates, losing all the data. And if the data cannot be recreated easily you may be forced to write export and import scripts yourself.

Luckily, we have a much better option.

We are going to use SQLAlchemy-migrate to keep track of database updates for us. It adds a bit of work to get a database started, but that is a small price to pay for never having to worry about manual database migrations.

Enough theory, let's get started!

Configuration

For our little application we will use a sqlite database. The sqlite databases are the most convenient choice for small applications, as each database is stored in a single file and there is no need to start a database server.

We have a couple of new configuration items to add to our config file (file `config.py`):

```
import os
basedir = os.path.abspath(os.path.dirname(__file__))

SQLALCHEMY_DATABASE_URI = 'sqlite:///' + os.path.join(basedir, 'app.db')
SQLALCHEMY_MIGRATE_REPO = os.path.join(basedir, 'db_repository')
```

The SQLALCHEMY_DATABASE_URI is required by the Flask-SQLAlchemy extension. This is the path of our database file.

The SQLALCHEMY_MIGRATE_REPO is the folder where we will store the SQLAlchemy-migrate data files.

Finally, when we initialize our app we also need to initialize our database. Here is our updated package init file (file `app/__init__.py`):

```
from flask import Flask
from flask.ext.sqlalchemy import SQLAlchemy

app = Flask(__name__)
app.config.from_object('config')
db = SQLAlchemy(app)
```

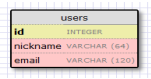
```
from app import views, models
```

Note the two changes we have made to our init script. We are now creating a `db` object that will be our database, and we are also importing a new module called `models`. We will write this module next.

The database model

The data that we will store in our database will be represented by a collection of classes that are referred to as the database models. The ORM layer will do the translations required to map objects created from these classes into rows in the proper database table.

Let's start by creating a model that will represent our users. Using the WWW SQL Designer tool, I have made the following diagram to represent our users table:



The `id` field is usually in all models, and is used as the *primary key*. Each user in the database will be assigned a unique id value, stored in this field. Luckily this is done automatically for us, we just need to provide the `id` field.

The `nickname` and `email` fields are defined as strings (or VARCHAR in database jargon), and their maximum lengths are specified so that the database can optimize space usage.

Now that we have decided what we want our users table to look like, the job of translating that into code is pretty easy (file `app/models.py`):

```
from app import db

class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    nickname = db.Column(db.String(64), index=True, unique=True)
    email = db.Column(db.String(120), index=True, unique=True)

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

The `User` class that we just created contains several fields, defined as class variables. Fields are created as instances of the `db.Column` class, which takes the field type as an argument, plus other optional arguments that allow us, for example, to indicate which fields are unique and indexed.

The `__repr__` method tells Python how to print objects of this class. We will use this for debugging.

Creating the database

With the configuration and model in place we are now ready to create our database file. The SQLAlchemy-migrate package comes with command line tools and APIs to create databases in a way that allows easy updates in the future, so that is what we will use. I find the command line tools a bit

awkward to use, so instead I have written my own set of little Python scripts that invoke the migration APIs.

Here is a script that creates the database (file `db_create.py`):

```
#!flask/bin/python
from migrate.versioning import api
from config import SQLALCHEMY_DATABASE_URI
from config import SQLALCHEMY_MIGRATE_REPO
from app import db
import os.path
db.create_all()
if not os.path.exists(SQLALCHEMY_MIGRATE_REPO):
    api.create(SQLALCHEMY_MIGRATE_REPO, 'database repository')
    api.version_control(SQLALCHEMY_DATABASE_URI, SQLALCHEMY_MIGRATE_REPO)
else:
    api.version_control(SQLALCHEMY_DATABASE_URI, SQLALCHEMY_MIGRATE_REPO,
api.version(SQLALCHEMY_MIGRATE_REPO))
```

Note how this script is completely generic. All the application specific pathnames are imported from the config file. When you start your own project you can just copy the script to the new app's directory and it will work right away.

To create the database you just need to execute this script (remember that if you are on Windows the command is slightly different):

```
./db_create.py
```

After you run the command you will have a new `app.db` file. This is an empty sqlite database, created from the start to support migrations. You will also have a `db_repository` directory with some files inside. This is the place where SQLAlchemy-migrate stores its data files. Note that we do not regenerate the repository if it already exists. This will allow us to recreate the database while leaving the existing repository if we need to.

Our first migration

Now that we have defined our model, we can incorporate it into our database. We will consider any changes to the structure of the app database a *migration*, so this is our first, which will take us from an empty database to a database that can store users.

To generate a migration I use another little Python helper script (file `db_migrate.py`):

```
#!flask/bin/python
import imp
from migrate.versioning import api
from app import db
from config import SQLALCHEMY_DATABASE_URI
from config import SQLALCHEMY_MIGRATE_REPO
v = api.db_version(SQLALCHEMY_DATABASE_URI, SQLALCHEMY_MIGRATE_REPO)
migration = SQLALCHEMY_MIGRATE_REPO + ('/versions/%03d_migration.py' % (v+1))
tmp_module = imp.new_module('old_model')
old_model = api.create_model(SQLALCHEMY_DATABASE_URI, SQLALCHEMY_MIGRATE_REPO)
```

```
exec(old_model, tmp_module.__dict__)
script = api.make_update_script_for_model(SQLALCHEMY_DATABASE_URI,
SQLALCHEMY_MIGRATE_REPO, tmp_module.meta, db.metadata)
open(migration, "wt").write(script)
api.upgrade(SQLALCHEMY_DATABASE_URI, SQLALCHEMY_MIGRATE_REPO)
v = api.db_version(SQLALCHEMY_DATABASE_URI, SQLALCHEMY_MIGRATE_REPO)
print('New migration saved as ' + migration)
print('Current database version: ' + str(v))
```

The script looks complicated, but it doesn't really do much. The way SQLAlchemy-migrate creates a migration is by comparing the structure of the database (obtained in our case from file `app.db`) against the structure of our models (obtained from file `app/models.py`). The differences between the two are recorded as a migration script inside the migration repository. The migration script knows how to apply a migration or undo it, so it is always possible to upgrade or downgrade a database format.

While I have never had problems generating migrations automatically with the above script, I could see that sometimes it would be hard to determine what changes were made just by comparing the old and the new format. To make it easy for SQLAlchemy-migrate to determine the changes I never rename existing fields, I limit my changes to adding or removing models or fields, or changing types of existing fields. And I always review the generated migration script to make sure it is right.

It goes without saying that you should never attempt to migrate your database without having a backup, in case something goes wrong. Also never run a migration for the first time on a production database, always make sure the migration works correctly on a development database.

So let's go ahead and record our migration:

```
$ ./db_migrate.py
```

And the output from the script will be:

```
New migration saved as db_repository/versions/001_migration.py
Current database version: 1
```

The script shows where the migration script was stored, and also prints the current database version. The empty database version was version 0, after we migrated to include users we are at version 1.

Database upgrades and downgrades

By now you may be wondering why is it that important to go through the extra hassle of recording database migrations.

Imagine that you have your application in your development machine, and also have a copy deployed to a production server that is online and in use.

Let's say that for the next release of your app you have to introduce a change to your models, for example a new table needs to be added. Without migrations you would need to figure out how to change the format of your database, both in your development machine and then again in your server,

and this could be a lot of work.

If you have database migration support, then when you are ready to release the new version of the app to your production server you just need to record a new migration, copy the migration scripts to your production server and run a simple script that applies the changes for you. The database upgrade can be done with this little Python script (file `db_upgrade.py`):

```
#!flask/bin/python
from migrate.versioning import api
from config import SQLALCHEMY_DATABASE_URI
from config import SQLALCHEMY_MIGRATE_REPO
api.upgrade(SQLALCHEMY_DATABASE_URI, SQLALCHEMY_MIGRATE_REPO)
v = api.db_version(SQLALCHEMY_DATABASE_URI, SQLALCHEMY_MIGRATE_REPO)
print('Current database version: ' + str(v))
```

When you run the above script, the database will be upgraded to the latest revision, by applying the migration scripts stored in the database repository.

It is not a common need to have to downgrade a database to an old format, but just in case, SQLAlchemy-migrate supports this as well (file `db_downgrade.py`):

```
#!flask/bin/python
from migrate.versioning import api
from config import SQLALCHEMY_DATABASE_URI
from config import SQLALCHEMY_MIGRATE_REPO
v = api.db_version(SQLALCHEMY_DATABASE_URI, SQLALCHEMY_MIGRATE_REPO)
api.downgrade(SQLALCHEMY_DATABASE_URI, SQLALCHEMY_MIGRATE_REPO, v - 1)
v = api.db_version(SQLALCHEMY_DATABASE_URI, SQLALCHEMY_MIGRATE_REPO)
print('Current database version: ' + str(v))
```

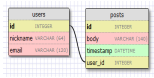
This script will downgrade the database one revision. You can run it multiple times to downgrade several revisions.

Database relationships

Relational databases are good at storing relations between data items. Consider the case of a user writing a blog post. The user will have a record in the `users` table, and the post will have a record in the `posts` table. The most efficient way to record who wrote a given post is to link the two related records.

Once a link between a user and a post is established there are two types of queries that we may need to use. The most trivial one is when you have a blog post and need to know what user wrote it. A more complex query is the reverse of this one. If you have a user, you may want to know all the posts that the user wrote. Flask-SQLAlchemy will help us with both types of queries.

Let's expand our database to store posts, so that we can see relationships in action. For this we go back to our database design tool and create a `posts` table:



Our `posts` table will have the required `id`, the `body` of the post and a `timestamp`. Not much new there. But the `user_id` field deserves an explanation.

We said we wanted to link users to the posts that they write. The way to do that is by adding a field to the post that contains the `id` of the user that wrote it. This `id` is called a *foreign key*. Our database design tool shows foreign keys as a link between the foreign key and the `id` field of the table it refers to. This kind of link is called a one-to-many relationship, *one* user writes *many* posts.

Let's modify our models to reflect these changes (`app/models.py`):

```
from app import db

class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    nickname = db.Column(db.String(64), index=True, unique=True)
    email = db.Column(db.String(120), index=True, unique=True)
    posts = db.relationship('Post', backref='author', lazy='dynamic')

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

class Post(db.Model):
    id = db.Column(db.Integer, primary_key = True)
    body = db.Column(db.String(140))
    timestamp = db.Column(db.DateTime)
    user_id = db.Column(db.Integer, db.ForeignKey('user.id'))

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

We have added the `Post` class, which will represent blog posts written by users. The `user_id` field in the `Post` class was initialized as a *foreign key*, so that Flask-SQLAlchemy knows that this field will link to a user.

Note that we have also added a new field to the `User` class called `posts`, that is constructed as a `db.relationship` field. This is not an actual database field, so it isn't in our database diagram. For a one-to-many relationship a `db.relationship` field is normally defined on the "one" side. With this relationship we get a `user.posts` member that gets us the list of posts from the user. The first argument to `db.relationship` indicates the "many" class of this relationship. The `backref` argument defines a field that will be added to the objects of the "many" class that points back at the "one" object. In our case this means that we can use `post.author` to get the User instance that created a post. Don't worry if these details don't make much sense just yet, we'll see examples of this at the end of this article.

Let's record another migration with this change. Simply run:

```
$ ./db_migrate.py
```

And the script will respond:

```
New migration saved as db_repository/versions/002_migration.py
```

```
Current database version: 2
```

It isn't really necessary to record each little change to the database model as a separate migration, a migration is normally only recorded at significant points in the history of the project. We are doing more migrations than necessary here only to show how the migration system works.

Play time

We have spent a lot of time defining our database, but we haven't seen how it works yet. Since our app does not have database code yet let's make use of our brand new database in the Python interpreter.

So go ahead and fire up Python. On Linux or OS X:

```
flask/bin/python
```

Or on Windows:

```
flask\Scripts\python
```

Once in the Python prompt enter the following:

```
>>> from app import db, models
>>>
```

This brings our database and models into memory.

Let's create a new user:

```
>>> u = models.User(nickname='john', email='john@email.com')
>>> db.session.add(u)
>>> db.session.commit()
>>>
```

Changes to a database are done in the context of a session. Multiple changes can be accumulated in a session and once all the changes have been registered you can issue a single `db.session.commit()`, which writes the changes atomically. If at any time while working on a session there is an error, a call to `db.session.rollback()` will revert the database to its state before the session was started. If neither `commit` nor `rollback` are issued then the system by default will roll back the session. Sessions guarantee that the database will never be left in an inconsistent state.

Let's add another user:

```
>>> u = models.User(nickname='susan', email='susan@email.com')
>>> db.session.add(u)
>>> db.session.commit()
>>>
```

Now we can query what our users are:

```
>>> users = models.User.query.all()
```

```
>>> users
[<User u'john'>, <User u'susan'>]
>>> for u in users:
...     print(u.id,u.nickname)
...
1 john
2 susan
>>>
```

For this we have used the `query` member, which is available in all model classes. Note how the `id` member was automatically set for us.

Here is another way to do queries. If we know the `id` of a user we can find the data for that user as follows:

```
>>> u = models.User.query.get(1)
>>> u
<User u'john'>
>>>
```

Now let's add a blog post:

```
>>> import datetime
>>> u = models.User.query.get(1)
>>> p = models.Post(body='my first post!', timestamp=datetime.datetime.utcnow(),
author=u)
>>> db.session.add(p)
>>> db.session.commit()
```

Here we set our `timestamp` in UTC time zone. All timestamps stored in our database will be in UTC. We can have users from all over the world writing posts and we need to use uniform time units. In a future tutorial we will see how to show these times to users in their local timezone.

You may have noticed that we have not set the `user_id` field of the Post class. Instead, we are storing a User object inside the `author` field. The `author` field is a virtual field that was added by Flask-SQLAlchemy to help with relationships, we have defined the name of this field in the `backref` argument to `db.relationship` in our model. With this information the ORM layer will know how to complete the `user_id` for us.

To complete this session, let's look at a few more database queries that we can do:

```
# get all posts from a user
>>> u = models.User.query.get(1)
>>> u
<User u'john'>
>>> posts = u.posts.all()
>>> posts
[<Post u'my first post!'>]

# obtain author of each post
>>> for p in posts:
...     print(p.id,p.author.nickname,p.body)
...
1 john my first post!
```

```
# a user that has no posts
>>> u = models.User.query.get(2)
>>> u
<User u'susan'>
>>> u.posts.all()
[]

# get all users in reverse alphabetical order
>>> models.User.query.order_by('nickname desc').all()
[<User u'susan'>, <User u'john'>]
>>>
```

The Flask-SQLAlchemy documentation is the best place to learn about the many options that are available to query the database.

Before we close, let's erase the test users and posts we have created, so that we can start from a clean database in the next chapter:

```
>>> users = models.User.query.all()
>>> for u in users:
...     db.session.delete(u)
...
>>> posts = models.Post.query.all()
>>> for p in posts:
...     db.session.delete(p)
...
>>> db.session.commit()
>>>
```

Final words

This was a long tutorial. We have learned the basics of working with a database, but we haven't incorporated the database into our application yet. In the next chapter we will put all we have learned about databases into practice when we look at our user login system.

In the meantime, if you haven't been writing the application along, you may want to download it in its current state:

Download microblog-0.4.zip.

Note that I have not included a database in the zip file above, but the repository with the migrations is there. To create a new database just use the `db_create.py` script, then use `db_upgrade.py` to upgrade the database to the latest revision.

I hope to see you next time!

Miguel

Part V: User Logins

Recap

In the previous chapter of the series we created our database and learned how to populate it with users and posts, but we haven't hooked up any of that into our app yet. And two chapters ago we've seen how to create web forms and left with a fully implemented login form.

In this article we are going to build on what we learned about web forms and databases and write our user login system. At the end of this tutorial our little application will register new users and log them in and out.

To follow this chapter along you need to have the `microblog` app as we left it at the end of the previous chapter. Please make sure the app is installed and running.

Configuration

As in previous chapters, we start by configuring the Flask extensions that we will use. For the login system we will use two extensions, Flask-Login and Flask-OpenID. Flask-Login will handle our users logged in state, while Flask-OpenID will provide authentication. These extensions are configured as follows (file `app/__init__.py`):

```
import os
from flask.ext.login import LoginManager
from flask.ext.openid import OpenID
from config import basedir

lm = LoginManager()
lm.init_app(app)
oid = OpenID(app, os.path.join(basedir, 'tmp'))
```

The Flask-OpenID extension requires a path to a temp folder where files can be stored. For this we provide the location of our `tmp` folder.

Python 3 Compatiblity

Unfortunately version 1.2.1 of Flask-OpenID (the current official version) does not work well with Python 3. Check what version you have by running the following command:

```
$ flask/bin/pip freeze
```

If you have a version newer than 1.2.1 then the problem is likely resolved, but if you have 1.2.1 and are following this tutorial on Python 3 then you have to install the development version from GitHub:

```
$ flask/bin/pip uninstall flask-openid
$ flask/bin/pip install git+git://github.com/mitsuhiko/flask-openid.git
```

Note that you need to have `git` installed for this to work.

Revisiting our User model

The Flask-Login extension expects certain methods to be implemented in our `User` class. Outside of these methods there are no requirements for how the class has to be implemented.

Below is our Flask-Login friendly `User` class (file `app/models.py`):

```
class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    nickname = db.Column(db.String(64), index=True, unique=True)
    email = db.Column(db.String(120), index=True, unique=True)
    posts = db.relationship('Post', backref='author', lazy='dynamic')

    def is_authenticated(self):
        return True

    def is_active(self):
        return True

    def is_anonymous(self):
        return False

    def get_id(self):
        try:
            return unicode(self.id)  # python 2
        except NameError:
            return str(self.id)  # python 3

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

The `is_authenticated` method has a misleading name. In general this method should just return `True` unless the object represents a user that should not be allowed to authenticate for some reason.

The `is_active` method should return True for users unless they are inactive, for example because they have been banned.

The `is_anonymous` method should return True only for fake users that are not supposed to log in to the system.

Finally, the `get_id` method should return a unique identifier for the user, in unicode format. We use the unique id generated by the database layer for this. Note that due to the differences in unicode handling between Python 2 and 3 we have to provide two alternative versions of this method.

User loader callback

Now we are ready to start implementing the login system using the Flask-Login and Flask-OpenID extensions.

First, we have to write a function that loads a user from the database. This function will be used by

Flask-Login (file `app/views.py`):

```
@lm.user_loader
def load_user(id):
    return User.query.get(int(id))
```

Note how this function is registered with Flask-Login through the `lm.user_loader` decorator. Also remember that user ids in Flask-Login are always unicode strings, so a conversion to an integer is necessary before we can send the id to Flask-SQLAlchemy.

The login view function

Next let's update our login view function (file `app/views.py`):

```
from flask import render_template, flash, redirect, session, url_for, request, g
from flask.ext.login import login_user, logout_user, current_user, login_required
from app import app, db, lm, oid
from .forms import LoginForm
from .models import User

@app.route('/login', methods=['GET', 'POST'])
@oid.loginhandler
def login():
    if g.user is not None and g.user.is_authenticated():
        return redirect(url_for('index'))
    form = LoginForm()
    if form.validate_on_submit():
        session['remember_me'] = form.remember_me.data
        return oid.try_login(form.openid.data, ask_for=['nickname', 'email'])
    return render_template('login.html',
                           title='Sign In',
                           form=form,
                           providers=app.config['OPENID_PROVIDERS'])
```

Notice we have imported several new modules, some of which we will use later.

The changes from our previous version are very small. We have added a new decorator to our view function. The `oid.loginhandler` tells Flask-OpenID that this is our login view function.

At the top of the function body we check if `g.user` is set to an authenticated user, and in that case we redirect to the index page. The idea here is that if there is a logged in user already we will not do a second login on top.

The `g` global is setup by Flask as a place to store and share data during the life of a request. As I'm sure you guessed by now, we will be storing the logged in user here.

The `url_for` function that we used in the `redirect` call is defined by Flask as a clean way to obtain the URL for a given view function. If you want to redirect to the index page you may very well use `redirect('/index')`, but there are very good reasons to let Flask build URLs for you.

The code that runs when we get a data back from the login form is also new. Here we do two things. First we store the value of the `remember_me` boolean in the flask *session*, not to be confused with the

`db.session` from Flask-SQLAlchemy. We've seen that the `flask.g` object stores and shares data though the life of a request. The `flask.session` provides a much more complex service along those lines. Once data is stored in the session object it will be available during that request and any future requests *made by the same client*. Data remains in the session until explicitly removed. To be able to do this, Flask keeps a different session container for each client of our application.

The `oid.try_login` call in the following line is the call that triggers the user authentication through Flask-OpenID. The function takes two arguments, the `openid` given by the user in the web form and a list of data items that we want from the OpenID provider. Since we defined our User class to include `nickname` and `email`, those are the items we are going to ask for.

The OpenID authentication happens asynchronously. Flask-OpenID will call a function that is registered with the `oid.after_login` decorator if the authentication is successful. If the authentication fails the user will be taken back to the login page.

The Flask-OpenID login callback

Here is our implementation of the `after_login` function (file `app/views.py`):

```
@oid.after_login
def after_login(resp):
    if resp.email is None or resp.email == "":
        flash('Invalid login. Please try again.')
        return redirect(url_for('login'))
    user = User.query.filter_by(email=resp.email).first()
    if user is None:
        nickname = resp.nickname
        if nickname is None or nickname == "":
            nickname = resp.email.split('@')[0]
        user = User(nickname=nickname, email=resp.email)
        db.session.add(user)
        db.session.commit()
    remember_me = False
    if 'remember_me' in session:
        remember_me = session['remember_me']
        session.pop('remember_me', None)
    login_user(user, remember = remember_me)
    return redirect(request.args.get('next') or url_for('index'))
```

The `resp` argument passed to the `after_login` function contains information returned by the OpenID provider.

The first `if` statement is just for validation. We require a valid email, so if an email was not provided we cannot log the user in.

Next, we search our database for the email provided. If the email is not found we consider this a new user, so we add a new user to our database, pretty much as we have learned in the previous chapter. Note that we handle the case of a missing `nickname`, since some OpenID providers may not have that information.

After that we load the `remember_me` value from the Flask session, this is the boolean that we stored in the login view function, if it is available.

Then we call Flask-Login's `login_user` function, to register this is a valid login.

Finally, in the last line we redirect to the *next* page, or the index page if a next page was not provided in the request.

The concept of the next page is simple. Let's say you navigate to a page that requires you to be logged in, but you aren't just yet. In Flask-Login you can protect views against non logged in users by adding the `login_required` decorator. If the user tries to access one of the affected URLs then it will be redirected to the login page automatically. Flask-Login will store the original URL as the `next` page, and it is up to us to return the user to this page once the login process completed.

For this to work Flask-Login needs to know what view logs users in. We can configure this in the app's module initializer (file `app/__init__.py`):

```
lm = LoginManager()
lm.init_app(app)
lm.login_view = 'login'
```

The g.user global

If you were paying attention, you will remember that in the login view function we check `g.user` to determine if a user is already logged in. To implement this we will use the `before_request` event from Flask. Any functions that are decorated with `before_request` will run before the view function each time a request is received. So this is the right place to setup our `g.user` variable (file `app/views.py`):

```
@app.before_request
def before_request():
    g.user = current_user
```

This is all it takes. The `current_user` global is set by Flask-Login, so we just put a copy in the `g` object to have better access to it. With this, all requests will have access to the logged in user, even inside templates.

The index view

In a previous chapter we left our `index` view function using fake objects, because at the time we did not have users or posts in our system. Well, we have users now, so let's hook that up:

```
@app.route('/')
@app.route('/index')
@login_required
def index():
    user = g.user
    posts = [
```

```
        {
            'author': {'nickname': 'John'},
            'body': 'Beautiful day in Portland!'
        },
        {
            'author': {'nickname': 'Susan'},
            'body': 'The Avengers movie was so cool!'
        }
    ]
    return render_template('index.html',
                           title='Home',
                           user=user,
                           posts=posts)
```

There are only two changes to this function. First, we have added the `login_required` decorator. This will ensure that this page is only seen by logged in users.

The other change is that we pass `g.user` down to the template, instead of the fake object we used in the past.

This is a good time to run the application.

When you navigate to `http://localhost:5000` you will instead get the login page. Keep in mind that to login with OpenID you have to use the OpenID URL from your provider. You can use one of the OpenID provider links below the URL text field to generate the correct URL for you.

As part of the login process you will be redirected to your provider's web site, where you will authenticate and authorize the sharing of some information with our application (just the email and nickname that we requested, no passwords or other personal information will be exposed).

Once the login is complete you will be taken to the index page, this time as a logged in user.

Feel free to try the `remember_me` checkbox. With this option enabled you can close and reopen your web browser and will continue to be logged in.

Logging out

We have implemented the log in, now it's time to add the log out.

The view function for logging out is extremely simple (file `app/views.py`):

```
@app.route('/logout')
def logout():
    logout_user()
    return redirect(url_for('index'))
```

But we are also missing a link to logout in the template. We are going to put this link in the top navigation bar which is in the base layout (file `app/templates/base.html`):

```
<html>
  <head>
    {% if title %}
    <title>{{ title }} - microblog</title>
```

```
    {% else %}
    <title>microblog</title>
    {% endif %}
  </head>
  <body>
    <div>Microblog:
        <a href="{{ url_for('index') }}">Home</a>
        {% if g.user.is_authenticated() %}
        | <a href="{{ url_for('logout') }}">Logout</a>
        {% endif %}
    </div>
    <hr>
    {% with messages = get_flashed_messages() %}
    {% if messages %}
    <ul>
    {% for message in messages %}
        <li>{{ message }} </li>
    {% endfor %}
    </ul>
    {% endif %}
    {% endwith %}
    {% block content %}{% endblock %}
  </body>
</html>
```

Note how easy it is to do this. We just needed to check if we have a valid user set in `g.user` and if we do we just add the logout link. We have also used the opportunity to use `url_for` in our template.

Final words

We now have a fully functioning user login system. In the next chapter we will be creating the user profile page and will be displaying user avatars on them.

In the meantime, here is the updated application code including all the changes in this article:

Download microblog-0.5.zip.

See you next time!

Miguel

Part VI: Profile Page

Recap

In the previous chapter of this tutorial we created our user login system, so we can now have users log in and out of the website using their OpenIDs.

Today, we are going to work on the user profiles. First, we'll create the user profile page, which shows the user's information and more recent blog posts, and as part of that we will learn how to show user avatars. Then we are going to create a web form for users to edit their profiles.

User Profile Page

Creating a user profile page does not really require any new major concepts to be introduced. We just need to create a new view function and its accompanying HTML template.

Here is the view function (file `app/views.py`):

```
@app.route('/user/<nickname>')
@login_required
def user(nickname):
    user = User.query.filter_by(nickname=nickname).first()
    if user == None:
        flash('User %s not found.' % nickname)
        return redirect(url_for('index'))
    posts = [
        {'author': user, 'body': 'Test post #1'},
        {'author': user, 'body': 'Test post #2'}
    ]
    return render_template('user.html',
                           user=user,
                           posts=posts)
```

The `@app.route` decorator that we used to declare this view function looks a little bit different than the previous ones. In this case we have an *argument* in it, which is indicated as `<nickname>`. This translates into an argument of the same name added to the view function. When the client requests, say, URL `/user/miguel` the view function will be invoked with `nickname` set to `'miguel'`.

The implementation of the view function should have no surprises. First we try to load the user from the database, using the nickname that we received as argument. If that doesn't work then we just redirect to the main page with an error message, as we have seen in the previous chapter.

Once we have our user, we just send it in the `render_template` call, along with some fake posts. Note that in the user profile page we will be displaying only posts by this user, so our fake posts have the `author` field correctly set.

Our initial view template will be extremely simple (file `app/templates/user.html`):

```
<!-- extend base layout -->
{% extends "base.html" %}

{% block content %}
  <h1>User: {{ user.nickname }}!</h1>
  <hr>
  {% for post in posts %}
  <p>
    {{ post.author.nickname }} says: <b>{{ post.body }}</b>
  </p>
  {% endfor %}
{% endblock %}
```

The profile page is now complete, but a link to it does not exist anywhere in the web site. To make it a bit more easy for a user to check his or her own profile, we are going to add a link to it in the navigation bar at the top (file `app/templates/base.html'):

```
    <div>Microblog:
        <a href="{{ url_for('index') }}">Home</a>
        {% if g.user.is_authenticated() %}
        | <a href="{{ url_for('user', nickname=g.user.nickname) }}">Your
Profile</a>
        | <a href="{{ url_for('logout') }}">Logout</a>
        {% endif %}
    </div>
```

Note how in the `url_for` function we have added the required `nickname` argument.

Give the application a try now. Clicking on the `Your Profile` link at the top should take you to your user page. Since we don't have any links that will direct you to an arbitrary user profile page you will need to type the URL by hand if you want to see someone else. For example, you would type `http://localhost:5000/user/miguel` to see the profile of user `miguel`.

Avatars

I'm sure you agree that our profile pages are pretty boring. To make them a bit more interesting, let's add user avatars.

Instead of having to deal with a possibly large collection of uploaded images in our server, we will rely on the Gravatar service to provide our user avatars.

Since returning an avatar is a user related task, we will be putting it in the `User` class (file `app/models.py`):

```
from hashlib import md5
# ...
class User(db.Model):
    # ...
    def avatar(self, size):
        return 'http://www.gravatar.com/avatar/%s?d=mm&s=%d' %
(md5(self.email.encode('utf-8')).hexdigest(), size)
```

The `avatar` method of `User` returns the URL of the user's avatar image, scaled to the requested size in pixels.

Turns out with the Gravatar service this is really easy to do. You just need to create an md5 hash of the user email and then incorporate it into the specially crafted URL that you see above. After the md5 of the email you can provide a number of options to customize the avatar. The `d=mm` determines what placeholder image is returned when a user does not have an Gravatar account. The `mm` option returns the "mystery man" image, a gray silhouette of a person. The `s=N` option requests the avatar scaled to the given size in pixels.

The Gravatar's website has documentation for the avatar URL.

Now that our `User` class knows how to return avatar images, we can incorporate them into our profile page layout (file `app/templates/user.html`):

```
<!-- extend base layout -->
{% extends "base.html" %}

{% block content %}
  <table>
      <tr valign="top">
          <td><img src="{{ user.avatar(128) }}"></td>
          <td><h1>User: {{ user.nickname }}</h1></td>
      </tr>
  </table>
  <hr>
  {% for post in posts %}
  <p>
    {{ post.author.nickname }} says: <b>{{ post.body }}</b>
  </p>
  {% endfor %}
{% endblock %}
```

The nice thing about making the `User` class responsible for returning avatars is that if some day we decide Gravatar avatars are not what we want, we just rewrite the `avatar` method to return different URLs (even ones that points to our own web server, if we decide we want to host our own avatars), and all our templates will start showing the new avatars automatically.

We have added the user avatar to the top of the profile page, but at the bottom of the page we have posts, and those could have a little avatar as well. For the user profile page we will of course be showing the same avatar for all the posts, but then when we move this functionality to the main page we will have each post decorated with the author's avatar, and that will look really nice.

To show avatars for the posts we just need to make a small change in our template (file `app/templates/user.html'):

```
<!-- extend base layout -->
{% extends "base.html" %}

{% block content %}
  <table>
```

```
      <tr valign="top">
          <td><img src="{{ user.avatar(128) }}"></td>
          <td><h1>User: {{ user.nickname }}</h1></td>
      </tr>
  </table>
  <hr>
  {% for post in posts %}
  <table>
      <tr valign="top">
          <td><img
src="{{ post.author.avatar(50) }}"></td><td><i>{{ post.author.nickname }}
says:</i><br>{{ post.body }}</td>
      </tr>
  </table>
  {% endfor %}
{% endblock %}
```

Here is how our profile page looks at this point:

Reusing at the sub-template level

We designed the user profile page so that it displays the posts written by the user. Our index page also displays posts, this time of any user. So now we have two templates that will need to display posts made by users. We could just copy/paste the portion of the template that deals with the rendering of a post, but that is really not ideal, because when we decide to change the layout of a post we'll have to remember to go update all the templates that have posts in them.

Instead, we are going to make a sub-template that just renders a post, then we'll include it in all the templates that need it.

To start, we create a post sub-template, which is really nothing more than a regular template. We do so by simply extracting the HTML for a post from our user template (file `/app/templates/post.html`):

```
<table>
    <tr valign="top">
        <td><img
src="{{ post.author.avatar(50) }}"></td><td><i>{{ post.author.nickname }}
says:</i><br>{{ post.body }}</td>
    </tr>
</table>
```

And then we invoke this sub-template from our user template using Jinja2's `include` command (file `app/templates/user.html`):

```
<!-- extend base layout -->
{% extends "base.html" %}

{% block content %}
  <table>
```

```
      <tr valign="top">
          <td><img src="{{ user.avatar(128) }}"></td>
          <td><h1>User: {{ user.nickname }}</h1></td>
      </tr>
  </table>
  <hr>
  {% for post in posts %}
      {% include 'post.html' %}
  {% endfor %}
{% endblock %}
```

Once we have a fully functioning index page we will invoke this same sub-template from over there, but we aren't quite ready to do that, we'll leave that for a future chapter of this tutorial.

More interesting profiles

While we now have a nice profile page, we don't really have much information to show on it. Users like to tell a bit about them on these pages, so we'll let them write something about themselves that we can show here. We will also keep track of what was the last time each user accessed the site, so that we can also show that on their profile page.

To add these things we have to start by modifying our database. More specifically, we need to add two new fields to our `User` class (file `app/models.py`):

```
class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    nickname = db.Column(db.String(64), index=True, unique=True)
    email = db.Column(db.String(120), index=True, unique=True)
    posts = db.relationship('Post', backref='author', lazy='dynamic')
    about_me = db.Column(db.String(140))
    last_seen = db.Column(db.DateTime)
```

Every time we modify the database we have to generate a new migration. Remember that in the database chapter we went through the pain of setting up a database migration system. We can see the fruits of that effort now. To add these two new fields to our database we just do this:

```
$ ./db_migrate.py
```

To which our script will respond:

```
New migration saved as db_repository/versions/003_migration.py
Current database version: 3
```

And our two new fields are now in our database. Remember that if you are on Windows the way to invoke this script is different.

If we did not have migration support you would have needed to edit your database manually, or worse, delete it and recreate it from scratch.

Next, let's modify our profile page template to show these fields (file `app/templates/user.html`):

```
<!-- extend base layout -->
{% extends "base.html" %}

{% block content %}
  <table>
    <tr valign="top">
      <td><img src="{{user.avatar(128)}}"></td>
      <td>
        <h1>User: {{user.nickname}}</h1>
          {% if user.about_me %}<p>{{ user.about_me }}</p>{% endif %}
          {% if user.last_seen %}<p><i>Last seen on: {{ user.last_seen }}</i></p>{%
endif %}
      </td>
    </tr>
  </table>
  <hr>
  {% for post in posts %}
    {% include 'post.html' %}
  {% endfor %}
{% endblock %}
```

Note we make use of Jinja2's conditionals to show these fields, because we only want to show them if they are set (at this point these two new fields are empty for all users, so nothing will show).

The `last_seen` field is pretty easy to support. Remember that in a previous chapter we created a `before_request` handler, to register the logged in user with the `flask.g` global, as `g.user`. That is the perfect time to update our database with the last access time for a user (file `app/views.py`):

```
from datetime import datetime
# ...
@app.before_request
def before_request():
    g.user = current_user
    if g.user.is_authenticated():
        g.user.last_seen = datetime.utcnow()
        db.session.add(g.user)
        db.session.commit()
```

If you log in to your profile page again the last seen time will now display, and each time you refresh the page the time will update as well, because each time the browser makes a request the `before_request` handler will update the time in the database.

Note that we are writing the time in the standard UTC timezone. We discussed this in a previous chapter, we will write all timestamps in UTC so that they are consistent. That has the undesired side effect that the time displayed in the user profile page is also in UTC. We will fix this in a future chapter that will be dedicated to date and time handling.

To display the user's about me information we have to give them a place to enter it, and the proper place for this is in the edit profile page.

Editing the profile information

Adding a profile form is surprisingly easy. We start by creating the web form (file `app/forms.py`):

```
from flask.ext.wtf import Form
from wtforms import StringField, BooleanField, TextAreaField
from wtforms.validators import DataRequired, Length

class EditForm(Form):
    nickname = StringField('nickname', validators=[DataRequired()])
    about_me = TextAreaField('about_me', validators=[Length(min=0, max=140)])
```

Then the view template (file `app/templates/edit.html`):

```
<!-- extend base layout -->
{% extends "base.html" %}

{% block content %}
  <h1>Edit Your Profile</h1>
  <form action="" method="post" name="edit">
      {{form.hidden_tag()}}
      <table>
          <tr>
              <td>Your nickname:</td>
              <td>{{ form.nickname(size=24) }}</td>
          </tr>
          <tr>
              <td>About yourself:</td>
              <td>{{ form.about_me(cols=32, rows=4) }}</td>
          </tr>
          <tr>
              <td></td>
              <td><input type="submit" value="Save Changes"></td>
          </tr>
      </table>
  </form>
{% endblock %}
```

And finally we write the view function (file `app/views.py`):

```
from forms import LoginForm, EditForm

@app.route('/edit', methods=['GET', 'POST'])
@login_required
def edit():
    form = EditForm()
    if form.validate_on_submit():
        g.user.nickname = form.nickname.data
        g.user.about_me = form.about_me.data
        db.session.add(g.user)
        db.session.commit()
        flash('Your changes have been saved.')
        return redirect(url_for('edit'))
    else:
        form.nickname.data = g.user.nickname
        form.about_me.data = g.user.about_me
    return render_template('edit.html', form=form)
```

To make this page easy to reach, we also add a link to it from the user profile page (file `app/templates/user.html`):

```
<!-- extend base layout -->
{% extends "base.html" %}

{% block content %}
  <table>
      <tr valign="top">
          <td><img src="{{ user.avatar(128) }}"></td>
          <td>
              <h1>User: {{user.nickname}}</h1>
              {% if user.about_me %}<p>{{ user.about_me }}</p>{% endif %}
              {% if user.last_seen %}<p><i>Last seen on:
{{ user.last_seen }}</i></p>{% endif %}
              {% if user.id == g.user.id %}<p><a
href="{{ url_for('edit') }}">Edit</a></p>{% endif %}
          </td>
      </tr>
  </table>
  <hr>
  {% for post in posts %}
      {% include 'post.html' %}
  {% endfor %}
{% endblock %}
```

Pay attention to the clever conditional we are using to make sure that the Edit link appears when you are viewing your own profile, but not when you are viewing someone else's.

Below is a new screenshot of the user profile page, with all the additions, and with some "about me" words written:

Final words... and your homework!

So it seems we are pretty much done with user profiles, right? We sort of are, but we have a nasty bug that we need to fix.

Can you find it?

Here is a clue. We have introduced this bug in the previous chapter of the tutorial, when we were looking at the login system. And today we have written a new piece of code that has the same bug.

Think about it, and if you know what the problem is feel free to show off in the comments below. I will explain the bug, and how to fix it properly in the next chapter.

As always, here is the download link for the complete application with today's additions:

Download microblog-0.6.zip.

Remember that I'm not including a database in the archive. If you have a database from the previous

chapter, just put it in the correct location and then run `db_upgrade.py` to upgrade it. If you don't have a previous database then call `db_create.py` to make a brand new one.

Thank you for following my tutorial. I hope to see you again in the next installment.

Miguel

Part VII: Unit Testing

Recap

In the previous chapters of this tutorial we were concentrating in adding functionality to our little application, a step at a time. By now we have a database enabled application that can register users, log them in and out and let them view and edit their profiles.

In this session we are not going to add any new features to our application. Instead, we are going to find ways to add robustness to the code that we have already written, and we will also create a testing framework that will help us prevent failures and regressions in the future.

Let's find a bug

I mentioned at the end of the last chapter that I have intentionally introduced a bug in the application. Let me describe what the bug is, then we will use it to see what happens to our application when it does not work as expected.

The problem in the application is that there is no effort to keep the nicknames of our users unique. The initial nickname of a user is chosen automatically by the application. If the OpenID provider provides a nickname for the user then we will just use it. If not we will use the username part of the email address as nickname. If we get two users with the same nickname then the second one will not be able to register. To make matters worse, in the profile edit form we let users change their nicknames to whatever they want, and again there is no effort to avoid name collisions.

We will address these problems later, after we analyze how the application behaves when an error occurs.

Flask debugging support

So let's see what happens when we trigger our bug.

Let's start by creating a brand new database. On Linux:

```
rm app.db
./db_create.py
```

or on Windows:

```
del app.db
flask/Scripts/python db_create.py
```

You need two OpenID accounts to reproduce this bug, ideally from different providers, so that their cookies don't make this more complicated. Follow these steps to create a nickname collision:

- login with your first account
- go to the edit profile page and change the nickname to 'dup'
- logout
- login with your second account
- go to the edit profile page and change the nickname to 'dup'

Oops! We've got an exception from sqlalchemy. The text of the error reads:

```
sqlalchemy.exc.IntegrityError
IntegrityError: (IntegrityError) column nickname is not unique u'UPDATE user SET
nickname=?, about_me=? WHERE user.id = ?' (u'dup', u'', 2)
```

What follows after the error is a stack trace of the error, and actually it is a pretty nice one, where you can go to any frame and inspect source code or even evaluate expressions right in the browser.

The error is pretty clear, we tried to insert a duplicated nickname in the database. The database model had a `unique` constrain on the `nickname` field, so this is an invalid operation.

In addition to the actual error, we have a secondary problem in our hands. If a user inadvertently causes an error in our application (this one or any other that causes an exception) it will be him or her that gets the error with the revealing error message and the stack trace, not us. While this is a fantastic feature while we are developing, it is something we definitely do not want our users to ever see.

All this time we have been running our application in *debug mode*. The debug mode is enabled when the application starts, by passing a `debug=True` argument to the `run` method. This is how we coded our `run.py` start-up script.

When we are developing the application this is convenient, but we need to make sure it is turned off when we run our application in *production mode*. Let's just create another starter script that runs with debugging disabled (file `runp.py`):

```
#!flask/bin/python
from app import app
app.run(debug=False)
```

Now restart the application with:

```
./runp.py
```

And now try again to rename the nickname on the second account to 'dup'.

This time we do not get an error. Instead, we get an HTTP error code 500, which is Internal Server Error. Not a great looking error, but at least we are not exposing any details of our application to strangers. The error 500 page is generated by Flask when debugging is off and an unhandled exception occurs.

While this is better, we are now having two new issues. First a cosmetic one: the default error 500 page is ugly. The second problem is much more important. With things as they are we would never know

when and if a user experiences a failure in our application because when debugging is turned off application failures are silently dismissed. Luckily there are easy ways to address both problems.

Custom HTTP error handlers

Flask provides a mechanism for an application to install its own error pages. As an example, let's define custom error pages for the HTTP errors 404 and 500, the two most common ones. Defining pages for other errors works in the same way.

To declare a custom error handler the `errorhandler` decorator is used (file `app/views.py`):

```
@app.errorhandler(404)
def not_found_error(error):
    return render_template('404.html'), 404

@app.errorhandler(500)
def internal_error(error):
    db.session.rollback()
    return render_template('500.html'), 500
```

Not much to talk about for these, as they are almost self-explanatory. The only interesting bit is the `rollback` statement in the error 500 handler. This is necessary because this function will be called as a result of an exception. If the exception was triggered by a database error then the database session is going to arrive in an invalid state, so we have to roll it back in case a working session is needed for the rendering of the template for the 500 error.

Here is the template for the 404 error:

```
<!-- extend base layout -->
{% extends "base.html" %}

{% block content %}
  <h1>File Not Found</h1>
  <p><a href="{{ url_for('index') }}">Back</a></p>
{% endblock %}
```

And here is the one for the 500 error:

```
<!-- extend base layout -->
{% extends "base.html" %}

{% block content %}
  <h1>An unexpected error has occurred</h1>
  <p>The administrator has been notified. Sorry for the inconvenience!</p>
  <p><a href="{{ url_for('index') }}">Back</a></p>
{% endblock %}
```

Note that in both cases we continue to use our `base.html` layout, so that the error page has the look and feel of the application.

Sending errors via email

To address our second problem we are going to configure two reporting mechanisms for application errors. The first of them is to have the application send us an email each time an error occurs.

Before we get into this let's configure an email server and an administrator list in our application (file `config.py`):

```
# mail server settings
MAIL_SERVER = 'localhost'
MAIL_PORT = 25
MAIL_USERNAME = None
MAIL_PASSWORD = None

# administrator list
ADMINS = ['you@example.com']
```

Of course it will be up to you to change these to what makes sense.

Flask uses the regular Python `logging` module, so setting up an email when there is an exception is pretty easy (file `app/__init__.py`):

```
from config import basedir, ADMINS, MAIL_SERVER, MAIL_PORT, MAIL_USERNAME,
MAIL_PASSWORD

if not app.debug:
    import logging
    from logging.handlers import SMTPHandler
    credentials = None
    if MAIL_USERNAME or MAIL_PASSWORD:
        credentials = (MAIL_USERNAME, MAIL_PASSWORD)
    mail_handler = SMTPHandler((MAIL_SERVER, MAIL_PORT), 'no-reply@' + MAIL_SERVER,
ADMINS, 'microblog failure', credentials)
    mail_handler.setLevel(logging.ERROR)
    app.logger.addHandler(mail_handler)
```

Note that we are only enabling the emails when we run without debugging.

Testing this on a development PC that does not have an email server is easy, thanks to Python's SMTP debugging server. Just open a new console window (command prompt for Windows users) and run the following to start a fake email server:

```
python -m smtpd -n -c DebuggingServer localhost:25
```

When this is running, the emails sent by the application will be received and displayed in the console window.

Logging to a file

Receiving errors via email is nice, but sometimes this isn't enough. There are some failure conditions that do not end in an exception and aren't a major problem, yet we may want to keep track of them in a log in case we need to do some debugging.

For this reason, we are also going to maintain a log file for the application.

Enabling file logging is similar to the email logging (file `app/__init__.py`):

```
if not app.debug:
    import logging
    from logging.handlers import RotatingFileHandler
    file_handler = RotatingFileHandler('tmp/microblog.log', 'a', 1 * 1024 * 1024,
10)
    file_handler.setFormatter(logging.Formatter('%(asctime)s %(levelname)s: %
(message)s [in %(pathname)s:%(lineno)d]'))
    app.logger.setLevel(logging.INFO)
    file_handler.setLevel(logging.INFO)
    app.logger.addHandler(file_handler)
    app.logger.info('microblog startup')
```

The log file will go to our `tmp` directory, with name `microblog.log`. We are using the `RotatingFileHandler` so that there is a limit to the amount of logs that are generated. In this case we are limiting the size of a log file to one megabyte, and we will keep the last ten log files as backups.

The `logging.Formatter` class provides custom formatting for the log messages. Since these messages are going to a file, we want them to have as much information as possible, so we write a timestamp, the logging level and the file and line number where the message originated in addition to the log message and the stack trace.

To make the logging more useful, we are lowering the logging level, both in the app logger and the file logger handler, as this will give us the opportunity to write useful messages to the log without having to call them errors. As an example, we start by logging the application start up as an informational level. From now on, each time you start the application without debugging the log will record the event.

While we don't have a lot of need for a logger at this time, debugging a web server that is online and in use is very difficult. Logging messages to a file is an extremely useful tool in diagnosing and locating issues, so we are now all ready to go should we need to use this feature.

The bug fix

Let's fix our `nickname` duplication bug.

As discussed earlier, there are two places that are currently not handling duplicates. The first is in the `after_login` handler for Flask-Login. This is called when a user successfully logs in to the system and we need to create a new User instance. Here is the affected snippet of code, with the fix in it (file `app/views.py`):

```
    if user is None:
        nickname = resp.nickname
        if nickname is None or nickname == "":
            nickname = resp.email.split('@')[0]
        nickname = User.make_unique_nickname(nickname)
        user = User(nickname = nickname, email = resp.email)
        db.session.add(user)
        db.session.commit()
```

The way we solve the problem is by letting the User class pick a unique name for us. This is what the new `make_unique_nickname` method does (file `app/models.py`):

```
    class User(db.Model):
    # ...
    @staticmethod
    def make_unique_nickname(nickname):
        if User.query.filter_by(nickname=nickname).first() is None:
            return nickname
        version = 2
        while True:
            new_nickname = nickname + str(version)
            if User.query.filter_by(nickname=new_nickname).first() is None:
                break
            version += 1
        return new_nickname
    # ...
```

This method simply adds a counter to the requested nickname until a unique name is found. For example, if the username "miguel" exists, the method will suggest "miguel2", but if that also exists it will go to "miguel3" and so on. Note that we coded the method as a static method, since it this operation does not apply to any particular instance of the class.

The second place where we have problems with duplicate nicknames is the view function for the edit profile page. This one is a little tricker to handle, because it is the user choosing the nickname. The correct thing to do here is to not accept a duplicated nickname and let the user enter another one. We will address this by adding custom validation to the nickname form field. If the user enters an invalid nickname we'll just fail the validation for the field, and that will send the user back to the edit profile page. To add our validation we just override the form's `validate` method (file `app/forms.py`):

```
from app.models import User

class EditForm(Form):
    nickname = StringField('nickname', validators=[DataRequired()])
    about_me = TextAreaField('about_me', validators=[Length(min=0, max=140)])

    def __init__(self, original_nickname, *args, **kwargs):
        Form.__init__(self, *args, **kwargs)
        self.original_nickname = original_nickname

    def validate(self):
        if not Form.validate(self):
            return False
        if self.nickname.data == self.original_nickname:
            return True
        user = User.query.filter_by(nickname=self.nickname.data).first()
        if user != None:
            self.nickname.errors.append('This nickname is already in use. Please
choose another one.')
            return False
        return True
```

The form constructor now takes a new argument `original_nickname`. The `validate` method

uses it to determine if the nickname has changed or not. If it hasn't changed then it accepts it. If it has changed, then it makes sure the new nickname does not exist in the database.

Next we add the new constructor argument to the view function:

```
@app.route('/edit', methods=['GET', 'POST'])
@login_required
def edit():
    form = EditForm(g.user.nickname)
    # ...
```

To complete this change we have to enable field errors to show in our template for the form (file `app/templates/edit.html`):

```
        <td>Your nickname:</td>
        <td>
            {{ form.nickname(size=24) }}
            {% for error in form.errors.nickname %}
            <br><span style="color: red;">[{{ error }}]</span>
            {% endfor %}
        </td>
```

Now the bug is fixed and duplicates will be prevented... except when they are not. We still have a potential problem with concurrent access to the database by two or more threads or processes, but this will be the topic of a future article.

At this point you can try again to select a duplicated name to see how the form nicely handles the error.

Unit testing framework

To close this session on testing, let's talk about automated testing a bit.

As the application grows in size it gets more and more difficult to ensure that code changes don't break existing functionality.

The traditional approach to prevent regressions is a very good one. You write tests that exercise all the different features of the application. Each test runs a focused part and verifies that the result obtained is the expected one. The tests are executed periodically to make sure that the application works as expected. When the test *coverage* is large you can have confidence that modifications and additions do not affect the application in a bad way just by running the tests.

We will now build a very simple testing framework using Python's `unittest` module (file `tests.py`):

```
#!flask/bin/python
import os
import unittest

from config import basedir
from app import app, db
from app.models import User
```

```
class TestCase(unittest.TestCase):
    def setUp(self):
        app.config['TESTING'] = True
        app.config['WTF_CSRF_ENABLED'] = False
        app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///' +
os.path.join(basedir, 'test.db')
        self.app = app.test_client()
        db.create_all()

    def tearDown(self):
        db.session.remove()
        db.drop_all()

    def test_avatar(self):
        u = User(nickname='john', email='john@example.com')
        avatar = u.avatar(128)
        expected =
'http://www.gravatar.com/avatar/d4c74594d841139328695756648b6bd6'
        assert avatar[0:len(expected)] == expected

    def test_make_unique_nickname(self):
        u = User(nickname='john', email='john@example.com')
        db.session.add(u)
        db.session.commit()
        nickname = User.make_unique_nickname('john')
        assert nickname != 'john'
        u = User(nickname=nickname, email='susan@example.com')
        db.session.add(u)
        db.session.commit()
        nickname2 = User.make_unique_nickname('john')
        assert nickname2 != 'john'
        assert nickname2 != nickname

if __name__ == '__main__':
    unittest.main()
```

Discussing the `unittest` module is outside the scope of this article. Let's just say that class `TestCase` holds our tests. The `setUp` and `tearDown` methods are special, these are run before and after each test respectively. A more complex setup could include several groups of tests each represented by a `unittest.TestCase` subclass, and each group then would have independent `setUp` and `tearDown` methods.

These particular `setUp` and `tearDown` methods are pretty generic. In `setUp` the configuration is edited a bit. For instance, we want the testing database to be different that the main database. In `tearDown` we just reset the database contents.

Tests are implemented as methods. A test is supposed to run some function of the application that has a known outcome, and should assert if the result is different than the expected one.

So far we have two tests in the testing framework. The first one verifies that the Gravatar avatar URLs from the previous article are generated correctly. Note how the expected avatar is hardcoded in the test and checked against the one returned by the `User` class.

The second test verifies the `make_unique_nickname` method we just wrote, also in the `User`

class. This test is a bit more elaborate, it creates a new user and writes it to the database, then ensures the same name is not allowed as a unique name. It then creates a second user with the suggested unique name and tries one more time to request the first nickname. The expected result for this second part is to get a suggested nickname that is different from the previous two.

To run the test suite you just run the `tests.py` script:

```
python tests.py
```

If there are any errors, you will get a report in the console.

Final words

This ends today's discussion of debugging, errors and testing. I hope you found this article useful.

As always, if you have any comments please write below.

The code of the microblog application update with today's changes is available below for download:

Download microblog-0.7.zip.

As always, the flask virtual environment and the database are not included. See previous articles for instructions on how to generate them.

I hope to see you again in the next installment of this series.

Thank you for reading!

Miguel

Part VIII: Followers, Contacts And Friends

Recap

Our `microblog` application has been growing little by little, and by now we have touched on most of the topics that are required to complete the application.

Today we are going to work on our database some more. Each user of our application needs to be able to select which other users he or she wants to follow, so our database must be able to keep track of who is following who. All social applications have this feature in some form. Some call it Contacts, others Connections, Friends, Buddies or Followers. Other sites use this same idea to implement Allowed and Ignored user lists. We will call them Followers, but the implementation is the same regardless of the name.

Design of the 'Follower' feature

Before we start coding, let's think about the functionality that we want to obtain from this feature.

Let's start with the most obvious one. We want our users to easily maintain a list of *followed* people.

Looking at it from the other side, for each user we want to know the list of its *followers*.

We also want to have a way to query if a user *is following* or *is followed* by another user.

Users will click a "follow" link in the profile page of any other user to begin following that user. Likewise, they'll click a "unfollow" link to stop following a user.

The final requirement is that given a user we can easily query the database to obtain all the posts that belong to the followed users.

So, if you thought this was going to be a quick and easy article, think again!

Database relationships

We said we wanted to have all users to have "followed" and "followers" lists. Unfortunately, a relational database does not have a `list` type, all we have are tables with records and relationships between records.

We already have a table in our database to represent users, so what's left is to come up with the proper relationship type that can model the follower/followed link. This is a good time to review the three database relationship types:

One-to-many

We have already seen a one-to-many relationship in the previous database article. Here is the diagram:

The two entities associated with this relationship are `users` and `posts`. We say that a user has *many* posts, and a post has *one* user. The relationship is represented in the database with the use of a *foreign key* on the "many" side. In the above example the foreign key is the `user_id` field added to the `posts` table. This field links each post to the record of its author in the user table.

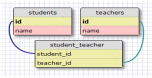
It is pretty clear that the `user_id` field provides direct access to the author of a given post, but what about the reverse? For the relationship to be useful we should be able to get the list of posts written by a given user. Turns out the `user_id` field in the `posts` table is enough to answer this question, as databases have indexes that allow for efficient queries such us "retrieve all posts that have a user_id of X".

Many-to-many

A many-to-many relationship is a bit more complex. As an example, consider a database that has `students` and `teachers`. We can say that a student has *many* teachers, and a teacher has *many* students. It's like two overlapped one-to-many relationships from both ends.

For a relationship of this type we should be able to query the database and obtain the list of teachers that teach a student, and the list of students in a teacher's class. Turns out this is pretty tricky to represent, it cannot be done by adding foreign keys to the existing tables.

The representation of a many-to-many relationship requires the use of an auxiliary table called an *association table*. Here is how the database would look for the students and teachers example:



While it may not seem straightforward, the association table with its two foreign keys is able to efficiently answer many types of queries, such as:

- Who are the teachers of student S?
- Who are the students of teacher T?
- How many students does teacher T have?
- How many teachers does student S have?
- Is teacher T teaching student S?
- Is student S in teacher T's class?

One-to-one

A one-to-one relationship is a special case of a one-to-many. The representation is similar, but a constrain is added to the database to prevent the "many" side to have more than one link.

While there are cases in which this type of relationship is useful, it isn't as common as the other two

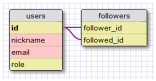
types, since any time one record in a table maps to one record in another table it can be argued that it may make sense for these two tables to be merged into one.

Representing followers and followed

From the above relationships we can easily determine that the proper data model is the many-to-many relationship, because a user follows *many* users, and a user has *many* followers. But there is a twist. We want to represent users following other users, so we just have users. So what should we use as the second entity of the many-to-many relationship?

Well, the second entity of the relationship is also the users. A relationship in which instances of an entity are linked to other instances of the same entity is called a *self-referential relationship*, and that is exactly what we need here.

Here is a diagram of our many-to-many relationship:



The `followers` table is our association table. The foreign keys are both pointing to the user table, since we are linking users to users. Each record in this table represents one link between a follower user and a followed user. Like the students and teachers example, a setup like this one allows our database to answer all the questions about followed and followers that we will need. Pretty neat.

Database model

The changes to our database model aren't that big. We start by adding the `followers` table (file `app/models.py`):

```
followers = db.Table('followers',
    db.Column('follower_id', db.Integer, db.ForeignKey('user.id')),
    db.Column('followed_id', db.Integer, db.ForeignKey('user.id'))
)
```

This is a direct translation of the association table from our diagram. Note that we are not declaring this table as a model like we did for `users` and `posts`. Since this is an auxiliary table that has no data other than the foreign keys, we use the lower level APIs in flask-sqlalchemy to create the table without an associated model.

Next we define the many-to-many relationship in the `users` table (file `app/models.py`):

```
class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    nickname = db.Column(db.String(64), index=True, unique=True)
    email = db.Column(db.String(120), index=True, unique=True)
    posts = db.relationship('Post', backref='author', lazy='dynamic')
    about_me = db.Column(db.String(140))
    last_seen = db.Column(db.DateTime)
    followed = db.relationship('User',
```

```
                secondary=followers,
                primaryjoin=(followers.c.follower_id == id),
                secondaryjoin=(followers.c.followed_id == id),
                backref=db.backref('followers', lazy='dynamic'),
                lazy='dynamic')
```

The setup of the relationship is non-trivial and requires some explanation. Like we did for the one-to-many relationship in the previous article, we are using the `db.relationship` function to define the relationship. We will be linking `User` instances to other `User` instances, so as a convention let's say that for a pair of linked users in this relationship the left side user is following the right side user. We define the relationship as seen from the left side entity with the name `followed`, because when we query this relationship from the left side we will get the list of followed users. Let's examine all the arguments to the `db.relationship()` call one by one:

- `'User'` is the right side entity that is in this relationship (the left side entity is the parent class). Since we are defining a self-referential relationship we use the same class on both sides.
- `secondary` indicates the association table that is used for this relationship.
- `primaryjoin` indicates the condition that links the left side entity (the follower user) with the association table. Note that because the `followers` table is not a model there is a slightly odd syntax required to get to the field name.
- `secondaryjoin` indicates the condition that links the right side entity (the followed user) with the association table.
- `backref` defines how this relationship will be accessed from the right side entity. We said that for a given user the query named `followed` returns all the right side users that have the target user on the left side. The back reference will be called `followers` and will return all the left side users that are linked to the target user in the right side. The additional `lazy` argument indicates the execution mode for this query. A mode of `dynamic` sets up the query to not run until specifically requested. This is useful for performance reasons, and also because we will be able to take this query and modify it before it executes. More about this later.
- `lazy` is similar to the parameter of the same name in the `backref`, but this one applies to the regular query instead of the back reference.

Don't despair if this is hard to understand. We will see how to use these queries in a moment, and then everything will become clearer.

Since we have made updates to the database, we now have to generate a new migration:

```
$ ./db_migrate.py
```

And with this we have completed the database changes. But we have quite a bit of coding left to do.

Adding and removing 'follows'

To promote reusability, we will implement the `follow` and `unfollow` functionality in the `User`

model instead of doing it directly in view functions. That way we can use this feature for the actual application (invoking it from the view functions) and also from our unit testing framework. As a matter of principle, it is always best to move the logic of our application away from view functions and into models, because that simplifies the testing. You want to have your view functions be as simple as possible, because those are harder to test in an automated way.

Below is the code to add and remove relationships, defined as methods of the `User` model (file `app/models.py`):

```
class User(db.Model):
    #...
    def follow(self, user):
        if not self.is_following(user):
            self.followed.append(user)
            return self

    def unfollow(self, user):
        if self.is_following(user):
            self.followed.remove(user)
            return self

    def is_following(self, user):
        return self.followed.filter(followers.c.followed_id == user.id).count() > 0
```

These methods are amazingly simple, thanks to the power of sqlalchemy who does a lot of work under the covers. We just add or remove items from the `followed` relationship and sqlalchemy takes care of managing the association table for us.

The `follow` and `unfollow` methods are defined so that they return an object when they succeed or `None` when they fail. When an object is returned, this object has to be added to the database session and committed.

The `is_following` method does a lot in its single line of code. We are taking the `followed` relationship query, which returns all the `(follower, followed)` pairs that have our user as the follower, and we filter it by the followed user. This is possible because the `followed` relationship has a lazy mode of `dynamic`, so instead of being the result of the query, this is the actual query object, before execution.

The return from the `filter` call is the modified query, still without having executed. So we then call `count()` on this query, and now the query will execute and return the number of records found. If we get one, then we know a link between these two uses is already present. If we get none then we know a link does not exist.

Testing

Let's write a test for our unit testing framework that exercises all that we have built so far (file `tests.py`):

```
class TestCase(unittest.TestCase):
    #...
    def test_follow(self):
        u1 = User(nickname='john', email='john@example.com')
        u2 = User(nickname='susan', email='susan@example.com')
        db.session.add(u1)
        db.session.add(u2)
        db.session.commit()
        assert u1.unfollow(u2) is None
        u = u1.follow(u2)
        db.session.add(u)
        db.session.commit()
        assert u1.follow(u2) is None
        assert u1.is_following(u2)
        assert u1.followed.count() == 1
        assert u1.followed.first().nickname == 'susan'
        assert u2.followers.count() == 1
        assert u2.followers.first().nickname == 'john'
        u = u1.unfollow(u2)
        assert u is not None
        db.session.add(u)
        db.session.commit()
        assert not u1.is_following(u2)
        assert u1.followed.count() == 0
        assert u2.followers.count() == 0
```

After adding this test to the testing framework we can run the entire test suite with the following command:

```
./tests.py
```

And if everything works it should say that all our tests pass.

Database queries

Our current database model supports most of the requirements we listed at the start. The one we are missing is, in fact, the hardest. Our index page will show the posts written by all the people that are followed by the logged in user, so we need a query that returns all these posts.

The most obvious solution is to run a query that gives us the list of followed users, which we can already do. Then for each of these returned users we run a query to get the posts. Once we have all the posts we merge them into a single list and sort them by date. Sounds good? Well, not really.

This approach has a couple of problems. What happens if a user is following a thousand people? We need to execute a thousand database queries just to collect all the posts. And now we have the thousand lists in memory that we need to merge and sort. As a secondary problem, consider that our index page will (eventually) have pagination implemented, so we will not display all the available posts but just the first, say, fifty, with links to get the next or previous set of fifty. If we are going to display posts sorted by their date, how can we know which posts are the most recent fifty of all followed users combined, unless we get all the posts and sort them first? This is actually an awful solution that does not scale well.

While this collecting and sorting work needs to be done somehow, us doing it results in a very inefficient process. This kind of work is what relational databases excel at. The database has indexes that allow it to perform the queries and the sorting in a much more efficient way that we can possibly do from our side.

So what we really want is to come up with a single database query that expresses what information we want to get, and then we let the database figure out what is the most efficient way to obtain the data for us.

To end the mystery, here is the query that achieves this. Unfortunately it is yet another heavily loaded one liner that we will add to the `User` model (file `app/models.py`):

```
class User(db.Model):
    #...
    def followed_posts(self):
        return Post.query.join(followers, (followers.c.followed_id ==
Post.user_id)).filter(followers.c.follower_id ==
self.id).order_by(Post.timestamp.desc())
```

Let's try to decipher this query one piece at a time. There are three parts: the *join*, the *filter* and the *order_by*.

Joins

To understand what a join operation does, let's look at an example. Let's assume we have a `User` table with the following contents:

User

id	nickname
1	john
2	susan
3	mary
4	david

There are some extra fields in the table that are not shown above just to simplify the example.

Let's say that our `followers` association table says that user "john" is following users "susan" and "david", user "susan" is following "mary" and user "mary" is following "david". The data that represents the above is this:

followers

follower_id	followed_id
1	2
1	4
2	3
3	4

And finally, our `Post` table contains one post from each user:

	Post	
id	**text**	**user_id**
1	post from susan	2
2	post from mary	3
3	post from david	4
4	post from john	1

Here again there are some fields that are omitted to keep the example simple.

Below is the join portion of our query, isolated from the rest of the query:

```
Post.query.join(followers,
    (followers.c.followed_id == Post.user_id))
```

The `join` operation is called on the `Post` table. There are two arguments, the first is another table, our `followers` table. The second argument to the join call is the join *condition*.

What the join operation will do with all this is create a temporary new table with data from the `Post` and `followers` table, merged according to the given condition.

In this example we want the field `followed_id` of the `followers` table to match the `user_id` field of the `Post` table.

To perform this merge, we take each record from the `Post` table (the left side of the join) and append the fields from the records in the `followers` table (the right side of the join) that match the condition. If there is no match, then that post record is removed.

The result of the join in our example results in this temporary table:

	Post		followers	
id	**text**	**user_id**	**follower_id**	**followed_id**
1	post from susan	2	1	2
2	post from mary	3	2	3
3	post from david	4	1	4
3	post from david	4	3	4

Note how the post with user_id=1 was removed from the join, because there is no record in the `followers` table that has a followed_id=1. Also note how the post with user_id=4 appears twice, because the `followers` table has two entries with a followed_id=4.

Filters

The join operation gave us a list of posts that are followed by someone, without specifying who is the follower. We are only interested in a subset of this list, we need just the posts that are followed by one specific user.

So we will filter this table by the follower user. The filter portion of the query then is:

```
filter(followers.c.follower_id == self.id)
```

Remember that the query is executed in the context of our target user, because it is a method in the User class, so `self.id` in this context is the id of the user we are interested in. With this filter we are telling the database that we want to keep just the records from the joined table that have our user as a follower. So following our example, if the user we are asking about is the one with id=1, then we would end up with yet another temporary table:

	Post		followers	
id	**text**	**user_id**	**follower_id**	**followed_id**
1	post from susan	2	1	2
3	post from david	4	1	4

And these are exactly the posts that we want!

Remember that the query was issued on the Post class, so even though we ended up with a temporary table that does not match any of our models, the result will be the posts that are included in this temporary table, without the extra columns added by the join operation.

Sorting

The final step of the process is to sort the results according to our criteria. The portion of the query that does that says:

```
order_by(Post.timestamp.desc())
```

Here we are saying that the results should be sorted by the timestamp field in descending order, so that the first result will be the most recent post.

There is only one more minor detail that we can add to improve our query. When users read their followed posts they will probably want to see their own posts inserted in the stream as well, so it would be nice to have those included in the query results.

And turns out there is a very simple way to achieve this that doesn't require any changes! We will just make sure that each user is added as a follower of him/herself in the database, and that will take care of this little problem for us.

To conclude our long discussion on queries, let's write a unit test for our query (file `tests.py`):

```
#...
from datetime import datetime, timedelta
from app.models import User, Post
#...
class TestCase(unittest.TestCase):
    #...
    def test_follow_posts(self):
        # make four users
        u1 = User(nickname='john', email='john@example.com')
        u2 = User(nickname='susan', email='susan@example.com')
        u3 = User(nickname='mary', email='mary@example.com')
        u4 = User(nickname='david', email='david@example.com')
        db.session.add(u1)
        db.session.add(u2)
```

```
        db.session.add(u3)
        db.session.add(u4)
        # make four posts
        utcnow = datetime.utcnow()
        p1 = Post(body="post from john", author=u1, timestamp=utcnow +
timedelta(seconds=1))
        p2 = Post(body="post from susan", author=u2, timestamp=utcnow +
timedelta(seconds=2))
        p3 = Post(body="post from mary", author=u3, timestamp=utcnow +
timedelta(seconds=3))
        p4 = Post(body="post from david", author=u4, timestamp=utcnow +
timedelta(seconds=4))
        db.session.add(p1)
        db.session.add(p2)
        db.session.add(p3)
        db.session.add(p4)
        db.session.commit()
        # setup the followers
        u1.follow(u1)  # john follows himself
        u1.follow(u2)  # john follows susan
        u1.follow(u4)  # john follows david
        u2.follow(u2)  # susan follows herself
        u2.follow(u3)  # susan follows mary
        u3.follow(u3)  # mary follows herself
        u3.follow(u4)  # mary follows david
        u4.follow(u4)  # david follows himself
        db.session.add(u1)
        db.session.add(u2)
        db.session.add(u3)
        db.session.add(u4)
        db.session.commit()
        # check the followed posts of each user
        f1 = u1.followed_posts().all()
        f2 = u2.followed_posts().all()
        f3 = u3.followed_posts().all()
        f4 = u4.followed_posts().all()
        assert len(f1) == 3
        assert len(f2) == 2
        assert len(f3) == 2
        assert len(f4) == 1
        assert f1 == [p4, p2, p1]
        assert f2 == [p3, p2]
        assert f3 == [p4, p3]
        assert f4 == [p4]
```

This test has a lot of setup code but the actual test is pretty short. We first check that the number of followed posts returned for each user is the expected one. Then for each user we check that the correct posts were returned and that they came in the correct order (note that we inserted the posts with timestamps that are guaranteed to always order in the same way).

Note the usage of the `followed_posts()` method. This method returns a query object, not the results. This is similar to how relationships with `lazy = 'dynamic'` work. It is always a good idea to return query objects instead of results, because that gives the caller the choice of adding more clauses to the query before it is executed.

There are several methods in the query object that trigger the query execution. We've seen that

`count()` runs the query and returns the number of results (throwing the actual results away). We have also used `first()` to return the first result and throw away the rest, if any. In this test we are using the `all()` method to get an array with all the results.

Possible improvements

We now have implemented all the required features of our 'follower' feature, but there are ways to improve our design and make it more flexible.

All the social networks that we love to hate support similar ways to connect users, but they have more options to control the sharing of information.

For example, we have not elected to support the ability to block users. This would add one more layer of complexity to our queries, since now we not only need to grab the posts of the users we follow, but we need to filter out those from users that decided to block us. How would you implement this? Simple, one more many-to-many self-referential relationship to record who's blocking who, and one more join+filter in the query that returns the followed posts.

Another popular feature in social networks is the ability to group followers into custom lists, and then sharing content only with specific groups. This is also implemented with additional relationships and added complexity to the queries.

We will not have these features in `microblog`, but if there is enough interest I would be happy to write an article on the topic. Let me know in the comments!

Tying up loose ends

We have made an impressive amount of progress today. But while we have solved all the problems related to database setup and querying, we have not enabled the new functionality through our application.

Luckily for us, there aren't any challenges in doing this. We just need to fix view functions and templates to call the new methods in the `User` model when appropriate. So let's do that before we close this session.

Being your own follower

We decided that we were going to mark all users as followers of themselves, so that they can see their own posts in their post stream.

We are going to do that at the point where users are getting their accounts setup, in the `after_login` handler for OpenID (file 'app/views.py'):

```
@oid.after_login
def after_login(resp):
    if resp.email is None or resp.email == "":
        flash('Invalid login. Please try again.')
```

```
        return redirect(url_for('login'))
    user = User.query.filter_by(email=resp.email).first()
    if user is None:
        nickname = resp.nickname
        if nickname is None or nickname == "":
            nickname = resp.email.split('@')[0]
        nickname = User.make_unique_nickname(nickname)
        user = User(nickname=nickname, email=resp.email)
        db.session.add(user)
        db.session.commit()
        # make the user follow him/herself
        db.session.add(user.follow(user))
        db.session.commit()
    remember_me = False
    if 'remember_me' in session:
        remember_me = session['remember_me']
        session.pop('remember_me', None)
    login_user(user, remember=remember_me)
    return redirect(request.args.get('next') or url_for('index'))
```

Follow and Unfollow links

Next, we will define view functions that follow and unfollow a user (file `app/views.py`):

```
@app.route('/follow/<nickname>')
@login_required
def follow(nickname):
    user = User.query.filter_by(nickname=nickname).first()
    if user is None:
        flash('User %s not found.' % nickname)
        return redirect(url_for('index'))
    if user == g.user:
        flash('You can\'t follow yourself!')
        return redirect(url_for('user', nickname=nickname))
    u = g.user.follow(user)
    if u is None:
        flash('Cannot follow ' + nickname + '.')
        return redirect(url_for('user', nickname=nickname))
    db.session.add(u)
    db.session.commit()
    flash('You are now following ' + nickname + '!')
    return redirect(url_for('user', nickname=nickname))

@app.route('/unfollow/<nickname>')
@login_required
def unfollow(nickname):
    user = User.query.filter_by(nickname=nickname).first()
    if user is None:
        flash('User %s not found.' % nickname)
        return redirect(url_for('index'))
    if user == g.user:
        flash('You can\'t unfollow yourself!')
        return redirect(url_for('user', nickname=nickname))
    u = g.user.unfollow(user)
    if u is None:
        flash('Cannot unfollow ' + nickname + '.')
        return redirect(url_for('user', nickname=nickname))
    db.session.add(u)
```

```
    db.session.commit()
    flash('You have stopped following ' + nickname + '.')
    return redirect(url_for('user', nickname=nickname))
```

These should be self-explanatory, but note how there is error checking all around, to prevent unexpected problems and try to provide a message to the user and a redirection when a problem has occurred.

Now we have the view functions, so we can hook them up. The links to follow and unfollow a user will appear in the profile page of each user (file `app/templates/user.html`):

```
<!-- extend base layout -->
{% extends "base.html" %}

{% block content %}
  <table>
      <tr valign="top">
          <td><img src="{{ user.avatar(128) }}"></td>
          <td>
              <h1>User: {{ user.nickname }}</h1>
              {% if user.about_me %}<p>{{ user.about_me }}</p>{% endif %}
              {% if user.last_seen %}<p><i>Last seen on:
{{ user.last_seen }}</i></p>{% endif %}
              <p>{{ user.followers.count() }} followers |
              {% if user.id == g.user.id %}
                  <a href="{{ url_for('edit') }}">Edit your profile</a>
              {% elif not g.user.is_following(user) %}
                  <a href="{{ url_for('follow',
nickname=user.nickname) }}">Follow</a>
              {% else %}
                  <a href="{{ url_for('unfollow',
nickname=user.nickname) }}">Unfollow</a>
              {% endif %}
              </p>
          </td>
      </tr>
  </table>
  <hr>
  {% for post in posts %}
      {% include 'post.html' %}
  {% endfor %}
{% endblock %}
```

In the line we had the "Edit" link we now show the number of followers the user has, followed by one of three possible links:

- if the user profile belongs to the logged in user, then the "Edit" link shows.
- else, if the user is not currently followed a "Follow" link shows.
- else, a "Unfollow" link shows.

At this point you can run the application, create a few users by logging in with different OpenID accounts and play with following and unfollowing users.

All that is left is to show posts of the followed users in the `index` page, but we are still missing an

important piece of the puzzle before we can do that, so this will have to wait until the next chapter.

Final words

We have implemented a core piece of our application today. The topic of database relationships and queries is a pretty complex one, so if there are any questions about the material presented above you are welcome to send your questions in the comments below.

In the next installment we will be looking at the fascinating world of pagination, and we will be finally hooking up posts from the database to our application.

For those of you that are lazy typists (or lazy copy-pasters), below is the updated `microblog` application:

Download microblog-0.8.zip.

As always, the above zip file does not include a database or the flask virtual environment. Previous articles explain how to regenerate these.

Thanks again for following my tutorial. See you next time!

Miguel

Part IX: Pagination

Recap

In the previous article in the series we've made all the database changes necessary to support the 'follower' paradigm, where users choose other users to follow.

Today we will build on what we did last time and enable our application to accept and deliver real content to its users. We are saying goodbye to the last of our fake objects today!

Submission of blog posts

Let's start with something simple. The home page should have a form for users to submit new posts.

First we define a single field form object (file `app/forms.py`):

```
class PostForm(Form):
    post = StringField('post', validators=[DataRequired()])
```

Next, we add the form to the template (file `app/templates/index.html`):

```
<!-- extend base layout -->
{% extends "base.html" %}

{% block content %}
  <h1>Hi, {{ g.user.nickname }}!</h1>
  <form action="" method="post" name="post">
      {{ form.hidden_tag() }}
      <table>
          <tr>
              <td>Say something:</td>
              <td>{{ form.post(size=30, maxlength=140) }}</td>
              <td>
              {% for error in form.post.errors %}
              <span style="color: red;">[{{ error }}]</span><br>
              {% endfor %}
              </td>
          </tr>
          <tr>
              <td></td>
              <td><input type="submit" value="Post!"></td>
              <td></td>
          </tr>
      </table>
  </form>
  {% for post in posts %}
  <p>
    {{ post.author.nickname }} says: <b>{{ post.body }}</b>
  </p>
  {% endfor %}
{% endblock %}
```

Nothing earth shattering so far, as you can see. We are simply adding yet another form, like the ones we've done before.

Last of all, the view function that ties everything together is expanded to handle the form (file `app/views.py`):

```
from forms import LoginForm, EditForm, PostForm
from models import User, Post

@app.route('/', methods=['GET', 'POST'])
@app.route('/index', methods=['GET', 'POST'])
@login_required
def index():
    form = PostForm()
    if form.validate_on_submit():
        post = Post(body=form.post.data, timestamp=datetime.utcnow(),
author=g.user)
        db.session.add(post)
        db.session.commit()
        flash('Your post is now live!')
        return redirect(url_for('index'))
    posts = [
        {
            'author': {'nickname': 'John'},
            'body': 'Beautiful day in Portland!'
        },
        {
            'author': {'nickname': 'Susan'},
            'body': 'The Avengers movie was so cool!'
        }
    ]
    return render_template('index.html',
                           title='Home',
                           form=form,
                           posts=posts)
```

Let's review the changes in this function one by one:

- We are now importing the `Post` and `PostForm` classes
- We accept POST requests in both routes associated with the `index` view function, since that is how we will receive submitted posts.
- When we arrive at this view function through a form submission we insert a new `Post` record into the database. When we arrive at it via a regular GET request we do as before.
- The template now receives an additional argument, the `form`, so that it can render the text field.

One final comment before we continue. Notice how after we insert a new `Post` into the detabase we do this:

```
return redirect(url_for('index'))
```

We could have easily skipped the redirect and allowed the function to continue down into the template rendering part, and it would have been more efficient. Because really, all the redirect does is return to this same view function to do that, after an extra trip to the client web browser.

So, why the redirect? Consider what happens after the user writes a blog post, submits it and then hits the browser's refresh key. What will the refresh command do? Browsers resend the last issued request as a result of a refresh command.

Without the redirect, the last request is the POST request that submitted the form, so a refresh action will resubmit the form, causing a second `Post` record that is identical to the first to be written to the database. Not good.

By having the redirect, we force the browser to issue another request after the form submission, the one that grabs the redirected page. This is a simple GET request, so a refresh action will now repeat the GET request instead of submitting the form again.

This simple trick avoids inserting duplicate posts when a user inadvertently refreshes the page after submitting a blog post.

Displaying blog posts

And now we get to the fun part. We are going to grab blog posts from the database and display them.

If you recall from a few articles ago, we created a couple of *fake* posts and we've been displaying those in our home page for a long time. The fake objects were created explicitly in the `index` view function as a simply Python list:

```
    posts = [
        {
            'author': {'nickname': 'John'},
            'body': 'Beautiful day in Portland!'
        },
        {
            'author': {'nickname': 'Susan'},
            'body': 'The Avengers movie was so cool!'
        }
    ]
```

But in the last article we created the query that allows us to get all the posts from followed users, so now we can simply replace the above with this (file `app/views.py`):

```
    posts = g.user.followed_posts().all()
```

And when you run the application you will be seeing blog posts from the database!

The `followed_posts` method of the `User` class returns a sqlalchemy query object that is configured to grab the posts we are interested in. Calling `all()` on this query just retrieves all the posts into a list, so we end up with a structure that is very much alike the fake one we've been using until now. It's so close that the template does not even notice.

At this point feel free to play with the application. You can create a few users, make them follow others, and finally post some messages to see how each user sees its blog post stream.

Pagination

The application is looking better than ever, but we have a problem. We are showing all of the followed posts in the home page. What happens if a user has a thousand followed posts? Or a million? As you can imagine, grabbing and handling such a large list of objects will be extremely inefficient.

Instead, we are going to show this potentially large number of posts in groups, or *pages*.

Flask-SQLAlchemy comes with very good support for *pagination*. If for example, we wanted to get the first three followed posts of some user we can do this:

```
posts = g.user.followed_posts().paginate(1, 3, False).items
```

The `paginate` method can be called on any query object. It takes three arguments:

- the page number, starting from 1,
- the number of items per page,
- an error flag. If True, when an out of range page is requested a 404 error will be automatically returned to the client web browser. If False, an empty list will be returned instead of an error.

The return value from `paginate` is a `Pagination` object. The `items` member of this object contains the list of items in the requested page. There are other useful things in the `Pagination` object that we will see a bit later.

Now let's think about how we can implement pagination in our `index` view function. We can start by adding a configuration item to our application that determines how many items per page we will display (file `config.py`):

```
# pagination
POSTS_PER_PAGE = 3
```

It is a good idea to have these global `knobs` that can change the behavior of our application in the configuration file all together, because then we can go to a single place to revise them all.

In the final application we will of course use a much larger number than 3, but for testing it is useful to work with small numbers.

Next, let's decide how the URLs that request different pages will look. We've seen before that Flask routes can take arguments, so we can add a suffix to the URL that indicates the desired page:

```
http://localhost:5000/         <-- page #1 (default)
http://localhost:5000/index    <-- page #1 (default)
http://localhost:5000/index/1  <-- page #1
http://localhost:5000/index/2  <-- page #2
```

This format of URLs can be easily implemented with an additional `route` added to our view function (file `app/views.py`):

```
from config import POSTS_PER_PAGE
```

```
@app.route('/', methods=['GET', 'POST'])
@app.route('/index', methods=['GET', 'POST'])
@app.route('/index/<int:page>', methods=['GET', 'POST'])
@login_required
def index(page=1):
    form = PostForm()
    if form.validate_on_submit():
        post = Post(body=form.post.data, timestamp=datetime.utcnow(),
author=g.user)
        db.session.add(post)
        db.session.commit()
        flash('Your post is now live!')
        return redirect(url_for('index'))
    posts = g.user.followed_posts().paginate(page, POSTS_PER_PAGE, False).items
    return render_template('index.html',
                           title='Home',
                           form=form,
                           posts=posts)
```

Our new route takes the page argument, and declares it as an integer. We also need to add the `page` argument to the `index` function, and we have to give it a default value because two of the three routes do not have this argument, so for those the default will always be used.

And now that we have a page number available to us we can easily hook it up to our `followed_posts` query, along with the POSTS_PER_PAGE configuration constant we defined earlier.

Note how easy these changes are, and how little code is affected each time we make a change. We are trying to write each part of the application without making any assumptions regarding how the other parts work, and this enables us to write modular and robust applications that are easier to test and are less likely to fail or have bugs.

At this point you can try the pagination by entering URLs for the different pages by hand into your browser's address bar. Make sure you have more than three posts available so that you can see more than one page.

Page navigation

We now need to add links that allow users to navigate to the next and/or previous pages, and luckily this is extremely easy to do, Flask-SQLAlchemy does most of the work for us.

We are going to start by making a small change in the view function. In our current version we use the paginate method as follows:

```
posts = g.user.followed_posts().paginate(page, POSTS_PER_PAGE, False).items
```

By doing this we are only keeping the `items` member of the `Pagination` object returned by `paginate`. But this object has a number of other very useful things in it, so we will instead keep the whole object (file `app/views.py`):

```
posts = g.user.followed_posts().paginate(page, POSTS_PER_PAGE, False)
```

To compensate for this change, we have to modify the template (file `app/templates/index.html`):

```
<!-- posts is a Paginate object -->
{% for post in posts.items %}
<p>
  {{ post.author.nickname }} says: <b>{{ post.body }}</b>
</p>
{% endfor %}
```

What this change does is make the full Paginate object available to our template. The members of this object that we will use are:

- `has_next`: True if there is at least one more page after the current one
- `has_prev`: True if there is at least one more page before the current one
- `next_num`: page number for the next page
- `prev_num`: page number for the previous page

With these for elements we can produce the following (file `app/templates/index.html`):

```
<!-- posts is a Paginate object -->
{% for post in posts.items %}
<p>
  {{ post.author.nickname }} says: <b>{{ post.body }}</b>
</p>
{% endfor %}
{% if posts.has_prev %}<a href="{{ url_for('index',
page=posts.prev_num) }}">&lt;&lt; Newer posts</a>{% else %}&lt;&lt; Newer posts{%
endif %} |
{% if posts.has_next %}<a href="{{ url_for('index', page=posts.next_num) }}">Older
posts &gt;&gt;</a>{% else %}Older posts &gt;&gt;{% endif %}
```

So we have two links. First we have one labeled "Newer posts" that sends us to the previous page (keep in mind we show posts sorted by newest first, so the first page is the one with the newest stuff). Conversely, the "Older posts" points to the next page.

When we are looking at the first page we do not want to show a link to go to the previous page, since there isn't one. This is easy to detect because `posts.has_prev` will be `False`. We handle that case simply by showing the same text of the link but without the link itself. The link to the next page is handled in the same way.

Implementing the Post sub-template

Back in the article where we added avatar pictures we defined a sub-template with the HTML rendering of a single post. The reason we created this sub-template was so that we can render posts with a consistent look in multiple pages, without having to duplicate the HTML code.

It is now time to implement this sub-template in our index page. And, as most of the things we are

doing today, it is surprisingly simple (file `app/templates/index.html`):

```
<!-- posts is a Paginate object -->
{% for post in posts.items %}
    {% include 'post.html' %}
{% endfor %}
```

Amazing, huh? We just discarded our old rendering code and replaced it with an `include` of the sub-template. Just with this, we get the nicer version of the post that includes the user's avatar.

Here is a screenshot of the index page of our application in its current state:

The user profile page

We are done with the index page for now. However, we have also included posts in the user profile page, not posts from everyone but just from the owner of the profile. To be consistent the user profile page should be changed to match the index page.

The changes are similar to those we made on the index page. Here is a summary of what we need to do:

- add an additional route that takes the page number
- add a `page` argument to the view function, with a default of 1
- replace the list of fake posts with the proper database query and pagination
- update the template to use the pagination object

Here is the updated view function (file `app/views.py`):

```
@app.route('/user/<nickname>')
@app.route('/user/<nickname>/<int:page>')
@login_required
def user(nickname, page=1):
    user = User.query.filter_by(nickname=nickname).first()
    if user is None:
        flash('User %s not found.' % nickname)
        return redirect(url_for('index'))
    posts = user.posts.paginate(page, POSTS_PER_PAGE, False)
    return render_template('user.html',
                           user=user,
                           posts=posts)
```

Note that this function already had an argument (the nickname of the user), so we add the page number as a second argument.

The changes to the template are also pretty simple (file `app/templates/user.html`):

```
<!-- posts is a Paginate object -->
{% for post in posts.items %}
    {% include 'post.html' %}
{% endfor %}
```

```
{% if posts.has_prev %}<a href="{{ url_for('user', nickname=user.nickname,
page=posts.prev_num) }}">&lt;&lt; Newer posts</a>{% else %}&lt;&lt; Newer posts{%
endif %} |
{% if posts.has_next %}<a href="{{ url_for('user', nickname=user.nickname,
page=posts.next_num) }}">Older posts &gt;&gt;</a>{% else %}Older posts &gt;&gt;{%
endif %}
```

Final words

Below I'm making available the updated version of the `microblog` application with all the pagination changes introduced in this article.

Download microblog-0.9.zip.

As always, a database isn't provided so you have to create your own. If you are following this series of articles you know how to do it. If not, then go back to the database article to find out.

As always, I thank you for following my tutorial. I hope to see you again in the next one!

Miguel

Part X: Full Text Search

Recap

In the previous article in the series we've enhanced our database queries so that we can get results on pages.

Today, we are going to continue working on our database, but in a different area. All applications that store content must provide a search capability.

For many types of web sites it is possible to just let Google, Bing, etc. index all the content and provide the search results. This works well for sites that have mostly static pages, like a forum. In our little `microblog` application the basic unit of content is just a short user post, not a whole page. The type of search results that we want are dynamic. For example, if we search for the word "dog" we want to see blog posts from any users that include that word. It is obvious that until someone searches for that word there is no page that the big search engines could have indexed with these results, so clearly we have no choice other than rolling our own search.

Introduction to full text search engines

Unfortunately support for full text search in relational databases is not well standardized. Each database implements full text search in its own way, and SQLAlchemy at this time does not have a full text search abstration.

We are currently using SQLite for our database, so we could just create a full text index using the facilities provided by SQLite, bypassing SQLAlchemy. But that isn't a good idea, because if one day we decide to switch to another database we would need to rewrite our full text search capability for another database.

So instead, we are going to let our database deal with the regular data, and we are going to create a specialized database that will be dedicated to text searches.

There are a few open source full text search engines. The only one that to my knowledge has a Flask extension is Whoosh, an engine also written in Python. The advantage of using a pure Python engine is that it will install and run anywhere a Python interpreter is available. The disadvantage is that search performance will not be up to par with other engines that are written in C or C++. In my opinion the ideal solution would be to have a Flask extension that can connect to several engines and abstract us from dealing with a particular one in the same way Flask-SQLAlchemy gives us the freedom to use several database engines, but nothing of that kind seems to be available for full text searching at this time. Django developers do have a very nice extension that supports several full text search engines called django-haystack. Maybe one day someone will create a similar extension for Flask.

But for now, we'll implement our text searching with Whoosh. The extension that we are going to use is

Flask-WhooshAlchemy, which integrates a Whoosh database with Flask-SQLAlchemy models.

Python 3 Compatibility

Unfortunately, we have a problem with Python 3 and these packages. The Flask-WhooshAlchemy extension was never made compatible with Python 3. I have forked this extension and made a few changes to make it work, so if you are on Python 3 you will need to uninstall the official version and install my fork:

```
$ flask/bin/pip uninstall flask-whooshalchemy
$ flask/bin/pip install git+git://github.com/miguelgrinberg/flask-whooshalchemy.git
```

Sadly this isn't the only problem. Whoosh also has issues with Python 3, it seems. In my testing I have encontered this bug, and to my knowledge there isn't a solution available, which means that at this time the full text search capability does not work well on Python 3. I will update this section once the issues are resolved.

Configuration

Configuration for Flask-WhooshAlchemy is pretty simple. We just need to tell the extension what is the name of the full text search database (file `config.py`):

```
WHOOSH_BASE = os.path.join(basedir, 'search.db')
```

Model changes

Since Flask-WhooshAlchemy integrates with Flask-SQLAlchemy, we indicate what data is to be indexed for searching in the proper model class (file `app/models.py`):

```
from app import app

import sys
if sys.version_info >= (3, 0):
    enable_search = False
else:
    enable_search = True
    import flask.ext.whooshalchemy as whooshalchemy

class Post(db.Model):
    __searchable__ = ['body']

    id = db.Column(db.Integer, primary_key=True)
    body = db.Column(db.String(140))
    timestamp = db.Column(db.DateTime)
    user_id = db.Column(db.Integer, db.ForeignKey('user.id'))

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

if enable_search:
    whooshalchemy.whoosh_index(app, Post)
```

The model has a new `__searchable__` field, which is an array with all the database fields that will be in the searchable index. In our case we only want to index the body field of our posts.

We also have to initialize the full text index for this model by calling the `whoosh_index` function. Note that since we know that the search capability currently does not work on Python 3 we have to skip its initialization. Once the problems in Whoosh are fixed the logic around `enable_search` can be removed.

Since this isn't a change that affects the format of our relational database we do not need to record a new migration.

Unfortunately any posts that were in the database before the full text engine was added will not be indexed. To make sure the database and the full text engine are synchronized we are going to delete all posts from the database and start over. First we start the Python interpreter. For Windows users:

```
flask\Scripts\python
```

And for everyone else:

```
flask/bin/python
```

Then in the Python prompt we delete all the posts:

```
>>> from app.models import Post
>>> from app import db
>>> for post in Post.query.all():
...    db.session.delete(post)
>>> db.session.commit()
```

Searching

And now we are ready to start searching. First let's add a few new posts to the database. We have two options to do this. We can just start the application and enter posts via the web browser, as regular users would do, or we can also do it in the Python prompt.

From the Python prompt we can do it as follows:

```
>>> from app.models import User, Post
>>> from app import db
>>> import datetime
>>> u = User.query.get(1)
>>> p = Post(body='my first post', timestamp=datetime.datetime.utcnow(), author=u)
>>> db.session.add(p)
>>> p = Post(body='my second post', timestamp=datetime.datetime.utcnow(), author=u)
>>> db.session.add(p)
>>> p = Post(body='my third and last post', timestamp=datetime.datetime.utcnow(),
author=u)
>>> db.session.add(p)
>>> db.session.commit()
```

The Flask-WhooshAlchemy extension is nice, because it hooks up into Flask-SQLAlchemy commits

automatically. We do not need to maintain the full text index, it is all done for us transparently.

Now that we have a few posts in our full text index we can issue searches:

```
>>> Post.query.whoosh_search('post').all()
[<Post u'my second post'>, <Post u'my first post'>, <Post u'my third and last
post'>]
>>> Post.query.whoosh_search('second').all()
[<Post u'my second post'>]
>>> Post.query.whoosh_search('second OR last').all()
[<Post u'my second post'>, <Post u'my third and last post'>]
```

As you can see in the examples above, the queries do not need to be limited to single words. In fact, Whoosh supports a pretty powerful search query language.

Integrating full text searches into the application

To make the searching capability available to our application's users we have to add just a few small changes.

Configuration

As far as configuration, we'll just indicate how many search results should be returned as a maximum (file `config.py`):

```
MAX_SEARCH_RESULTS = 50
```

Search form

We are going to add a search form to the navigation bar at the top of the page. Putting the search box at the top is nice, because then the search will be accessible from all pages.

First we add a search form class (file `app/forms.py`):

```
class SearchForm(Form):
    search = StringField('search', validators=[DataRequired()])
```

Then we need to create a search form object and make it available to all templates, since we will be putting the search form in the navigation bar that is common to all pages. The easiest way to achieve this is to create the form in the `before_request` handler, and then stick it in Flask's global `g` (file `app/views.py`):

```
from forms import SearchForm

@app.before_request
def before_request():
    g.user = current_user
    if g.user.is_authenticated():
        g.user.last_seen = datetime.utcnow()
        db.session.add(g.user)
        db.session.commit()
        g.search_form = SearchForm()
```

Then we add the form to our template (file `app/templates/base.html`):

```
<div>Microblog:
    <a href="{{ url_for('index') }}">Home</a>
    {% if g.user.is_authenticated() %}
    | <a href="{{ url_for('user', nickname=g.user.nickname) }}">Your Profile</a>
    | <form style="display: inline;" action="{{ url_for('search') }}" method="post"
name="search">{{ g.search_form.hidden_tag() }}
{{ g.search_form.search(size=20) }}<input type="submit" value="Search"></form>
    | <a href="{{ url_for('logout') }}">Logout</a>
    {% endif %}
</div>
```

Note that we only display the form when we have a logged in user. Likewise, the `before_request` handler will only create a form when a user is logged in, since our application does not show any content to guests that are not authenticated.

Search view function

The `action` field of our form was set above to send all search requests the the `search` view function. This is where we will be issuing our full text queries (file `app/views.py`):

```
@app.route('/search', methods=['POST'])
@login_required
def search():
    if not g.search_form.validate_on_submit():
        return redirect(url_for('index'))
    return redirect(url_for('search_results', query=g.search_form.search.data))
```

This function doesn't really do much, it just collects the search query from the form and then redirects to another page passing this query as an argument. The reason the search work isn't done directly here is that if a user then hits the refresh button the browser will put up a warning indicating that form data will be resubmitted. This is avoided when the response to a POST request is a redirect, because after the redirect the browser's refresh button will reload the redirected page.

Search results page

Once a query string has been received the form POST handler sends it via page redirection to the `search_results` handler (file `app/views.py`):

```
from config import MAX_SEARCH_RESULTS

@app.route('/search_results/<query>')
@login_required
def search_results(query):
    results = Post.query.whoosh_search(query, MAX_SEARCH_RESULTS).all()
    return render_template('search_results.html',
                           query=query,
                           results=results)
```

The search results view function sends the query into Whoosh, passing a maximum number of search results, since we don't want to be presenting a potentially large number of hits, we are happy showing

just the first fifty.

The final piece is the search results template (file `app/templates/search_results.html`):

```
<!-- extend base layout -->
{% extends "base.html" %}

{% block content %}
  <h1>Search results for "{{ query }}":</h1>
  {% for post in results %}
      {% include 'post.html' %}
  {% endfor %}
{% endblock %}
```

And here, once again, we can reuse our `post.html` sub-template, so we don't need to worry about rendering avatars or other formatting elements, since all of that is done in a generic way in the sub-template.

Final words

We now have completed yet another important, though often overlooked piece that any decent web application must have.

The source code for the updated `microblog` application is available below:

Download microblog-0.10.zip.

As always, the above download does not include a database or a flask virtual environment. See previous articles in the series to learn how to create these.

I hope you enjoyed this tutorial. If you have any questions feel free to write in the comments below. Thank you for reading, and I will be seeing you again in the next installment!

Miguel

Part XI: Email Support

Recap

In the most recent installments of this tutorial we've been looking at improvements that mostly had to do with our database.

Today we are letting our database rest for a bit, and instead we'll look at another important function that most web applications have: the ability to send emails to its users.

In our little `microblog` application we are going to implement one email related function, we will send an email to a user each time he/she gets a new follower. There are several more ways in which email support can be useful, so we'll make sure we design a generic framework for sending emails that can be reused.

Configuration

Luckily for us, Flask already has an extension that handles email called Flask-Mail, and while it will not take us 100% of the way, it gets us pretty close.

Back when we looked at unit testing, we added configuration for Flask to send us an email should an error occur in the production version of our application. That same information is used for sending application related emails.

Just as a reminder, what we need is two pieces of information:

- the email server that will be used to send the emails, along with any required authentication
- the email address(es) of the admins

This is what we did in the previous article (file `config.py`):

```
# email server
MAIL_SERVER = 'your.mailserver.com'
MAIL_PORT = 25
MAIL_USERNAME = None
MAIL_PASSWORD = None

# administrator list
ADMINS = ['you@example.com']
```

It goes without saying that you have enter the details of an actual email server and administrator above before the application can actually send emails. We are not going to enhance the server setup to allow those that require an encrypted communication through TLS or SSL. For example, if you want the application to send emails via your gmail account you would enter the following:

```
# email server
MAIL_SERVER = 'smtp.googlemail.com'
```

```
MAIL_PORT = 465
MAIL_USE_TLS = False
MAIL_USE_SSL = True
MAIL_USERNAME = os.environ.get('MAIL_USERNAME')
MAIL_PASSWORD = os.environ.get('MAIL_PASSWORD')

# administrator list
ADMINS = ['your-gmail-username@gmail.com']
```

Note that the username and password are read from environment variables. You will need to set `MAIL_USERNAME` and `MAIL_PASSWORD` to your Gmail login credentials. Putting sensitive information in environment variables is safer than writing down the information on a source file.

We also need to initialize a `Mail` object, as this will be the object that will connect to the SMTP server and send the emails for us (file `app/__init__.py`):

```
from flask.ext.mail import Mail
mail = Mail(app)
```

Let's send an email!

To learn how Flask-Mail works we'll just send an email from the command line. So let's fire up Python from our virtual environment and run the following:

```
>>> from flask.ext.mail import Message
>>> from app import app, mail
>>> from config import ADMINS
>>> msg = Message('test subject', sender=ADMINS[0], recipients=ADMINS)
>>> msg.body = 'text body'
>>> msg.html = '<b>HTML</b> body'
>>> with app.app_context():
...     mail.send(msg)
....
```

The snippet of code above will send an email to the list of admins that are configured in `config.py`. The sender will be the first admin in the list. The email will have text and HTML versions, so depending on how your email client is setup you may see one or the other. Note that we needed to create an `app_context` to send the email. Recent releases of Flask-Mail require this. An application context is created automatically when a request is handled by Flask. Since we are not inside a request we have to create the context by hand just so that Flask-Mail can do its job.

Pretty, neat. Now it's time to integrate this code into our application!

A simple email framework

We will now write a helper function that sends an email. This is just a generic version of the above test. We'll put this function in a new source file that will be dedicated to our email support functions (file `app/emails.py`):

```
from flask.ext.mail import Message
```

```
from app import mail

def send_email(subject, sender, recipients, text_body, html_body):
    msg = Message(subject, sender=sender, recipients=recipients)
    msg.body = text_body
    msg.html = html_body
    mail.send(msg)
```

Note that Flask-Mail support goes beyond what we are using. Bcc lists and attachments are available, for example, but we won't use them in this application.

Follower notifications

Now that we have the basic framework to send an email in place, we can write the function that sends out the follower notification (file `app/emails.py`):

```
from flask import render_template
from config import ADMINS

def follower_notification(followed, follower):
    send_email("[microblog] %s is now following you!" % follower.nickname,
               ADMINS[0],
               [followed.email],
               render_template("follower_email.txt",
                               user=followed, follower=follower),
               render_template("follower_email.html",
                               user=followed, follower=follower))
```

Do you find any surprises in here? Our old friend the `render_template` function is making an appearance. If you recall, we used this function to render all the HTML templates from our views. Like the HTML from our views, the bodies of email messages are an ideal candidate for using templates. As much as possible we want to keep logic separate from presentation, so emails will also go into the `templates` folder along with our views.

So we now need to write the templates for the text and HTML versions of our follower notification email. Here is the text version (file `app/templates/follower_email.txt`):

```
Dear {{ user.nickname }},

{{ follower.nickname }} is now a follower. Click on the following link to visit
{{ follower.nickname }}'s profile page:

{{ url_for('user', nickname=follower.nickname, _external=True) }}

Regards,

The microblog admin
```

For the HTML version we can do a little bit better and even show the follower's avatar and profile information (file `app/templates/follower_email.html`):

```
<p>Dear {{ user.nickname }},</p>
<p><a href="{{ url_for('user', nickname=follower.nickname,
```

```
_external=True) }}">{{ follower.nickname }}</a> is now a follower.</p>
<table>
    <tr valign="top">
        <td><img src="{{ follower.avatar(50) }}"></td>
        <td>
            <a href="{{ url_for('user', nickname=follower.nickname, _external=True)
}}">{{ follower.nickname }}</a><br />
            {{ follower.about_me }}
        </td>
    </tr>
</table>
<p>Regards,</p>
<p>The <code>microblog</code> admin</p>
```

Note the `_external=True` argument to `url_for` in the above templates. By default, the `url_for` function generates URLs that are relative to the domain from which the current page comes from. For example, the return value from `url_for("index")` will be `/index`, while in this case we want `http://localhost:5000/index`. In an email there is no domain context, so we have to force fully qualified URLs that include the domain, and the `_external` argument is just for that.

The final step is to hook up the sending of the email with the actual view function that processes the "follow" (file `app/views.py`):

```
from .emails import follower_notification

@app.route('/follow/<nickname>')
@login_required
def follow(nickname):
    user = User.query.filter_by(nickname=nickname).first()
    # ...
    follower_notification(user, g.user)
    return redirect(url_for('user', nickname=nickname))
```

Now you can create two users (if you haven't yet) and make one follow the other to see how the email notification works.

So that's it? Are we done?

We could now pat ourselves in the back for a job well done and take email notifications out of our list of features yet to implement.

But if you played with the application for some time and paid attention you may have noticed that now that we have email notifications when you click the `follow` link it takes 2 to 3 seconds for the browser to refresh the page, whereas before it was almost instantaneous.

So what happened?

The problem is that Flask-Mail sends emails synchronously. The web server blocks while the email is being sent and only returns its response back to the browser once the email has been delivered. Can you imagine what would happen if we try to send an email to a server that is slow, or even worse, temporarily offline? Not good.

This is a terrible limitation, sending an email should be a background task that does not interfere with the web server, so let's see how we can fix this.

Asynchronous calls in Python

What we really want is for the `send_email` function to return immediately, while the work of sending the email is moved to a background process.

Turns out Python already has support for running asynchronous tasks, actually in more than one way. The `threading` and `multiprocessing` modules can both do this.

Starting a thread each time we need to send an email is much less resource intensive than starting a brand new process, so let's move the `mail.send(msg)` call into thread (file `app/emails.py`):

```
from threading import Thread
from app import app

def send_async_email(app, msg):
    with app.app_context():
        mail.send(msg)

def send_email(subject, sender, recipients, text_body, html_body):
    msg = Message(subject, sender=sender, recipients=recipients)
    msg.body = text_body
    msg.html = html_body
    thr = Thread(target=send_async_email, args=[app, msg])
    thr.start()
```

The `send_async_email` function now runs in a background thread. Because it is a separate thread, the application context required by Flask-Mail will not be automatically set for us, so the `app` instance is passed to the thread, and the application context is set up manually, like we did above when we sent an email from the Python console.

If you test the 'follow' function of our application now you will notice that the web browser shows the refreshed page before the email is actually sent.

So now we have asynchronous emails implemented, but what if in the future we need to implement other asynchronous functions? The procedure would be identical, but we would need to duplicate the threading code for each particular case, which is not good.

We can improve our solution by implementing a decorator. With a decorator the above code would change to this:

```
from .decorators import async

@async
def send_async_email(app, msg):
    with app.app_context():
        mail.send(msg)

def send_email(subject, sender, recipients, text_body, html_body):
    msg = Message(subject, sender=sender, recipients=recipients)
```

```
    msg.body = text_body
    msg.html = html_body
    send_async_email(app, msg)
```

Much nicer, right?

The code that allows this magic is actually pretty simple. We will put it in a new source file (file `app/decorators.py`):

```
from threading import Thread

def async(f):
    def wrapper(*args, **kwargs):
        thr = Thread(target=f, args=args, kwargs=kwargs)
        thr.start()
    return wrapper
```

And now that we indirectly have created a useful framework for asynchronous tasks we can say we are done!

Just as an exercise, let's consider how this solution would look using processes instead of threads. We do not want a new process started for each email that we need to send, so instead we could use the `Pool` class from the `multiprocessing` module. This class creates a specified number of processes (which are forks of the main process) and all those processes wait to receive jobs to run, given to the pool via the `apply_async` method. This could be an interesting approach for a busy site, but we will stay with the threads for now.

Final words

The source code for the updated `microblog` application is available below:

Download microblog-0.11.zip.

I've got a few requests for putting this application up on github or similar, which I think is a pretty good idea. I will be working on that in the near future. Stay tuned.

Thank you again for following me on this tutorial series. I look forward to see you on the next chapter.

Miguel

Part XII: Facelift

Introduction

If you have been playing with the `microblog` application you must have noticed that we haven't spent too much time on its looks. Up to now the templates we put together were pretty basic, with absolutely no styling. This was useful, because we did not want the distraction of having to write good looking HTML when we were coding.

But we've been hard at work coding for a while now, so today we are taking a break and will see what we can do to make our application look a bit more appealing to our users.

This article is going to be different than previous ones because writing good looking HTML/CSS is a vast topic that falls outside of the intended scope of this series. There won't be any detailed HTML or CSS, we will just discuss basic guidelines and ideas so on how to approach the task.

How do we do this?

While we can argue that coding is hard, our pains are nothing compared to those of web designers, who have to write templates that have a nice and consistent look on a list of web browsers, most with obscure bugs or quirks. And in this modern age they not only need to make their designs look good on regular browsers but also on the resource limited browsers of tablets and smartphones.

Unfortunately, learning HTML, CSS and Javascript and on top of that being aware of the idiosyncrasies of each web browser is a task of uncertain dimensions. We really do not have time (or interest) to do that. We just want our application to look decent without having to invest a lot of energy.

So how can we approach the task of styling `microblog` with these constraints?

Introducing Bootstrap

Our good friends at Twitter have released an open source web framework called Bootstrap that might be our winning ticket.

Bootstrap is a collection of CSS and Javascript utilities for the most common types of web pages. If you want to see the kind of pages that can be designed with this framework here are some examples.

These are some of the things Bootstrap is good at:

- Similar looks in all major web browsers
- Handling of desktop, tablet and phone screen sizes
- Customizable layouts
- Fully styled navigation bars

- Fully styled forms
- And much, much more...

Bootstrapping `microblog`

Before we can add Bootstrap to our application we have to install the Bootstrap CSS, Javascript and image files in a place where our web server can find them.

In Flask applications the `app/static` folder is where regular files go. The web server knows to go look for files in these location when a URL has a `/static` prefix.

For example, if we store a file named `image.png` in `/app/static` then in an HTML template we can display the image with the following tag:

```
<img src="/static/image.png" />
```

We will install the Bootstrap framework according to the following structure:

```
/app
    /static
        /css
            bootstrap.min.css
            bootstrap-responsive.min.css
        /img
            glyphicons-halflings.png
            glyphicons-halflings-white.png
        /js
            bootstrap.min.js
```

Then in the `head` section of our base template we load the framework according to the instructions:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    ...
    <link href="/static/css/bootstrap.min.css" rel="stylesheet" media="screen">
    <link href="/static/css/bootstrap-responsive.min.css" rel="stylesheet">
    <script src="http://code.jquery.com/jquery-latest.js"></script>
    <script src="/static/js/bootstrap.min.js"></script>
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    ...
  </head>
  ...
</html>
```

The `link` and `script` tags load the CSS and Javascript files that come with Bootstrap. Note that one of the required Javascript files is jQuery, which is used by some of the Bootstrap plugins.

The `meta` tag enables Bootstrap's responsive mode, which scales the page appropriately for desktops, tablets and smartphones.

With these changes incorporated into our `base.html` template we are ready to start implementing

Bootstrap, which simply involves changing the HTML in our templates.

The changes that we will make are:

- Enclose the entire page contents in a single column fixed layout with responsive features.
- Adapt all forms to use Bootstrap form styles.
- Replace our navigation bar with a Navbar.
- Convert the previous and next pagination links to Pager buttons.
- Use the Boostrap alert styles for flashed messages.
- Use styled images to represent the suggested OpenID providers in the login form.

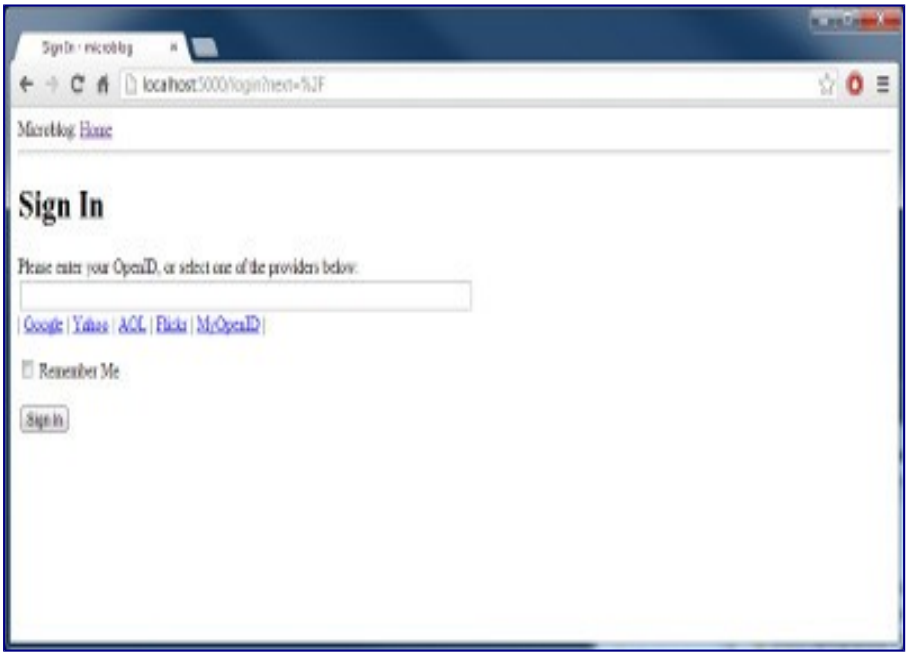
We will not discuss the specific changes to achieve the above since these are pretty simple. For those interested, the actual changes can be viewed in diff form on this github commit. The Bootstrap reference documentation will be useful when trying to analyze the new `microblog` templates.

Note: At the time this tutorial was written the current version of Bootstrap was 2.3.2. Twitter has released a new major version since then, and with it several of the CSS classes have changed. Visit the Bootstrap site for more information.

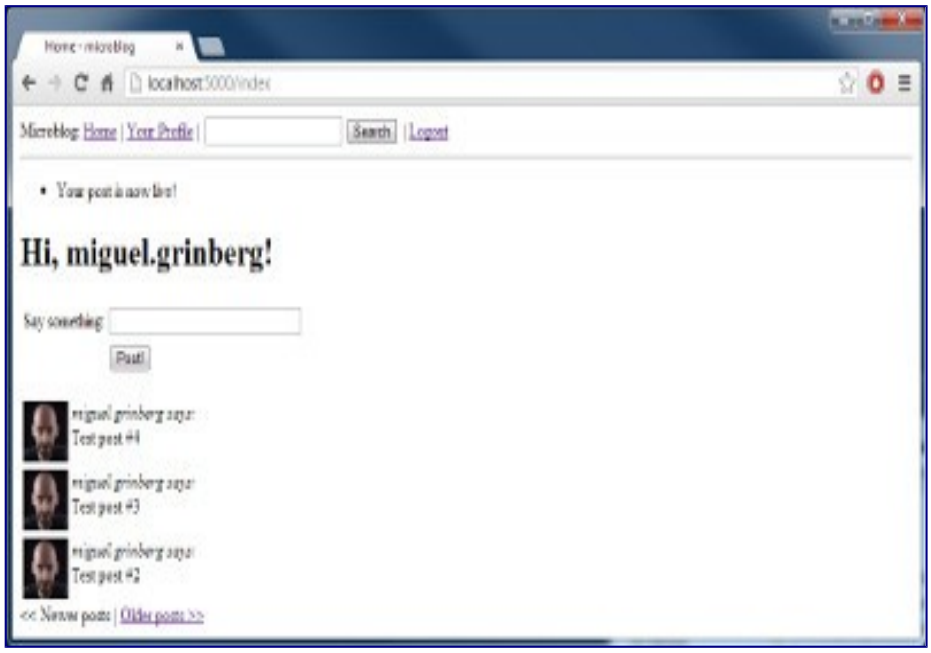
Final words

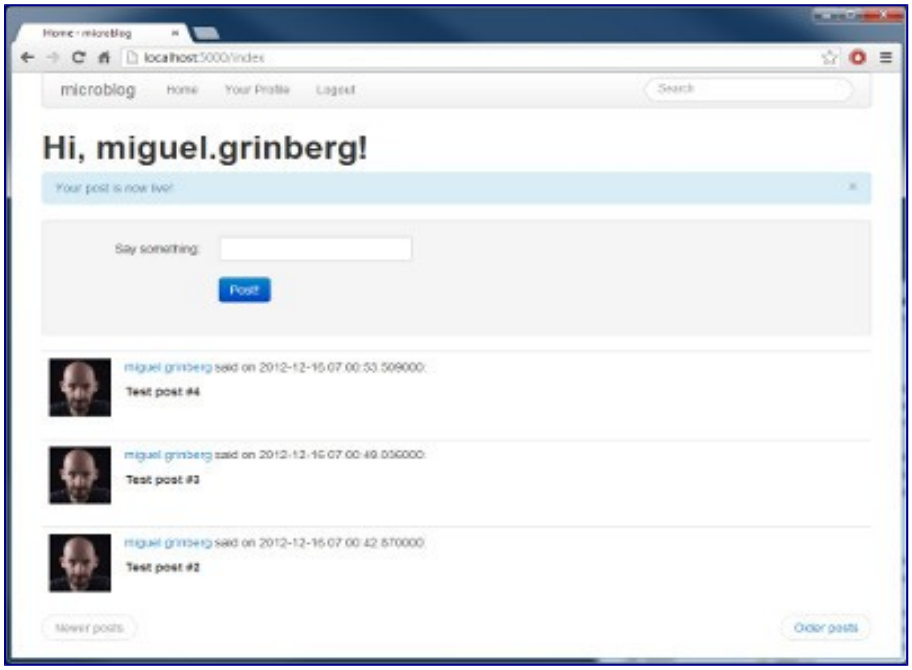
Today we've made the promise to not write a single line of code, and we stuck to it. All the improvements we've made were done with edits to the template files.

To give you an idea of the magnitude of the transformation, here are a few before and after screenshots. Click on the images to enlarge.




Sign In - microblog
localhost:5000/login?next=%2F
microblog
Home
Please Sign In
Click on your OpenID provider below:
Google
YAHOO!
AOL
flickr
myOpenID
Or enter your OpenID here:
Remember Me
Sign In


Home - microblog
localhost:5000/index
Microblog: Home | Your Profile | Search | Logout
Your post is now live!
Hi, miguel.grinberg!
Say something:
Post!
miguel.grinberg says:
Test post #4
miguel.grinberg says:
Test post #3
miguel.grinberg says:
Test post #2
<< Newer posts | Older posts >>



The updated application can be downloaded below:

Download microblog-0.12.zip.

In the next chapter we will look at improving the formatting of dates and times in our application. I look forward to see you then!

Miguel

Part XIII: Dates and Times

A quick note about github

For those that did not notice, I recently moved the hosting of the `microblog` application to github. You can find the repository at this location:

https://github.com/miguelgrinberg/microblog

I have added tags that point to each tutorial step for your convenience.

The problem with timestamps

One of the aspects of our `microblog` application that we have left ignored for a long time is the display of dates and times.

Until now, we just trusted Python to render the `datetime` objects in our `User` and `Post` objects on its own, and that isn't really a good solution.

Consider the following example. I'm writing this at 3:54PM on December 31st, 2012. My timezone is PST (or UTC-8 if you prefer). Running in a Python interpreter I get the following:

```
>>> from datetime import datetime
>>> now = datetime.now()
>>> print now
2012-12-31 15:54:42.915204
>>> now = datetime.utcnow()
>>> print now
2012-12-31 23:55:13.635874
```

The `now()` call returns the correct time for my location, while the `utcnow()` call returns the time in the UTC time zone.

So which one is better to use?

If we go with `now()` then all the timestamps that we store in the database will be local to where the application server is running, and this presents a few problems.

One day we may need to move the server to another location across time zones, and then all the times will have to be corrected to the new local time in the database before the server can be restarted.

But there is a more important problem with this approach. For users in different timezones it will be awfully difficult to figure out when a post was made if they see times in the PST timezone. They would need to know in advance that the times are in PST so that they can do the proper adjustments.

Clearly this is not a good option, and this is why back when we started our database we decided the we would always store timestamps in the UTC timezone.

While standardizing the timestamps to UTC solves the issue of moving the server across timezones, it does not address the second issue, dates and times are now presented to users from any part of the world in UTC.

This can still be pretty confusing to many. Imagine a user in the PST timezone that posts something at 3:00pm. The post immediately shows up in his or her index page with a 11:00pm time, or to be more exact 23:00.

The goal of today's article is to address the issue of date and time display so that our users do not get confused.

User specific timestamps

The obvious solution to the problem is to individually convert all timestamps from UTC to local time for each user. This allows us to continue using UTC in our database so that we have consistency, while an on-the-fly conversion for each user makes times consistent for everyone.

But how do we know the location of each of our users?

Many websites have a configuration page where users can specify their timezone. This would require us to add a new page with a form in which we present users with a dropdown with the list of timezones. Users should be asked to enter their timezone when they access the site for the first time, as part of their registration.

While this is a decent solution that solves our problem, it is a bit cumbersome to ask our users to enter a piece of information that they have already configured in their systems. It seems it would be more efficient if we could just grab the timezone setting from their computers.

For security reasons web browsers will not allow us to get into the computers of our users to obtain this information. Even if this was possible we would need to know where to find the timezone on Windows, Linux, Mac, iOS, and Android, without counting other less common operating systems.

But as it turns out, the web browser knows the user's timezone, and exposes it through the standard Javascript APIs. In this Web 2.0 world it is safe to assume that users will have Javascript enabled (no modern site will work without scripting), so this solution has potential.

We have two ways to take advantage of the timezone configuration available via Javascript:

- The "old-school" approach would be to have the web browser somehow send the timezone information to the server when the user first logs on to the server. This could be done with an Ajax call, or much more simply with a meta refresh tag. Once the server knows the timezone it can keep it in the user's session and adjust all timestamps with it when templates are rendered.
- The "new-school" approach would be to not change a thing in the server, which will continue to send UTC timestamps to the client browser. The conversion from UTC to local then happens in the client, using Javascript.

Both options are valid, but the second one has an advantage. The browser is best able to render times

according to the system locale configuration. Things like AM/PM vs. 24 hour clock, DD/MM/YYYY vs. MM/DD/YYYY and many other cultural styles are all accessible to the browser and unknown to the server.

And if that wasn't enough, there is yet one more advantage for the new-school approach. Turns out someone else has done all the work for us!

Introducing moment.js

Moment.js is a small, free and open source Javascript library that takes date and time rendering to another level. It provides every imaginable formatting option, and then some.

To use moment.js in our application we need to write a little bit of Javascript into our templates. We start by constructing a `moment` object from an ISO 8601 time. For example, using the UTC time from the Python example above we create a moment object like this:

```
moment("2012-12-31T23:55:13Z")
```

Once the object is constructed it can be rendered to a string in a large variety of formats. For example, a very verbose rendering according to the system locale would be done as follows:

```
moment("2012-12-31T23:55:13Z").format('LLLL');
```

Below is how your system renders the above date:

Here are some more examples of this same timestamp rendered in different formats:

Format	Result
L	
LL	
LLL	
LLLL	
dddd	

The library support for rendering options does not end there. In addition to `format()` it offers `fromNow()` and `calendar()`, with much more friendly renderings of timestamps:

Format	Result
fromNow()	
calendar()	

Note that in all the examples above the server is rendering the same UTC time, the different renderings were executed by your own web browser.

The last bit of Javascript magic that we are missing is the code that actually makes the string returned by `moment` visible in the page. The simplest way to accomplish this is with Javascript's `document.write` function, as follows:

```
<script>
```

```
document.write(moment("2012-12-31T23:55:13Z").format('LLLL'));
</script>
```

While using `document.write` is extremely simple and straightforward as a way to generate portions of an HTML document via javascript, it should be noted that this approach has some limitations. The most important one to note is that `document.write` can only be used while a document is loading, it cannot be used to modify the document once it has completed loading. As a result of this limitation this solution would not work when loading data via Ajax.

Integrating `moment.js`

There are a few things we need to do to be able to use `moment.js` in our microblog application.

First, we place the downloaded library `moment.min.js` in the `/app/static/js` folder, so that it can be served to clients as a static file.

Next we add the reference to this library in our base template (file `app/templates/base.html`):

```
<script src="/static/js/moment.min.js"></script>
```

We can now add `<script>` tags in the templates that show timestamps and we would be done. But instead of doing it that way, we are going to create a wrapper for `moment.js` that we can invoke from the templates. This is going to save us time in the future if we need to change our timestamp rendering code, because we will have it just in one place.

Our wrapper will be a very simple Python class (file `app/momentjs.py`):

```
from jinja2 import Markup

class momentjs(object):
    def __init__(self, timestamp):
        self.timestamp = timestamp

    def render(self, format):
        return Markup("<script>\ndocument.write(moment(\"%s\").%s);\n</script>" %
(self.timestamp.strftime("%Y-%m-%dT%H:%M:%S Z"), format))

    def format(self, fmt):
        return self.render("format(\"%s\")" % fmt)

    def calendar(self):
        return self.render("calendar()")

    def fromNow(self):
        return self.render("fromNow()")
```

Note that the `render` method does not directly return a string but instead wraps the string inside a `Markup` object provided by Jinja2, our template engine. The reason is that Jinja2 escapes all strings by default, so for example, our `<script>` tag will not arrive as such to the client but as `&lt;script&gt;`. Wrapping the string in a `Markup` object tells Jinja2 that this string should not be

escaped.

So now that we have a wrapper we need to hook it up with Jinja2 so that our templates can call it (file `app/__init__.py`):

```
from .momentjs import momentjs
app.jinja_env.globals['momentjs'] = momentjs
```

This just tells Jinja2 to expose our class as a global variable to all templates.

Now we are ready to modify our templates. we have two places in our application where we display dates and times. One is in the user profile page, where we show the "last seen" time. For this timestamp we will use the `calendar()` formatting (file `app/templates/user.html`):

```
{% if user.last_seen %}
<p><em>Last seen: {{ momentjs(user.last_seen).calendar() }}</em></p>
{% endif %}
```

The second place is in the post sub-template, which is invoked from the index, user and search pages. In posts we will use the `fromNow()` formatting, because the exact time of a post isn't as important as how long ago it was made. Since we abstracted the rendering of a post into the sub-template we now need to make this change in just one place to affect all the pages that render posts (file `app/templates/post.html`):

```
<p><a href="{{ url_for('user',
nickname=post.author.nickname)}}">{{ post.author.nickname }}</a> said
{{ momentjs(post.timestamp).fromNow() }}:</p>
<p><strong>{{ post.body }}</strong></p>
```

And with these simple template changes we have solved all our timestamps issues. We didn't need to make a single change to our server code!

Final words

Without even noticing it, today we've made an important step towards making `microblog` accessible to international users with the change to render dates and times according to the client configured locale.

In the next installment of this series we are going to make our international users even happier, as we will enable `microblog` to render in multiple languages.

In the meantime, here is the download link for the application with the new moment.js integration:

Download microblog-0.13.zip.

Or if you prefer, you can find the code on github here.

Thank you for following my tutorials, I hope to see you in the next one.

Miguel

Part XIV: I18n and L10n

The topics of today's article are Internationalization and Localization, commonly abbreviated I18n and L10n. We want our `microblog` application to be used by as many people as possible, so we can't forget that there is a lot of people out there that can't speak English, or that can, but prefer to speak their native language.

To make our application accessible to foreign visitors we are going to use the Flask-Babel extension, which provides an extremely easy to use framework to translate the application into different languages.

Configuration

To work with translations we are going to use a package called Babel, along with its Flask extension Flask-Babel. Flask-Babel is initialized simply by creating an instance of class `Babel` and passing the Flask application to it (file `app/__init__.py`):

```
from flask.ext.babel import Babel
babel = Babel(app)
```

We also need to decide which languages we will offer translations for. For now we'll start with a Spanish version, since we have a translator at hand (yours truly), but it'll be easy to add more languages later. The list of supported languages will go to our configuration (file `config.py`):

```
# -*- coding: utf-8 -*-
# ...
# available languages
LANGUAGES = {
    'en': 'English',
    'es': 'Español'
}
```

The `LANGUAGES` dictionary has keys that are the language codes for the available languages, and values that are the printable name of the language. We are using the short language codes here, but when necessary the long codes that specify language and region can be used as well. For example if we wanted to support American and British English as separate languages we would have `'en-US'` and `'en-GB'` in our dictionary.

Note that because the word `Español` has a foreign character in it we have to add the `coding` comment at the top of the Python source file, to tell the Python interpreter that we are using the UTF-8 encoding and not ASCII, which lacks the ñ character.

The next piece of configuration that we need is a function that Babel will use to decide which language to use (file `app/views.py`):

```
from app import babel
```

```
from config import LANGUAGES

@babel.localeselector
def get_locale():
    return request.accept_languages.best_match(LANGUAGES.keys())
```

The function that is marked with the `localeselector` decorator will be called before each request to give us a chance to choose the language to use when producing its response. For now we will do something very simple, we'll just read the `Accept-Languages` header sent by the browser in the HTTP request and find the best matching language from the list that we support. This is actually pretty simple, the `best_match` method does all the work for us.

The `Accept-Languages` header in most browsers is configured by default with the language selected at the operating system level, but all browsers give users the chance to select other languages. Users can provide a list of languages, each with a weight. As an example, here is a complex `Accept-Languages` header:

```
Accept-Language: da, en-gb;q=0.8, en;q=0.7
```

This says that Danish is the preferred language (with default weight = 1.0), followed by British English (weight = 0.8) and as a last option generic English (weight = 0.7).

The final piece of configuration that we need is a Babel configuration file that tells Babel where to look for texts to translate in our code and templates (file `babel.cfg`):

```
[python: **.py]
[jinja2: **/templates/**.html]
extensions=jinja2.ext.autoescape,jinja2.ext.with_
```

The first two lines give Babel the filename patterns for our Python and template files respectively. The third line tells Babel to enable a couple of extensions that make it possible to find text to translate in Jinja2 templates.

Marking texts to translate

Now comes the most tedious aspect of this task. We need to review all our code and templates and mark all English texts that need translating so that Babel can find them. For example, take a look at this snippet from function `after_login`:

```
if resp.email is None or resp.email == "":
    flash('Invalid login. Please try again.')
    redirect(url_for('login'))
```

Here we have a flash message that we want to translate. To expose this text to Babel we just wrap the string in Babel's `gettext` function:

```
from flask.ext.babel import gettext
# ...
if resp.email is None or resp.email == "":
```

```
    flash(gettext('Invalid login. Please try again.'))
    redirect(url_for('login'))
```

In a template we have to do something similar, but we have the option to use `_()` as a shorter alias to `gettext()`. For example, the word `Home` in this link from our base template:

```
  <li><a href="{{ url_for('index') }}">Home</a></li>
```

can be marked for translation as follows:

```
  <li><a href="{{ url_for('index') }}">{{ _('Home') }}</a></li>
```

Unfortunately not all texts that we want to translate are as simple as the above. As an example of a tricky one, consider the following snippet from our `post.html` subtemplate:

```
<p><a href="{{ url_for('user',
nickname=post.author.nickname) }}">{{ post.author.nickname }}</a> said
{{ momentjs(post.timestamp).fromNow() }}:</p>
```

Here the sentence that we want to translate has this structure: "`<nickname>` said `<when>`:". One would be tempted to just mark the word "said" for translation, but we can't really be sure that the order of the name and the time components in this sentence will be the same in all languages. The correct thing to do here is to mark the entire sentence for translation using placeholders for the name and the time, so that a translator can change the order if necessary. To complicate matters more, the name component has a hyperlink embedded in it!

There isn't really a nice and easy way to handle cases like this. The `gettext` function supports placeholders using the syntax `%(name)s` and that's the best we can do. Here is a simple example of a placeholder in a much simpler situation:

```
gettext('Hello, %(name)s', name=user.nickname)
```

The translator will need to be aware that there are placeholders and that they should not be touched. Clearly the name of a placeholder (what's between the `%(` and `)s`) must not be translated or else the connection to the actual value would be lost.

But back to our post template example, here is how it is marked for translation:

```
{% autoescape false %}
<p>{{ _('%(nickname)s said %(when)s:', nickname = '<a href="%s">%s</a>' %
(url_for('user', nickname=post.author.nickname), post.author.nickname),
when=momentjs(post.timestamp).fromNow()) }}</p>
{% endautoescape %}
```

The text that the translator will see for the above example is:

```
%(nickname)s said %(when)s:
```

Which is pretty decent. The values for `nickname` and `when` are what gives this translatable sentence its complexity, but these are given as additional arguments to the `_()` wrapper function and are not

seen by the translator.

The `nickname` and `when` placeholders contain a lot of stuff in them. In particular, for the nickname we had to build an entire HTML link because we want this nickname to be clickable.

Because we are putting HTML in the `nickname` placeholder we need to turn off autoescaping to render this portion of the template, if not Jinja2 would render our HTML elements as escaped text. But requesting to render a string without escaping is considered a security risk, it is unsafe to render texts entered by users without escaping.

The text assigned to the `when` placeholder is safe because it is text that is entirely generated by our `momentjs()` wrapper function. What goes in the `nickname` argument, however, is coming from the `nickname` field of our `User` model, which in turn comes from our database, which can be entered by the user in a web form. If someone registers into our application with a nickname that contains embedded HTML or Javascript and then we render that malicious nickname unescaped, then we are effectively opening the door to an attacker. We certainly do not want that, so we are going to take a quick detour and remove any risks.

The solution that makes most sense is to prevent any attacks by restricting the characters that can be used in a nickname. We'll start by creating a function that converts an invalid nickname into a valid one (file `app/models.py`):

```
import re

class User(db.Model):
    #...
    @staticmethod
    def make_valid_nickname(nickname):
        return re.sub('[^a-zA-Z0-9_\.]', '', nickname)
```

Here we just take the nickname and remove any characters that are not letters, numbers, the dot or the underscore.

When a user registers with the site we receive his or her nickname from the OpenID provider, so we make sure we convert this nickname to something valid (file `app/views.py`):

```
@oid.after_login
def after_login(resp):
    #...
    nickname = User.make_valid_nickname(nickname)
    nickname = User.make_unique_nickname(nickname)
    user = User(nickname=nickname, email=resp.email)
    #...
```

And then in the Edit Profile form, where the user can change the nickname, we can enhance our validation to not allow invalid characters (file `app/forms.py`):

```
class EditForm(Form):
    #...
    def validate(self):
```

```
        if not Form.validate(self):
            return False
        if self.nickname.data == self.original_nickname:
            return True
        if self.nickname.data != User.make_valid_nickname(self.nickname.data):
            self.nickname.errors.append(gettext('This nickname has invalid
characters. Please use letters, numbers, dots and underscores only.'))
            return False
        user = User.query.filter_by(nickname=self.nickname.data).first()
        if user is not None:
            self.nickname.errors.append(gettext('This nickname is already in use.
Please choose another one.'))
            return False
        return True
```

With these simple measures we eliminate any possible attacks resulting from rendering the nickname to a page without escaping.

Extracting texts for translation

I'm not going to enumerate here all the changes required to mark all texts in the code and the templates. Interested readers can check the github diffs for this.

So let's assume we've found all the texts and wrapped them in `gettext()` or `_()` calls. What now?

Now we run `pybabel` to extract the texts into a separate file:

```
flask/bin/pybabel extract -F babel.cfg -o messages.pot app
```

Windows users should use this command instead:

```
flask\Scripts\pybabel extract -F babel.cfg -o messages.pot app
```

The `pybabel extract` command reads the given configuration file, then scans all the code and template files in the directories given as arguments (just `app` in our case) and when it finds a text marked for translation it copies it to the `messages.pot` file.

The `messages.pot` file is a template file that contains all the texts that need to be translated. This file is used as a model to generate language files.

Generating a language catalog

The next step in the process is to create a translation for a new language. We said we were going to do Spanish (language code `es`), so this is the command that adds Spanish to our application:

```
flask/bin/pybabel init -i messages.pot -d app/translations -l es
```

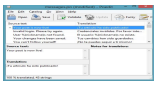
The `pybabel init` command takes the `.pot` file as input and writes a new language catalog to the directory given in the `-d` command line option for the language specified in the `-l` option. By default, Babel expects the translations to be in a `translations` folder at the same level as the templates, so

that's where we'll put them.

After running the above comment a directory `app/translations/es` is created. Inside it there is yet another directory called `LC_MESSAGES` and inside it there is a file called `messages.po`. The command can be executed multiple times with different language codes to add support for other languages.

The `messages.po` file that is created in each language folder uses a format that is the de facto standard for language translations, the format used by the venerable gettext utility. There are many translation applications that work with `.po` files. For our translation needs we will use poedit, because it is one of the most popular and because it runs on all the major operating systems.

If you want to put your translator hat and give this task a try go ahead and install poedit from this link. The usage of this application is straightforward. Below is a screenshot after all the texts have been translated to Spanish:



The top section shows the texts in their original and translated languages. The bottom left has a box where the translator writes the text.

Once the texts have been translated and saved back to the `messages.po` file there is yet another step to publish these texts:

```
flask/bin/pybabel compile -d app/translations
```

The `pybabel compile` step just reads the contents of the `.po` file and writes a compiled version as a `.mo` file in the same directory. This file contains the translated texts in an optimized format that can be efficiently used by our application.

The translations are now ready to be used. To check them you can modify the language settings in your browser so that Spanish is the preferred language, or if you don't feel like messing with your browser configuration you can also fake it by temporarily changing the `localeselector` function to always request Spanish (file `app/views.py`):

```
@babel.localeselector
def get_locale():
    return 'es'  # request.accept_languages.best_match(LANGUAGES.keys())
```

Now when we run the server each time the `gettext()` or `_()` functions are called instead of getting the English text we will get the translation defined for the language returned by the `localeselector` function.

Updating the translations

What happens if we leave the `messages.po` file incomplete, with some of the texts missing a

translation? Nothing happens, the application will run just fine regardless, and those texts that don't have a translation will continue to appear in English.

What happens if we missed some of the English texts in our code or templates? Any strings that were not wrapped with `gettext()` or `_()` will not be in the translation files, so they'll be unknown to Babel and remain in English. Once we spot a missing text we can add the `gettext()` wrapper to it and then run the following `pybabel` commands to update the translation files:

```
flask/bin/pybabel extract -F babel.cfg -o messages.pot app
flask/bin/pybabel update -i messages.pot -d app/translations
```

The `extract` command is identical to the one we issued earlier, it just generates an updated `messages.pot` file that adds the new texts. The `update` call takes the new `messages.pot` file and merges the new texts into all the translations found in the folder given by the `-d` argument.

Once the `messages.po` files in each language folder have been updated we can run `poedit` again to enter translations for the new texts, and then repeat the `pybabel compile` command to make those new texts available to our application.

Translating `moment.js`

So now that we have entered a Spanish translation for all the texts in code and templates we can run the application to see how it looks.

And right there we'll notice that all the timestamps are still in English. The `moment.js` library that we are using to render our dates and times hasn't been informed that we need a different language.

Reading the `moment.js` documentation we find that there is a large list of translations available, and that we simply need to load a second javascript with the selected language to switch to that language. So we just download the Spanish translation from the `moment.js` website and put it in our `static/js` folder as `moment-es.min.js`. We will follow the convention that any language file for `moment.js` will be added with the format `moment-<language>.min.js`, so that we can then select the correct one dynamically.

To be able to load a javascript that has the language code in its name we need to expose this code to the templates. The simplest way to do that is to add the language code to Flask's `g` global, in a similar way to how we expose the logged in user (file `app/views.py`):

```
@app.before_request
def before_request():
    g.user = current_user
    if g.user.is_authenticated():
        g.user.last_seen = datetime.utcnow()
        db.session.add(g.user)
        db.session.commit()
        g.search_form = SearchForm()
    g.locale = get_locale()
```

And now that we can see the language code in the templates we can load the `moment.js` language script in our base template (file `app/templates/base.html`):

```
{% if g.locale != 'en' %}
<script src="/static/js/moment-{{ g.locale }}.min.js"></script>
{% endif %}
```

Note that we make it conditional, because if we are showing the English version of the site we already have the correct texts from the first `moment.js` javascript file.

Lazy evaluation

While we continue playing with the Spanish version of our site we notice another problem. When we log out and try to log back in there is a flash message that reads "Please log in to access this page." in English. But where is this message? Unfortunately we aren't putting out this message, it is the Flask-Login extension that does it on its own.

Flask-Login allows this message to be configured by the user, so we are going to take advantage of that not to change the message but to make it translatable. So in our first try we do this (file `app/__init__.py`):

```
from flask.ext.babel import gettext
lm.login_message = gettext('Please log in to access this page.')
```

But this really does not work. The `gettext` function needs to be used in the context of a request to be able to produce translated messages. If we call it outside of a request it will just give us the default text, which will be the English version.

For cases like this Flask-Babel provides another function called `lazy_gettext`, which doesn't look for a translation immediately like `gettext()` and `_()` do but instead delay the search for a translation until the string is actually used. So here is how to properly set up the login message (file `app/__init__.py`):

```
from flask.ext.babel import lazy_gettext
lm.login_message = lazy_gettext('Please log in to access this page.')
```

When using `lazy_gettext` the `pybabel extract` command needs to be informed that the `lazy_gettext` function also wraps translatable texts with the `-k` option:

```
flask/bin/pybabel extract -F babel.cfg -k lazy_gettext -o messages.pot app
```

So after extracting yet another `messages.pot` template we update the language catalogs (`pybabel update`), translate the added text (`poedit`) and finally compile the translations one more time (`pybabel compile`).

With the advent of Flask 0.10 user sessions are serialized to JSON. This introduces a problem with lazy evaluated texts that are given as argument to the `flash()` function. Flashed messages are written to

the user session, but the object used to wrap the lazily evaluated texts is a complex object that does not have a direct conversion to a JSON type. Flask 0.9 did not serialize sessions to JSON so this was not a problem, but until Flask-Babel addresses this we have to provide a solution from our side, and this solution comes in the form of a custom JSON encoder (file `app/__init__.py`):

```
from flask.json import JSONEncoder

class CustomJSONEncoder(JSONEncoder):
    """This class adds support for lazy translation texts to Flask's
    JSON encoder. This is necessary when flashing translated texts."""
    def default(self, obj):
        from speaklater import is_lazy_string
        if is_lazy_string(obj):
            try:
                return unicode(obj)  # python 2
            except NameError:
                return str(obj)  # python 3
        return super(CustomJSONEncoder, self).default(obj)

app.json_encoder = CustomJSONEncoder
```

This installs a custom JSON encoder that forces the lazy texts to be evaluated into strings prior to being converted to JSON. Note the complexity in getting this done differently for Python 2 vs. Python 3.

And now we can say that we have a fully internationalized application!

Shortcuts

Since the `pybabel` commands are long and hard to remember we are going to end this article with a few quick and dirty little scripts that simplify the most common tasks we've seen above.

First a script to add a language to the translation catalog (file `tr_init.py`):

```
#!flask/bin/python
import os
import sys
if sys.platform == 'win32':
    pybabel = 'flask\\Scripts\\pybabel'
else:
    pybabel = 'flask/bin/pybabel'
if len(sys.argv) != 2:
    print "usage: tr_init <language-code>"
    sys.exit(1)
os.system(pybabel + ' extract -F babel.cfg -k lazy_gettext -o messages.pot app')
os.system(pybabel + ' init -i messages.pot -d app/translations -l ' + sys.argv[1])
os.unlink('messages.pot')
```

Then a script to update the catalog with new texts from source and templates (file `tr_update.py`):

```
#!flask/bin/python
import os
import sys
if sys.platform == 'win32':
    pybabel = 'flask\\Scripts\\pybabel'
```

```
else:
    pybabel = 'flask/bin/pybabel'
os.system(pybabel + ' extract -F babel.cfg -k lazy_gettext -o messages.pot app')
os.system(pybabel + ' update -i messages.pot -d app/translations')
os.unlink('messages.pot')
```

And finally, a script to compile the catalog (file `tr_compile.py`):

```
#!flask/bin/python
import os
import sys
if sys.platform == 'win32':
    pybabel = 'flask\\Scripts\\pybabel'
else:
    pybabel = 'flask/bin/pybabel'
os.system(pybabel + ' compile -d app/translations')
```

These scripts should make working with translation files an easy task.

Final words

Today we have implemented an often overlooked aspect of web applications. Users want to work in their native language, so being able to publish our application in as many languages as we can find translators for is a huge accomplishment.

In the next article we will look at what is probably the most complex aspect in the area of I18n and L10n, which is the real time automated translation of user generated content. And we will use this as an excuse to add some Ajax magic to our application.

Here is the download link for the latest version of `microblog`, including the complete Spanish translation:

Download microblog-0.14.zip.

If you prefer, you can also find the code on github here.

Thank you for being a loyal reader. See you next time!

Miguel

Part XV: Ajax

This will be the last of the I18n and L10n themed articles, as we complete our effort to make our `microblog` application accessible and friendly to non-English speakers.

In this article we will take a departure from the "safe zone" of server-side development we've come to love and will work on a feature that has equally important server and client-side components. Have you seen the "Translate" links that some sites show next to user generated content? These are links that trigger a real time automated translation of content that is not in the user's native language. The translated content is typically inserted below the original version. Google shows it for search results in foreign languages. Facebook shows it for posts. Today we are adding that very same feature to `microblog`!

Server-side vs. Client-side

In the traditional server-side model that we've followed so far we have a client (the user's web browser) making requests to our server. A request can simply ask for a web page, like when you click the "Your Profile" link, or it can make us perform an action, like when the user edits his or her profile information and clicks the Submit button. In both types of requests the server completes the request by sending a new web page to the client, either directly or by issuing a redirect. The client then replaces the current page with the new one. This cycle repeats for as long as the user stays on our web site. We call this model server-side because the server does all the work while the client just displays the web pages as it receives them.

In the client-side model we have a web browser that again, issues a request to the server. The server responds with a web page like in server-side, but not all the page data is HTML, there is also code, typically written in Javascript. Once the client receives the page it will display it and then execute the code that came with it. From then on you have an active client that can do work on its own without reaching out to the server. In a strict client-side application the entire application is downloaded to the client with the initial page request, and then the application runs on the client without ever refreshing the page, only contacting the server to retrieve or store data. This type of applications are called Single Page Applications or SPAs.

Most applications are a hybrid between the two models and combine techniques of both worlds. Our `microblog` application is clearly a server-side application, but today we will be adding a little bit of client-side action to the mix. To do real time translations of user posts the client browser will send requests to the server, but the server will respond with translated texts without causing a page refresh. The client will then insert the translations into the current page dynamically. This technique is known as Ajax, which is short for Asynchronous Javascript and XML (even though these days XML is often replaced with JSON).

Translating user generated content

We now have pretty good support for foreign languages thanks to Flask-Babel. Assuming we can find translators willing to help us, we could publish our application in as many languages as we want.

But there is one element missing. We made the application very friendly and inviting to people of all places, so now we are going to get content in a variety of languages, so our users are likely to come across blog posts that are written in languages they don't understand. Wouldn't it be nice if we could offer an automated translation service? The quality of automated translations isn't great, but in most cases it is good enough to get an idea of what someone said in a language we don't speak, so all our users (even ourselves!) can benefit from such a feature.

This is an ideal feature to implement as an Ajax service. Consider that our index page could be showing several posts, some of which might be in several different foreign languages. If we implemented the translation using traditional server-side techniques a request for a translation would cause the original page to get replaced with a new page showing just the translation for the selected post. After reading the translation the user would have to hit the back button to go back to the post listing. The fact is that requesting a translation isn't a big enough action to require a full page update, this feature works much better if the translated text is dynamically inserted below the original text while leaving the rest of the page untouched. So today we are implementing our first Ajax service!

Implementing live automated translations requires a few steps. First we need to identify the source language for the text we want to translate. Once we know the language we also know if a translation is needed for a given user, because we also know what language the user has chosen. When a translation is offered and the user wishes to see it we invoke the Ajax translation service, which will run on our server. In the final step the client-side javascript code will dynamically insert the translated text into the page.

Identifying the language of a post

Our first problem is identifying what language a post was written in. This isn't an exact science, it will not always be possible to detect a language, so we will consider it a "best effort" kind of thing. We are going to use the `guess-language` Python module for this. Unfortunately if you are using Python 3 the package guess-language will have issues, but a fork exists that can be used in its place:

```
flask/bin/pip uninstall guess-language
flask/bin/pip install
https://bitbucket.org/spirit/guess_language/downloads/guess_language-spirit-
0.5a4.tar.bz2
```

With this module we will scan the text of each blog post to try to guess its language. Since we don't want to scan the same posts over and over again we will do this once for each post, at the time the post is submitted by the user. We will then store the detected language along with the post in our database.

So let's begin by adding a `language` field to our `Posts` table:

```
class Post(db.Model):
    __searchable__ = ['body']

    id = db.Column(db.Integer, primary_key=True)
    body = db.Column(db.String(140))
    timestamp = db.Column(db.DateTime)
    user_id = db.Column(db.Integer, db.ForeignKey('user.id'))
    language = db.Column(db.String(5))
```

Each time we modify the database we also need to do a migration:

```
$ ./db_migrate.py
New migration saved as microblog/db_repository/versions/005_migration.py
Current database version: 5
```

Now we have a place to store the language of each post, so let's detect the language of each added blog post:

```
from guess_language import guessLanguage

@app.route('/', methods=['GET', 'POST'])
@app.route('/index', methods=['GET', 'POST'])
@app.route('/index/<int:page>', methods=['GET', 'POST'])
@login_required
def index(page=1):
    form = PostForm()
    if form.validate_on_submit():
        language = guessLanguage(form.post.data)
        if language == 'UNKNOWN' or len(language) > 5:
            language = ''
        post = Post(body=form.post.data,
                    timestamp=datetime.utcnow(),
                    author=g.user,
                    language=language)
        db.session.add(post)
        db.session.commit()
        flash(gettext('Your post is now live!'))
        return redirect(url_for('index'))
    posts = g.user.followed_posts().paginate(page, POSTS_PER_PAGE, False)
    return render_template('index.html',
                           title='Home',
                           form=form,
                           posts=posts)
```

If the language guessing doesn't work or returns an unexpectedly long result we play it safe and save an empty string to the database, a convention that we will use to signal that a post has an unknown language.

Displaying the "Translate" link

The next step is to show a "Translate" link next to any posts that are not in the language used by the user (file `app/templates/posts.html`):

```
{% if post.language != None and post.language != '' and post.language != g.locale
%}
```

```
<div><a href="#">{{ _('Translate') }}</a></div>
{% endif %}
```

We are doing this in the `post.html` sub-template, so any page that displays blog posts will get this functionality. The Translate link will only be visible if we could detect the language of the blog post and the language we detected is different than the language selected by the `localeselector` function from Flask-Babel, which based on the work we did in the previous article we can now access at `g.locale`.

This link required us to add a new text, the word "Translate" which needs to be included in our translation files, so we entered it with the translation wrapper to make it visible to Flask-Babel. To get it translated we have to refresh our language catalog (`tr_update.py`), translate in poedit and recompile (`tr_compile.py`), as seen in the I18n article.

We don't really know what we will be doing to trigger a translation yet, so for now the link does nothing.

Translation services

Before we can proceed with our application we need to find a translation service that we can use.

There are many translation services available, but unfortunately most are either paid services or free with big limitations.

The two major translation services are Google Translate and Microsoft Translator. Both are paid services, but the Microsoft offering has an entry level option for low volume of translations that is free. Google offered a free translation service in the past but that isn't available anymore. That makes our choice of translation service very easy.

Using the Microsoft Translator service

To use the Microsoft Translator there are a couple of requirements that need to be met:

- The application developer needs to register with the Microsoft Translator app at the Azure Marketplace. Here the level of service can be selected (the free option is at the bottom).
- Then the developer needs to register an application. The registered application will get Client ID and Client Secret codes to be used as part of the requests sent to this service.

Once the registration part is complete the process to request a translation is as follows:

- Get an access token, passing in the client id and secret.
- Call the desired translation method, either Ajax, HTTP or SOAP, providing the access token and the text to translate.

This really sounds more complicated than it really is, so without going into details here is a function that does all the dirty work and translates a text to another language (file `app/translate.py`):

```
try:
    import httplib  # Python 2
except ImportError:
    import http.client as httplib  # Python 3
try:
    from urllib import urlencode  # Python 2
except ImportError:
    from urllib.parse import urlencode  # Python 3
import json
from flask.ext.babel import gettext
from config import MS_TRANSLATOR_CLIENT_ID, MS_TRANSLATOR_CLIENT_SECRET

def microsoft_translate(text, sourceLang, destLang):
    if MS_TRANSLATOR_CLIENT_ID == "" or MS_TRANSLATOR_CLIENT_SECRET == "":
        return gettext('Error: translation service not configured.')
    try:
        # get access token
        params = urlencode({
            'client_id': MS_TRANSLATOR_CLIENT_ID,
            'client_secret': MS_TRANSLATOR_CLIENT_SECRET,
            'scope': 'http://api.microsofttranslator.com',
            'grant_type': 'client_credentials'})
        conn = httplib.HTTPSConnection("datamarket.accesscontrol.windows.net")
        conn.request("POST", "/v2/OAuth2-13", params)
        response = json.loads (conn.getresponse().read())
        token = response[u'access_token']

        # translate
        conn = httplib.HTTPConnection('api.microsofttranslator.com')
        params = {'appId': 'Bearer ' + token,
                  'from': sourceLang,
                  'to': destLang,
                  'text': text.encode("utf-8")}
        conn.request("GET", '/V2/Ajax.svc/Translate?' + urlencode(params))
        response = json.loads("{\"response\":" +
conn.getresponse().read().decode('utf-8') + "}")
        return response["response"]
    except:
        return gettext('Error: Unexpected error.')
```

This function imports two new items from our configuration file, the id and secret codes assigned by Microsoft (file `config.py`):

```
# microsoft translation service
MS_TRANSLATOR_CLIENT_ID = '' # enter your MS translator app id here
MS_TRANSLATOR_CLIENT_SECRET = '' # enter your MS translator app secret here
```

To use the service you will need to register yourself and the application to receive the codes that go in the above configuration variables. Even if you just intend to test this application you can register with the service for free.

We have added more texts, this time a few error messages. These need translations, so we have to re-run `tr_update.py`, `poedit` and `tr_compile.py` to update our translation files.

Let's translate some text!

So how do we use the translator service? It's actually very simple. Here is an example:

```
$ flask/bin/python
Python 2.6.8 (unknown, Jun  9 2012, 11:30:32)
>>> from app import translate
>>> translate.microsoft_translate('Hi, how are you today?', 'en', 'es')
u'¿Hola, cómo estás hoy?'
```

Ajax in the server

Now we can translate texts between languages, so we are ready to integrate this functionality into our application.

When the user clicks the Translate link in a post there will be an Ajax call issued to our server. We'll see how this call is made in a bit, for now let's concentrate on implementing the server side of the Ajax call.

An Ajax service in the server is like a regular view function with the only difference that instead of returning an HTML page or a redirect it returns data, typically formatted as either XML or JSON. Since JSON is much more friendly to Javascript we will go with that format (file `app/views.py`):

```
from flask import jsonify
from .translate import microsoft_translate

@app.route('/translate', methods=['POST'])
@login_required
def translate():
    return jsonify({
        'text': microsoft_translate(
            request.form['text'],
            request.form['sourceLang'],
            request.form['destLang']) })
```

There is not much new here. This route handles a POST request that should come with the text to translate and the source and destination language codes. Since this is a POST request we access these items as if this was data entered in an HTML form, using the `request.form` dictionary. We call one of our translation functions with this data and once we get the translated text we just convert it to JSON using Flask's `jsonify` function. The data that the client will see as a response to this request will have this format:

```
{ "text": "<translated text goes here>" }
```

Ajax in the client

Now we need to call the Ajax view function from the web browser, so we need to go back to the `post.html` sub-template to complete the work we started earlier.

We start by wrapping the post text in a `span` element with a unique id, so that we can later find it in

the DOM and replace it with the translated text (file `app/templates/post.html`):

```
<p><strong><span id="post{{ post.id }}">{{ post.body }}</span></strong></p>
```

Note how we construct the unique id using the post's id number. If a given post has an `id = 3` then the id of the `<span>` element that wraps it will be `post3`.

We are also going to wrap the Translate link in a span with a unique id, in this case so that we can hide it once the translation is shown:

```
<div><span id="translation{{post.id}}"><a
href="#">{{ _('Translate') }}</a></span></div>
```

Continuing with the example above, the translation link would get the id `translation3`.

And to make it a nice and user friendly feature we are also going to throw in a spinning wheel animation that starts hidden and will only appears while the translation service is running on the server, also with a unique id:

```
<img id="loading{{post.id}}" style="display: none" src="/static/img/loading.gif">
```

So now we have an element called `post<id>` that contains the text to translate, an element called `translation<id>` that contains the Translate link but will later be replaced with the translated text and an image with id `loading<id>` that will be shown while the translation service is working. The `<id>` suffix is what makes these elements unique, we can have many posts in a page and each will get its own set of three elements.

We now need to trigger the Ajax call when the Translate link is clicked. Instead of triggering the call directly from the link we'll create a Javascript function that does all the work, since we have a few things to do there and we don't want to duplicate code in every post. Let's start by adding a call to this function when the translate link is clicked:

```
<a href="javascript:translate('{{ post.language }}', '{{ g.locale }}',
'#post{{ post.id }}', '#translation{{ post.id }}',
'#loading{{ post.id }}');">{{ _('Translate') }}</a>
```

The template variables obscure this code a bit, but really the function that we are calling is simple. Assuming a post with id = 23 that is written in Spanish and is viewed by a user that uses English then this function will be called as follows:

```
translate('es', 'en', '#post23', '#translation23', '#loading23')
```

So this function will receive the source and destination languages and the three DOM elements associated with the post to translate.

We can't write this function in the `post.html` sub-template because its contents are repeated for each post. We'll implement the function in our base template, so that there is a single copy of it that is accessible in all pages (file `app/templates/base.html`):

```
<script>
function translate(sourceLang, destLang, sourceId, destId, loadingId) {
    $(destId).hide();
    $(loadingId).show();
    $.post('/translate', {
        text: $(sourceId).text(),
        sourceLang: sourceLang,
        destLang: destLang
    }).done(function(translated) {
        $(destId).text(translated['text'])
        $(loadingId).hide();
        $(destId).show();
    }).fail(function() {
        $(destId).text("{{ _('Error: Could not contact server.') }}");
        $(loadingId).hide();
        $(destId).show();
    });
}
</script>
```

We rely on jQuery for this functionality. Recall that we included jQuery back when we styled the application using Twitter's Bootstrap framework.

We start by hiding the element that contains the translate link and showing the spinning wheel image.

Then we use jQuery's `$.post()` function to send the Ajax request to our server. The `$.post` function issues a POST request that to the server will be identical to a request issued by the browser when a form is submitted. The difference is that in the client this request happens in the background without causing a page reload. When a response is received from the server the function given as an argument to the `done()` call will execute and will have a chance to insert the data received into the page. This function receives the response as an argument, so we just overwrite the translate link in the DOM with the translated text, then hide the spinning wheel and finally display the translated text for the user to see. Neat!

If there is any error that prevents the client from getting a response from the server then the function inside the `fail()` call will be invoked. In this case we just show a generic error message, which also needs to be translated into all languages and thus requires a refresh of our translation database.

Unit testing

Remember our unit test framework? It is always a good idea to evaluate if it makes sense to write tests every time we add a new feature. Invoking the text translation service could be something we could inadvertently break while we work on our code, or it could break due to updates in the service. Let's write a quick test that makes sure we can contact the service and obtain translations:

```
from app.translate import microsoft_translate

class TestCase(unittest.TestCase):
    #...
    def test_translation(self):
        assert microsoft_translate(u'English', 'en', 'es') == u'Inglés'
```

```
        assert microsoft_translate(u'Español', 'es', 'en') == u'Spanish'
```

We do not have a client-side testing framework at this time, so we will not test the end-to-end Ajax procedure.

Running our testing framework should give us no errors:

```
$ ./tests.py
.....
----------------------------------------------------------------------
Ran 5 tests in 5.932s

OK
```

But if you run the test framework without having done the proper configuration for the Microsoft Translator you would get:

```
$ ./tests.py
....F
======================================================================
FAIL: test_translation (__main__.TestCase)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "./tests.py", line 120, in test_translation
    assert microsoft_translate(u'English', 'en', 'es') == u'Inglés'
AssertionError

----------------------------------------------------------------------
Ran 5 tests in 3.877s

FAILED (failures=1)
```

Final words

And with this we conclude this article. I hope it was as fun to read for you as it was for me to write!

I have recently been alerted of some issues with the database when Flask-WhooshAlchemy is used for the full text search. In the next chapter I will use this problem as an excuse to go over some of my debugging techniques for Flask applications. Stay tuned for episode XVI!

The code for the updated application is available on github, here. For those that do not use github below is a friendly download link:

Download microblog-0.15.zip.

I look forward to see you in the next article!

Miguel

Part XVI: Debugging, Testing and Profiling

Our little `microblog` application is starting to feel decent enough for a release, so it is time to start thinking about cleaning things up as much as we can. Recently, a reader of this blog (Hi, George!) reported a strange database bug that we are going to debug today. This should help us realize that no matter how careful we are and how much we test our application there is a good chance we will miss some bugs. Unfortunately users are very good at finding them!

Instead of just fixing this bug and forgetting about it until we come across another one, we are going to take some proactive measures to be better prepared for the next one.

In the first part of this article we'll cover *debugging*, and I'll show you some techniques I use to debug tough problems.

Later we will see how we can have an idea of how effective our testing strategy is. We will specifically look at measuring how much of our application our unit tests are exercising, something called *test coverage*.

And finally, we will think about another type of problems that many applications can suffer from, bad performance. We will look at *profiling* techniques to find the parts of our application that are slow.

Sounds good? Let's get started then.

The Bug

This problem was found by a reader of this blog, after he implemented a new function that allows users to delete their own posts. The official version of `microblog` does not have that, so we'll quickly put together this feature so that we can work on the bug.

The view function that we will use to delete a post is below (file `app/views.py`):

```
@app.route('/delete/<int:id>')
@login_required
def delete(id):
    post = Post.query.get(id)
    if post is None:
        flash('Post not found.')
        return redirect(url_for('index'))
    if post.author.id != g.user.id:
        flash('You cannot delete this post.')
        return redirect(url_for('index'))
    db.session.delete(post)
    db.session.commit()
    flash('Your post has been deleted.')
    return redirect(url_for('index'))
```

To invoke this function we will add a delete link on all posts that are authored by the logged in user (file `app/templates/post.html`):

```
{% if post.author.id == g.user.id %}
<div><a href="{{ url_for('delete', id=post.id) }}">{{ _('Delete') }}</a></div>
{% endif %}
```

There is nothing earth shattering here, we've done this many times already.

NOTE: The bug discussed in this section only existed when using an old version of Flask-WhooshAlchemy. If you would like to experience the bug then uninstall the current version and then install version 0.54a of this extension.

Now let's go ahead and start our application in production mode, to experience it like a regular user would. Linux and Mac users do it like this:

```
$ ./runp.py
```

Windows users do it like this instead:

```
flask/Scripts/python runp.py
```

Now as a user, write a post, then change your mind and delete it. And right when you click the delete link... Boom!

We get a terse message that says that the application has encountered an error and that the administrator has been notified. This message is actually our `500.html` template. In production mode Flask issues the status code 500 template back to the client when an exception occurs while handling a request. Since we are now in production mode we will not see error messages or stack traces.

Debugging issues from the field

If you recall the unit testing article, we enabled a couple of debugging services to run on the production version of our application. Back then we created a logger that writes to a log file, so that the application can write debugging or diagnostic information at run time. Flask itself writes the stack trace of any exceptions that aren't handled before ending the request with a code 500 error. As an extra option, we have also enabled an email based logger that will send the people in the admin list an email when an error is written to the log message.

So for a bug like the one above we would get some debugging information captured in two places, the log file and an email sent to us.

A stack trace isn't much to go on, but it's far better than nothing. Assuming we know nothing about the problem, we now need to figure out what happened from the stack trace alone. Here is a copy of this particular stack trace:

```
127.0.0.1 - - [03/Mar/2013 23:57:39] "GET /delete/12 HTTP/1.1" 500 -
Traceback (most recent call last):
  File "/home/microblog/flask/lib/python2.7/site-packages/flask/app.py", line 1701,
in __call__
    return self.wsgi_app(environ, start_response)
  File "/home/microblog/flask/lib/python2.7/site-packages/flask/app.py", line 1689,
in wsgi_app
```

```
    response = self.make_response(self.handle_exception(e))
  File "/home/microblog/flask/lib/python2.7/site-packages/flask/app.py", line 1687,
in wsgi_app
    response = self.full_dispatch_request()
  File "/home/microblog/flask/lib/python2.7/site-packages/flask/app.py", line 1360,
in full_dispatch_request
    rv = self.handle_user_exception(e)
  File "/home/microblog/flask/lib/python2.7/site-packages/flask/app.py", line 1358,
in full_dispatch_request
    rv = self.dispatch_request()
  File "/home/microblog/flask/lib/python2.7/site-packages/flask/app.py", line 1344,
in dispatch_request
    return self.view_functions[rule.endpoint](**req.view_args)
  File "/home/microblog/flask/lib/python2.7/site-packages/flask_login.py", line
496, in decorated_view
    return fn(*args, **kwargs)
  File "/home/microblog/app/views.py", line 195, in delete
    db.session.delete(post)
  File "/home/microblog/flask/lib/python2.7/site-
packages/sqlalchemy/orm/scoping.py", line 114, in do
    return getattr(self.registry(), name)(*args, **kwargs)
  File "/home/microblog/flask/lib/python2.7/site-
packages/sqlalchemy/orm/session.py", line 1400, in delete
    self._attach(state)
  File "/home/microblog/flask/lib/python2.7/site-
packages/sqlalchemy/orm/session.py", line 1656, in _attach
    state.session_id, self.hash_key))
InvalidRequestError: Object '<Post at 0xff35e7ac>' is already attached to session
'1' (this is '3')
```

If you are used to reading stack traces in other languages, then be aware that Python shows the stack traces in reverse order, with the bottom frame being the one that caused the exception.

Now how do we make sense of this?

From the above stack trace we can see that the exception was triggered from SQLAlchemy's session handling code, which is safe to assume from the `sqlalchemy/orm/session.py` filename in the bottom most frame.

With stack traces it is always useful to find out what was the last statement in our own code that executed. If we start at the bottom and we go up one frame at a time we'll find that the fourth frame is in our `app/views.py`, more specifically in the `db.session.delete(post)` statement in our `delete()` view function.

Now we know that SQLAlchemy is unable to register the deletion of this post in the database session. But we still don't know why.

If you look at the text of the exception in the bottom most line of the stack trace the problem appears to be that the `Post` object is attached to a session `'1'` from before, and now we are trying to attach the same object to another session `'3'`.

If you Google the exception message you will find that most people having issues with this are using a multithreaded web server, and they get two requests trying to add the same object to different sessions

at the same time. But we are using Python's development web server which is single threaded, so that could not be our case. There is a different problem here that causes two sessions to be active at the same time, while there should only be one.

To see if we can learn more about the problem we should try to reproduce the bug in a more controlled environment. Luckily, trying to do this in the development version of the application does reproduce, and this is a bit better because when the exception occurs in development mode we get Flask's web based stack trace instead of the `500.html` template.

The web stack trace is nice because it allows you to inspect code and evaluate expressions all from the browser. Without really understanding much of what's going on in this code we know there is a session `'1'` (which we can assume is the first session ever created) that for some reason didn't get deleted like normal sessions do when the request that created them ends. So a good approach to make progress would be to figure out who is creating this first session that never goes away.

Using the Python Debugger

The easiest way to find out who is creating an object is to put a *breakpoint* in the object's constructor. A breakpoint is a command that interrupts the program when certain condition is met. At this point it is possible to inspect the program, like obtaining the stack trace at the time of interruption, checking or even changing variable values, etc. Breakpoints are one of the basic features found in *debuggers*. For this task we will use the debugger that comes with Python, appropriately called `pdb`.

But what class are we looking for? Let's go back to the web based stack trace and look some more there. In the bottom most stack frame we can use the code viewer and the Python console (note the icons to enable these on the right side when you hover over the frame) to find out the class that is used for sessions. In the code pane we see that we are inside a `Session` class, which is likely the base class for database sessions in SQLAlchemy. Since the context in the bottom stack frame is inside the session object we can just get the actual class of the session in the console, by running:

```
>>> print self
<flask_sqlalchemy._SignallingSession object at 0xff34914c>
```

And now we know that the sessions that we are using are defined by Flask-SQLAlchemy, because likely this extension defines its own session class, which is a subclass of SQLAlchemy's `Session`.

Now we can go and inspect the source code for the Flask-SQLAlchemy extension in `flask/lib/python2.7/site-packages/flask_sqlalchemy.py` and locate class `_SignallingSession` and its `__init__()` constructor, and now we are ready to play with the debugger.

There is more than one way to insert a breakpoint in a Python application. The simplest one is to just write the following at the place we want the program to stop:

```
import pdb; pdb.set_trace()
```

So we'll go ahead and temporarily insert this breakpoint in the `_SignallingSession` class constructor (file `flask/lib/python2.7/site-packages/flask_sqlalchemy.py`):

```
class _SignallingSession(Session):

    def __init__(self, db, autocommit=False, autoflush=False, **options):
        import pdb; pdb.set_trace() # <-- this is temporary!
        self.app = db.get_app()
        self._model_changes = {}
        Session.__init__(self, autocommit=autocommit, autoflush=autoflush,
                         extension=db.session_extensions,
                         bind=db.engine,
                         binds=db.get_binds(self.app), **options)

    # ...
```

So let's run the application again to see what happens:

```
$ ./run.py
> /home/microblog/flask/lib/python2.7/site-
packages/flask_sqlalchemy.py(198)__init__()
-> self.app = db.get_app()
(Pdb)
```

Since there is no "Running on..." message printed we know the server didn't actually complete its start up procedures. The interruption that brought us to the `pdb` prompt occurred because in some part of the application someone requested the creation of our mysterious session!

The most important question that we need to answer right now is where are we in the application, because that will tell us who is requesting the creation of this session `'1'` we can't get rid of later. We will use the `bt` command (short for backtrace) to get a stack trace:

```
(Pdb) bt
  /home/microblog/run.py(2)<module>()
-> from app import app
  /home/microblog/app/__init__.py(44)<module>()
-> from app import views, models
  /home/microblog/app/views.py(6)<module>()
-> from forms import LoginForm, EditForm, PostForm, SearchForm
  /home/microblog/app/forms.py(4)<module>()
-> from app.models import User
  /home/microblog/app/models.py(92)<module>()
-> whooshalchemy.whoosh_index(app, Post)
  /home/microblog/flask/lib/python2.6/site-
packages/flask_whooshalchemy.py(168)whoosh_index()
-> _create_index(app, model))
  /home/microblog/flask/lib/python2.6/site-
packages/flask_whooshalchemy.py(199)_create_index()
-> model.query = _QueryProxy(model.query, primary_key,
  /home/microblog/flask/lib/python2.6/site-
packages/flask_sqlalchemy.py(397)__get__()
-> return type.query_class(mapper, session=self.sa.session())
  /home/microblog/flask/lib/python2.6/site-
packages/sqlalchemy/orm/scoping.py(54)__call__()
-> return self.registry()
  /home/microblog/flask/lib/python2.6/site-
```

```
packages/sqlalchemy/util/_collections.py(852)__call__()
-> return self.registry.setdefault(key, self.createfunc())
> /home/microblog/flask/lib/python2.6/site-
packages/flask_sqlalchemy.py(198)__init__()
-> self.app = db.get_app()
(Pdb)
```

As we did before, we start from the bottom and locate the first stack frame that has code that we recognize as ours. That turns out to be our `models.py` file at line 92, which is the initialization of our full text search engine:

```
whooshalchemy.whoosh_index(app, Post)
```

Hmm. We are not doing anything intentional that would trigger the creation of a database session this early in the application's life, but it appears that the act of initializing Flask-WhooshAlchemy does create a database session.

This feels like this isn't our bug after all, maybe some sort of interaction between the two Flask extensions that wrap SQLAlchemy and Whoosh. We could stop here and ask for help from the developers of these two fine extensions or the communities around them. Or we could continue debugging to see if we can figure this out and have the problem solved today. I'll keep going, if you are bored already feel free to jump to the next section.

Let's have another look at the stack trace. We call `whoosh_index()`, which in turn calls `_create_index()`. The line of code in `_create_index()` is this:

```
model.query = _QueryProxy(model.query, primary_key,
            searcher, model)
```

The `model` variable in this context is set to our `Post` class, we passed it as an argument in our call to `whoosh_index()`. With that in mind, it appears Flask-WhooshAlchemy is creating a `Post.query` wrapper that takes the original `Post.query` as an argument, plus some other Whoosh specific stuff. And here is the interesting part. According to the stack trace above, the next function in the call stack is `__get__()`, one of Python's descriptor methods.

The `__get__()` method is used to implement descriptors, which are attributes that have behavior associated with them instead of just a value. Each time the descriptor is referenced the function `__get__()` is called. Then the function is supposed to return the value of the attribute. The only attribute that is mentioned in this line of code is `query`, so now we know that this seemingly simple attribute that we have used in the past to generate our database queries isn't really an attribute but a descriptor. The remainder of the stack trace then is dealing with computing the value of the `model.query` expression that appears in the right side, in preparation of creating the `_QueryProxy` constructor.

Let's continue down the stack trace to see what happens next. The instruction in `__get__()` is this one:

```
return type.query_class(mapper, session=self.sa.session())
```

And this is a pretty revealing piece of code. When we say, for example, `User.query.get(id)` we are indirectly calling this `__get__()` method to provide the query object, and here we can see that this query object implicitly brings a database session along with it!

When Flask-WhooshAlchemy says `model.query` that also triggers a session to be created and associated with the query object. But the query object that Flask-WhooshAlchemy requests isn't short lived like any normal queries we run inside our view functions. Flask-WhooshAlchemy is wrapping this query object into its own query object, which it then stores back into `model.query`. Since there is no `__set__()` method counterpart, the new object will be stored as an attribute. For our `Post` class that means that after Flask-WhooshAlchemy completes its initialization we will have a descriptor and an attribute with the same name. According to the precedence rules, in this case the attribute wins, which is expected, since if not our Whoosh searches would not have worked.

The important aspect of all this is that this code is setting a persistent attribute that inside has our session `'1'`. Even though the first request handled by the application will use this session and then forget about it, the session does not go away because it is still referenced by the `Post.query` attribute. This is our bug!

The bug is caused by the confusing (my opinion) nature of descriptors. They look like regular attributes, so people tend to use them as such. The Flask-WhooshAlchemy developer just wanted to create an enhanced query object that can store some state useful for Whoosh queries, but didn't realize that referencing the `query` attribute of a model does way more than it seems, as there is hidden behavior associated with this attribute that starts a database session.

Regression Testing

For many, the most logical thing to do at this point would be to fix the Flask-WhooshAlchemy code and move on. But if we just do that, then what protects us from this or a similar bug happening in the future? For example, what happens if a year from now we decide to update Flask-WhooshAlchemy to a new version and forget that we had applied a custom fix to it?

The best option every time a bug is discovered is to create a unit test for it, so that we can make sure we don't have a *regression* in the future.

There is some trickiness in creating a test for this bug, as we need to simulate two requests inside a single test. The first request will query a Post object, simulating the query that we make to request data for displaying in the web page. Since this is the first query it will use the session `'1'`. Then we need to forget that session and make a new one, exactly like Flask-SQLAlchemy does. Trying to delete the Post object on the second session should trigger this bug, because the first session would not have gone away as expected.

After taking another peek at the source code for Flask-SQLAlchemy we can see that new sessions are

created using the `db.create_scoped_session()` function, and when a request ends a session is destroyed by calling `db.session.remove()`. Knowing this makes it easy to write a test for this bug:

```
def test_delete_post(self):
    # create a user and a post
    u = User(nickname='john', email='john@example.com')
    p = Post(body='test post', author=u, timestamp=datetime.utcnow())
    db.session.add(u)
    db.session.add(p)
    db.session.commit()
    # query the post and destroy the session
    p = Post.query.get(1)
    db.session.remove()
    # delete the post using a new session
    db.session = db.create_scoped_session()
    db.session.delete(p)
    db.session.commit()
```

And sure enough, now when we run our test suite the failure appears:

```
$ ./tests.py
.E....
======================================================================
ERROR: test_delete_post (__main__.TestCase)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "./tests.py", line 133, in test_delete_post
    db.session.delete(p)
  File "/home/microblog/flask/lib/python2.7/site-
packages/sqlalchemy/orm/scoping.py", line 114, in do
    return getattr(self.registry(), name)(*args, **kwargs)
  File "/home/microblog/flask/lib/python2.7/site-
packages/sqlalchemy/orm/session.py", line 1400, in delete
    self._attach(state)
  File "/home/microblog/flask/lib/python2.7/site-
packages/sqlalchemy/orm/session.py", line 1656, in _attach
    state.session_id, self.hash_key))
InvalidRequestError: Object '<Post at 0xff09b7ac>' is already attached to session
'1' (this is '3')

----------------------------------------------------------------------
Ran 6 tests in 3.852s

FAILED (errors=1)
```

The Fix

To address this problem we need to find an alternative way of attaching Flask-WhooshAlchemy's query object to the model.

The documentation for Flask-SQLAlchemy mentions there is a model.query_class attribute that contains the class to use for queries. This is actually a much cleaner way to make Flask-SQLAlchemy use a custom query class than what Flask-WhooshAlchemy is doing. If we configure Flask-

SQLAlchemy to create queries using the Whoosh enabled query class (which is already a subclass of Flask-SQLAlchemy's `BaseQuery`), then we should have the same result as before, but without the bug.

I have created a fork of the Flask-WhooshAlchemy project on github where I have implemented these changes. If you want to see the changes you can see the github diff for my commit, or you can also download the fixed extension and install it in place of your original `flask_whooshalchemy.py` file.

I have submitted my fix to the developer of Flask-WhooshAlchemy, so I hope at some point it will get included in an official version (Update: my fix has been included in version 0.56).

Test Coverage

One way we can greatly reduce the chances of having bugs after we deploy our application is to have a comprehensive test suite. We already have a unit testing framework, but how do we know how much of our application are we currently testing with it?

A *test coverage* tool can observe a running application and take note of which lines of code execute and which do not. After execution ends it can produce a report showing what lines have not executed. If we had such report for our test suite we would know right away what parts of our code need tests that exercise them.

Python has a coverage tool that we can use called simply coverage that we installed way back when we started this tutorial. This tool can be used as a command line tool or can also be started from inside a script. To make it easier to not forget to run it we will go with the latter.

These are the changes that we need to make to add a coverage report to our test suite (file `tests.py`):

```
from coverage import coverage
cov = coverage(branch=True, omit=['flask/*', 'tests.py'])
cov.start()

# ...

if __name__ == '__main__':
    try:
        unittest.main()
    except:
        pass
    cov.stop()
    cov.save()
    print("\n\nCoverage Report:\n")
    cov.report()
    print("HTML version: " + os.path.join(basedir, "tmp/coverage/index.html"))
    cov.html_report(directory='tmp/coverage')
    cov.erase()
```

We begin by initializing the `coverage` module at the very top of the script. The `branch = True` argument requests that branch analysis is done in addition to regular line based coverage. The `omit`

argument makes sure we do not get coverage report for the modules we have installed in our virtual environment and for the unit testing framework itself, we just want coverage for our application code.

To gather coverage statistics we just call `cov.start()`, then run our unit tests. We have to catch and pass exceptions from the unit testing framework, because if not the script would end without giving us a chance to produce a coverage report. After we are back from the testing we stop coverage with `cov.stop()`, and write the results with `cov.save()`. Finally, `cov.report()` dumps the data to the console, `cov.html_report()` generates a nicer HTML report with the same date, and `cov.erase()` deletes the data file.

Here is an example test run with coverage report included (note I left an intentional failure in there):

```
$ ./tests.py
.....F
    ======================================================================
FAIL: test_translation (__main__.TestCase)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "./tests.py", line 143, in test_translation
    assert microsoft_translate(u'English', 'en', 'es') == u'Inglés'
AssertionError

----------------------------------------------------------------------
Ran 6 tests in 3.981s

FAILED (failures=1)


Coverage Report:

Name             Stmts   Miss Branch BrMiss  Cover   Missing
------------------------------------------------------------
app/__init__        39      0      6      3    93%
app/decorators       6      2      0      0    67%   5-6
app/emails          14      6      0      0    57%   9, 12-15, 21
app/forms           30     14      8      8    42%   15-16, 19-30
app/models          63      8     10      1    88%   32, 37, 47, 50, 53, 56, 78, 90
app/momentjs        12      5      0      0    58%   5, 8, 11, 14, 17
app/translate       33     24      4      3    27%   10-36, 39-56
app/views          169    124     46     46    21%   16, 20, 24-30, 34-37, 41, 45-
46, 53-67, 75-81, 88-109, 113-114, 120-125, 132-143, 149-164, 169-183, 188-198,
203-205, 210-211, 218
config              22      0      0      0   100%
------------------------------------------------------------
TOTAL              388    183     74     61    47%

HTML version: /home/microblog/tmp/coverage/index.html
```

With this report we know that we are testing 47% of our application. And we are given the list of lines that didn't execute during the test run, so we just need to review those lines and think about what tests we can write that use them.

We can see that our coverage for `app/models.py` is pretty high (88%), since we have mostly focused on testing our models. The `app/views.py` coverage is pretty low (21%) because we aren't

executing view function code in our tests yet.

In addition to lines of code that were missed, this tool gives us *branch coverage* information in the Branch and BrMiss columns. Consider the following example script:

```
def f(x):
    if x >= 0:
        x = x + 1
    return x

f(1)
```

If we run coverage in this simple function we will get 100% coverage, because when the function gets `1` as an argument its three lines execute. But we have never executed this function with the argument set to 0 or less, and that causes a different behavior. In a more complex case that could cause a bug.

Branch coverage tells us how many branches of code we have missed, and this is another pointer to the kinds of tests we might be missing in our suite.

The coverage tool also generates a very nice HTML based report that displays the source code annotated with color coded markers for line and branches covered and missed.

Continuing with our testing strategy centered in testing the models we can look at the parts of our `app/models.py` file that have no test coverage. This is easy to visualize in the HTML report, from where we can obtain the following list:

- `User.make_valid_nickname()`
- `User.is_authenticated()`
- `User.is_active()`
- `User.is_anonymous()`
- `User.get_id()`
- `User.__repr__()`
- `Post.__repr__()`
- `User.make_unique_nickname()` (only for the branch case where the given name is already unique)

We can put the first five in the list in a new test:

```
def test_user(self):
    # make valid nicknames
    n = User.make_valid_nickname('John_123')
    assert n == 'John_123'
    n = User.make_valid_nickname('John_[123]\n')
    assert n == 'John_123'
    # create a user
    u = User(nickname='john', email='john@example.com')
    db.session.add(u)
    db.session.commit()
    assert u.is_authenticated() is True
    assert u.is_active() is True
```

```
    assert u.is_anonymous() is False
    assert u.id == int(u.get_id())
```

The `__repr__()` functions are really used internally, so we don't need to test them. For this, we can mark them as not interesting as follows:

```
def __repr__(self):  # pragma: no cover
    return '<User %r>' % (self.nickname)
```

Finally, back when we wrote the test for `make_unique_nickname()` we focused only on how the function resolves a name collision, but we forgot to provide a test case that is unique and requires no handling. We can expand our existing test to cover that case as well:

```
def test_make_unique_nickname(self):
    # create a user and write it to the database
    u = User(nickname='john', email='john@example.com')
    db.session.add(u)
    db.session.commit()
    nickname = User.make_unique_nickname('susan')
    assert nickname == 'susan'
    nickname = User.make_unique_nickname('john')
    assert nickname != 'john'
    #...
```

And with these simple changes we get to almost 100% coverage on our `models.py` source file. When running on Python 2.7 there are only two Python 3 specific lines of code that are missed.

For now we'll call it good. Some other day we may decide to pick this up again and figure out a good way to test view functions, but we should feel good that we have full coverage on the code that talks to the database.

Profiling for performance

The next topic of the day is performance. There is nothing more frustrating for users than having to wait a long time for pages to load. We want to make sure our application is as fast as it can be so we need to take some measures to be prepared to analyze performance problems.

The technique that we are going to use is called `profiling`. A code profiler watches a running program, pretty much like coverage tools do, but instead of noting which lines execute and which don't it notes how much time is spent on each function. At the end of the profiling period a report is generated that lists all the functions that executed and how long was spent in each. Sorting this list from the largest to the smallest time will give us a pretty good idea of where we should spend time optimizing code.

Python comes with a nice source code profiler called cProfile. We could embed this profiler into our application directly, but before we do any work it is a good idea to search if someone has done the integration work already. A quick search for "Flask profiler" tells us that the Werkzeug module used by Flask comes with a profiler plugin that is all ready to go, so we'll just use that.

To enable the Werkzeug profiler we can create another starter script like `run.py`. Let's call it `profile.py`:

```
#!flask/bin/python
from werkzeug.contrib.profiler import ProfilerMiddleware
from app import app

app.config['PROFILE'] = True
app.wsgi_app = ProfilerMiddleware(app.wsgi_app, restrictions=[30])
app.run(debug = True)
```

Starting the application with the above script will enable the profiler to show the 30 most expensive functions for each request (the syntax for the `restrictions` argument is documented here).

Once the application starts, each request will show a profiler summary. Here is an example:

```
--------------------------------------------------------------------------------
PATH: '/'
         95477 function calls (89364 primitive calls) in 0.202 seconds

   Ordered by: internal time, call count
   List reduced from 1587 to 30 due to restriction <30>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.061    0.061    0.061    0.061 {method 'commit' of
'sqlite3.Connection' objects}
        1    0.013    0.013    0.018    0.018 flask/lib/python2.7/site-
packages/sqlalchemy/dialects/sqlite/pysqlite.py:278(dbapi)
    16807    0.006    0.000    0.006    0.000 {isinstance}
     5053    0.006    0.000    0.012    0.000 flask/lib/python2.7/site-
packages/jinja2/nodes.py:163(iter_child_nodes)
8746/8733    0.005    0.000    0.005    0.000 {getattr}
      817    0.004    0.000    0.011    0.000 flask/lib/python2.7/site-
packages/jinja2/lexer.py:548(tokeniter)
        1    0.004    0.004    0.004    0.004
/usr/lib/python2.7/sqlite3/dbapi2.py:24(<module>)
        4    0.004    0.001    0.015    0.004 {__import__}
        1    0.004    0.004    0.009    0.009 flask/lib/python2.7/site-
packages/sqlalchemy/dialects/sqlite/__init__.py:7(<module>)
   1808/8    0.003    0.000    0.033    0.004 flask/lib/python2.7/site-
packages/jinja2/visitor.py:34(visit)
     9013    0.003    0.000    0.005    0.000 flask/lib/python2.7/site-
packages/jinja2/nodes.py:147(iter_fields)
     2822    0.003    0.000    0.003    0.000 {method 'match' of '_sre.SRE_Pattern'
objects}
      738    0.003    0.000    0.003    0.000 {method 'split' of 'str' objects}
     1808    0.003    0.000    0.006    0.000 flask/lib/python2.7/site-
packages/jinja2/visitor.py:26(get_visitor)
     2862    0.003    0.000    0.003    0.000 {method 'append' of 'list' objects}
  110/106    0.002    0.000    0.008    0.000 flask/lib/python2.7/site-
packages/jinja2/parser.py:544(parse_primary)
       11    0.002    0.000    0.002    0.000 {posix.stat}
        5    0.002    0.000    0.010    0.002 flask/lib/python2.7/site-
packages/sqlalchemy/engine/base.py:1549(_execute_clauseelement)
        1    0.002    0.002    0.004    0.004 flask/lib/python2.7/site-
packages/sqlalchemy/dialects/sqlite/base.py:124(<module>)
  1229/36    0.002    0.000    0.008    0.000 flask/lib/python2.7/site-
packages/jinja2/nodes.py:183(find_all)
```

```
    416/4    0.002    0.000    0.006    0.002 flask/lib/python2.7/site-
packages/jinja2/visitor.py:58(generic_visit)
   101/10    0.002    0.000    0.003    0.000
flask/lib/python2.7/sre_compile.py:32(_compile)
       15    0.002    0.000    0.003    0.000 flask/lib/python2.7/site-
packages/sqlalchemy/schema.py:1094(_make_proxy)
        8    0.002    0.000    0.002    0.000 {method 'execute' of 'sqlite3.Cursor'
objects}
        1    0.002    0.002    0.002    0.002
flask/lib/python2.7/encodings/base64_codec.py:8(<module>)
        2    0.002    0.001    0.002    0.001 {method 'close' of
'sqlite3.Connection' objects}
        1    0.001    0.001    0.001    0.001 flask/lib/python2.7/site-
packages/sqlalchemy/dialects/sqlite/pysqlite.py:215(<module>)
        2    0.001    0.001    0.002    0.001 flask/lib/python2.7/site-
packages/wtforms/form.py:162(__call__)
      980    0.001    0.000    0.001    0.000 {id}
  936/127    0.001    0.000    0.008    0.000 flask/lib/python2.7/site-
packages/jinja2/visitor.py:41(generic_visit)


--------------------------------------------------------------------------------

127.0.0.1 - - [09/Mar/2013 19:35:49] "GET / HTTP/1.1" 200 -
```

The columns in this report are as follows:

- **ncalls**: number of times this function was called.
- **tottime**: total time spent inside this function.
- **percall**: this is tottime divided by ncalls.
- **cumtime**: total time spent inside this function and any functions called from it.
- **percall**: cumtime divided by ncalls.
- **filename:lineno(function)**: the function name and location.

It is interesting to note that our templates will also appear here as functions. This is because Jinja2 templates are compiled to Python code. That means that the profiler will not only point us to slow code we have written but also to slow templates!

We really don't have a performance problem at this point, at least not in this one request. We can see that the most time consuming functions are related to the sqlite3 database and Jinja2 template rendering, which is totally expected. Note in the header at the top that the this request took just 0.2 seconds to complete, so the scale of these per-function times is so small that it is in the noise.

As the application grows, it is useful to run new requests we add through the profiler, to make sure we are not doing anything that is slow.

Database Performance

To end this article, we are going to look at database performance. We've seen above that our database handling is at the top of the profiler report, so it would be good to have a system in place to get an alert when and if our database becomes too slow during production use.

The Flask-SQLAlchemy documentation mentions a get_debug_queries function, which returns a list of all the queries issued during the request with their durations.

This is very useful information. The intended use appears to be to time queries during debugging or testing, but being able to send an alert when a query takes too long is useful as a feature for the production version, even if it adds a little bit of overhead.

To use this feature during production we need to enable it in the configuration (file `config.py`):

```
SQLALCHEMY_RECORD_QUERIES = True
```

We are also going to setup a threshold for what we will consider a slow query (file `config.py`):

```
# slow database query threshold (in seconds)
DATABASE_QUERY_TIMEOUT = 0.5
```

To check if we need to send any alerts we'll add a hook after each request. In Flask this is easy, we just set up a `after_request` handler (file `app/views.py`):

```
from flask.ext.sqlalchemy import get_debug_queries
from config import DATABASE_QUERY_TIMEOUT

@app.after_request
def after_request(response):
    for query in get_debug_queries():
        if query.duration >= DATABASE_QUERY_TIMEOUT:
            app.logger.warning("SLOW QUERY: %s\nParameters: %s\nDuration:
%fs\nContext: %s\n" % (query.statement, query.parameters, query.duration,
query.context))
    return response
```

This will write to the log any queries that took longer than half a second. The information in the log will include the SQL statement, the actual parameters used in that statement, the duration and the location in the sources from where the query was issued.

With a small database like ours it is unlikely that we will have slow queries, but as the database and the application grow we may find that some database queries need to be optimized, for example with the addition of indexes. By simply checking the log file from time to time we will learn if some of the queries we are issuing need optimization.

Final words

Today we have done a number of mundane, yet important things all robust applications must have. The code for the updated application can be downloaded below:

Download microblog-0.16.zip.

Users comfortable with github can also get the code from here.

It feels like we are reaching the end of this tutorial series, as I'm running out of ideas for what to present. In the next, possibly last chapter we will look at deployment options, both traditional and cloud

based.

If you have any topics that you feel have not been given coverage in this tutorial please let me know in the comments below.

I hope to see you next time!

Miguel

Part XVII: Deployment on Linux (even on the Raspberry Pi!)

Today we are reaching a huge milestone in the life of our `microblog` application. We are going public with it, as we will be looking at different ways in which we can deploy it and make it accessible to our users.

In this article I will explore the traditional hosting options, and will be looking at hands-on deployment on a Linux box and on the widely popular Raspberry Pi minicomputer. In the next article we will be looking at deployment on cloud services.

To begin we will be looking at the more traditional of all deployment options. We will install the application behind a web server on a dedicated host that we will manage.

But where can we find a host? These days there are many cheap server offers out there. The cheapest options are for a VPS (virtual private server), which is virtual machine that looks like a standalone and dedicated server to you, but is really sharing the physical hardware with a few others like it. You can check on lowendbox.com for deals if you want to get one to experiment with (note that I'm not affiliated with them, I do not endorse them or make money if you click the link above).

Another easy way to obtain a host is to install a virtual machine on your regular PC. If you are a Windows or Mac user and would like to experience deploying the application to a Linux server without spending a dime, then this is the way to go. You can install VirtualBox on your system, download an ISO image for the Linux distribution of your choice and then create a VM on which you install this ISO image.

Let's talk a bit about operating system choices. I think there are four decent choices for hosting a web server:

- Linux
- BSD
- OS X
- Windows

From a technical point of view you could setup a Python based web server on any of the above, so how do we choose?

In my opinion the choice isn't too hard to make. For me it is important that we are on a platform that is open source, because the community at large does a much better job at keeping a platform safe and secure than what a single company can do on a closed source product. We also want to be on a platform that has a large server installation base, because that gives us more chances of finding solutions to our problems. Based on the above criteria, I consider Windows and OS X inferior to Linux and BSD as servers.

So now we are down to two options. The choice of Linux vs. BSD is an easy one for me: It does not matter. Both are excellent options for hosting a web server. Depending on the hosting provider you may have Linux or BSD options to choose from, so I just simply use what's available.

There are many Linux and BSD distributions out there, so how to pick one? This is largely a matter of preference, but if we are going to try to use what the majority use we can look at the distributions according to some of their major characteristics, and then pick a popular distribution in the group of choice:

- RPM based (RedHat, CentOS, Fedora)
- Debian based (Debian, Ubuntu, Mint)
- BSD based (FreeBSD, NetBSD, OpenBSD)
- others

I'm sure my simplistic grouping of distros will offend some. I'm also sure many will have the need to mention that Mac OS X is BSD based and as such should be in my list. Keep in mind that this classification is just my own, I don't expect everyone will agree with me.

As an exercise in preparing this article I tested a few Linux distros (I'm using a VPS that does not offer BSD choices). I tried Fedora, Ubuntu, Debian and CentOS. None of them was straightforward, unfortunately, but out of all of these, the CentOS install was the one with less complications, so that's the one I'm going to describe here.

If you have never done this you may find the setup task to be extremely laborious, and you would be correct if you think that. The thing is, once the server is setup it takes very little effort to keep it running.

Ready? Here we go!

Hosting on CentOS 6

I'm going to assume that we have a clean install of CentOS 6 on a dedicated server or VPS. A VM in your PC also works, but go ahead and do a standard install before starting.

Client side setup

We will be managing this server remotely from our own PC, so we need a tool that we can use to login into this system and run commands. If your PC runs Linux or OS X then you already have OpenSSH installed. If you are on Windows then there are two very decent ways to get SSH functionality:

- Cygwin (select the OpenSSH package in the installer)
- Putty

Cygwin is much more than SSH, it provides a Linux like environment inside your Windows box. Putty, on the other side, is just an SSH tool. Since I want to cover the most ground, the instructions below will be for Linux, OS X and Cygwin users. If you use Putty on Windows, then there is a little bit of

translation required.

Let's begin by logging in to our shiny new CentOS 6 box. Open a command line prompt on your PC (a Cygwin prompt if you are on Windows) and run the following:

```
$ ssh root@<your-server>
```

This is saying that you want to login to the system as user `root`, the admin of the system. You should replace `<your-server>` with the IP address or hostname of your server. Note the `$` is the command line prompt, you do not type that.

You will be asked to provide the password for user `root`. If this is a dedicated server or VPS that you purchased, you must have been asked about this password during the setup process. If this is your own VM then you chose a root password during installation.

Installing software packages

Now that we are logged in, we need to install all the software that we will be using to deploy our server, which obviously include Python and a web server. We will also switch from sqlite to a more robust MySQL database.

The command to install software on CentOS is called `yum`. We can install all the packages we need using a single command:

```
$ yum install python python-devel python-virtualenv httpd httpd-devel mysql-server
mysql-devel git gcc sudo
```

Some of these could be already installed, but it doesn't hurt to be safe and request all the packages we need, `yum` will only install what's missing.

The packages we just installed are:

- `python` and `python-devel`: the Python interpreter and its development package
- `virtualenv`: the Python script that creates virtual environments
- `httpd` and `httpd-devel`: The Apache web server and its development package
- `mysql-server` and `mysql-devel`: The MySQL database server and its development package
- `git`: source code version control system (we will use it to download and update the application)
- `gcc`: the C/C++ compiler (needed to compile Python extensions)
- `sudo`: a tool that helps users run commands as other users.

Creating a user account to host the application

We will now setup a user account on which we will host our application. In case the reasons aren't clear, the `root` account has the maximum privileges, so we could easily delete or destroy important parts of the system by mistake. If we work inside a restricted account then all the important parts of the

system will be out of reach so we are protected against mistakes. A second, maybe more important reason to not use the root account is that if a hacker gets hold of this account he will own our server. We want to hide the root account as much as possible, I like to disable root account logins completely on production servers.

We will call this new account `apps`, assuming we could host several applications inside it. Since the applications will be run by the web server account, we will make the group of our `apps` account the group of the web server account (which was created for us when Apache was installed), so that the web server has permission to read and write on these files. If you are not familiar with Unix user permissions, then an introduction is available over at this Wikipedia page.

The command to create the `apps` account on CentOS is:

```
$ adduser -g apache apps
```

The `-g` option makes the primary group of this account the `apache` group, which on CentOS is the group of the `apache` user account. As part of the account setup you will have to enter a password for this account.

The `adduser` command will create a home directory for the new account, at `/home/apps`. Unfortunately, when we go check the permissions of this directory we find it to only be accessible to the owner:

```
$ ls /home -l
total 7
drwx------ 2 apps  apache 4096 Apr  7 11:46 apps
```

We decided we want the `apache` group users to be able to freely work on the files on this account, so we have to give the group enough permissions:

```
$ chmod 775 /home/apps
```

Consult the documentation for the `chmod` tool for detailed information. The `775` permisions will allow the owner and the group full access, and will also give others permission to read and execute, but not write to this directory.

Password-less logins

The next step is to be able to login to this account from our PC without using a password. The SSH tool supports another type of authentication, called public key. If you are on a non-Windows OS it is very likely that you already have keys installed on your system.

Now open a terminal window on your own PC (Cygwin's bash shell for Windows users). To check if you have keys installed run the following command:

```
$ ls ~/.ssh
id_rsa  id_rsa.pub
```

If the directory listing shows files named `id_rsa` and `id_rsa.pub` like above then you are good to go. If you don't have these files then you have to run the following command:

```
$ keygen -t rsa
```

This application will prompt you to enter a few things, for which I recommend you accept the defaults by pressing Enter on all the prompts. If you know what you are doing and want to do otherwise, you are certainly welcome to.

After this command runs you should have the two files listed above. The file `id_rsa.pub` is your public key, the data in this file can be shared with others without creating a security risk. The `id_rsa` file is your private key, so you should never share the contents of this file with anyone.

We will now add our public key to our server, so that it knows who we are. Still on our local PC we start by printing the public key file:

```
$ cat ~/.ssh/id_rsa.pub
ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAABAQCjwaK4JVuY6PZAr8HocCIOszrLIzzCjO0Xlt9zkFNKvVpP1B92u3J
vwwiagqR+k0kHih2SmYnycmXjAcE60tvu+sIDA/7tEJZh4kO4nUYM5PJ17E+qTqUleBXQM74eITydq/USkO
qc5p+
+qUUgA60gUUuNum3igbZiNi71zK4m8g/IDywWYk+5vzNt2i7Sm8NEuauy/xWgnWhCBXZ/tXfkgWgC/4Hzpm
sfO+nniNh8VgTZp8Q+y+4psSE+p14qUg7KdDbf0Wo/D35wDkMvto96bIT8RF0np9dTkFj8TgNW8inP+6MC+
4vCd8F/NpESCVt8hRlBVERMF8Xv4f/0+7WT miguel@miguelspc
```

Now we need to copy this data to the clipboard and then switch to the CentOS system, where we will issue these commands:

```
$ mkdir /home/apps/.ssh
$ echo <paste-your-key-here> > /home/apps/.ssh/authorized_keys
```

For my server and public key the following commands are necessary:

```
$ mkdir /home/apps/.ssh
$ echo ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAABAQCjwaK4JVuY6PZAr8HocCIOszrLIzzCjO0Xlt9zkFNKvVpP1B92u3J
vwwiagqR+k0kHih2SmYnycmXjAcE60tvu+sIDA/7tEJZh4kO4nUYM5PJ17E+qTqUleBXQM74eITydq/USkO
qc5p+
+qUUgA60gUUuNum3igbZiNi71zK4m8g/IDywWYk+5vzNt2i7Sm8NEuauy/xWgnWhCBXZ/tXfkgWgC/4Hzpm
sfO+nniNh8VgTZp8Q+y+4psSE+p14qUg7KdDbf0Wo/D35wDkMvto96bIT8RF0np9dTkFj8TgNW8inP+6MC+
4vCd8F/NpESCVt8hRlBVERMF8Xv4f/0+7WT miguel@miguelspc >
/home/apps/.ssh/authorized_keys
```

These commands write your public key to the file `authorized_keys` in the server. This makes your public key known to OpenSSH on the server. The remaining task is to secure the `.ssh` directory and the `authorized_keys` file inside it:

```
$ chown -R apps:apache /home/apps/.ssh
$ chmod 700 /home/apps/.ssh
$ chmod 600 /home/apps/.ssh/authorized_keys
```

These commands change the ownership of these files to the `apps` account and then make the directory

and file only accessible by the new owner.

The password-less login should now be working. We will now logout of the `root` account of our server and then login again, but this time we will login as the `apps` user:

```
$ ssh apps@<your-server>
```

If everything goes well you should not need to enter a password to obtain access.

Installing the application

We will now use `git` to download and install `microblog` into the server. If you are not familiar with `git` I recommend that you read git for beginners.

The application must be available in a git server that can be reached from our web server. I will be using my own github hosted application. You are welcome to use it as well, or if you prefer you can also clone it and make one that is yours.

To install `microblog` in our server we just issue a `git clone` command:

```
$ git clone https://github.com/miguelgrinberg/microblog.git
$ cd microblog
```

The latest version of the application has a few changes from how we left it in the previous article.

First, there are a few new files at the top level, `runp-sqlite.fcgi`, `runp-mysql.fcgi`, `killpython` and `requirements.txt`. The `*.fcgi` scripts are starter scripts that are invoked by servers that use the FastCGI protocol. The `killpython` script will help us restart the application after we upgrade it. The `requirements.txt` file lists all the dependencies that we need to install to run our application. We will look at all these later.

Another interesting change is in our `config.py` file. Up to now we initialized our database as follows:

```
SQLALCHEMY_DATABASE_URI = 'sqlite:///' + os.path.join(basedir, 'app.db')
```

Now we have this instead:

```
if os.environ.get('DATABASE_URL') is None:
    SQLALCHEMY_DATABASE_URI = 'sqlite:///' + os.path.join(basedir, 'app.db')
else:
    SQLALCHEMY_DATABASE_URI = os.environ['DATABASE_URL']
```

This is a simple change that enables us to override the database the application will use by setting an environment variable. You will see how we take advantage of this change in the next section.

Setting up MySQL

The sqlite database that we've been using all this time is great for simple applications, but when we get into a full blown web server that can serve multiple requests at a time it is a better idea to use a more

robust database. So we will setup a MySQL database for `microblog`.

We have installed MySQL already, so all that is left is to create a database and a user that has permissions on it. To manage our database server we use the `mysql` tool:

```
$ mysql -u root -p
Enter password: (enter the mysql root password, or empty if one is not defined)
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 11
Server version: 5.1.67 Source distribution

Copyright (c) 2000, 2012, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

And now that we are in the mysql prompt we can go ahead and create a database and a user, both named `apps`:

```
mysql> create database apps character set utf8 collate utf8_bin;
mysql> create user 'apps'@'localhost' identified by 'apps';
mysql> grant all privileges on apps.* to 'apps'@'localhost';
mysql> flush privileges;
mysql> quit;
```

Note that you will need to pick a password for the `apps` MySQL user in the `identified by` section. For simplicity I have picked `apps` as password, but on a real installation a hard to guess password should be used. Do not confuse the database `apps` user with the system's `apps` user.

Initializing the application

Now that we have a database ready we can initialize `microblog`.

We start by creating and populating the Python virtualenv:

```
$ virtualenv flask
$ flask/bin/pip install -r requirements.txt
$ flask/bin/pip install mysql-python
```

Note that in addition to installing all the dependencies listed in the `requirements.txt` file we add the MySQL support, which is required by SQLAlchemy when connecting to that database server.

Then we create the database:

```
$ DATABASE_URL=mysql://apps:apps@localhost/apps ./db_create.py
```

Note how we set an environment variable to point to our new MySQL database instead of the sqlite default.

Next we compile all the messages in our translation database:

```
$ ./tr_compile.py
```

And to finalize, we enable group write access to the two folders that need to be written to by the web server:

```
$ chmod -R g+w search.db tmp
```

The `search.db` folder is used by our Whoosh full text search database. The `tmp` directory should be left open so that new files can be created there when needed.

Setting up Apache

The last thing to do is to configure our Apache web server.

We wil be using the `mod_fcigd` module to handle the FastCGI communication with our application. Many Linux distributions offer a package for `mod_fcgid`, but unfortunately CentOS is not one of them, so we will build this module from its source code.

Here is the list of commands that build and install the current release of `mod_fcgid`:

```
$ wget http://mirror.metrocast.net/apache/httpd/mod_fcgid/mod_fcgid-2.3.9.tar.gz
$ tar xvzf mod_fcgid-2.3.9.tar.gz
$ cd mod_fcgid-2.3.9
$ APXS=/usr/sbin/apxs ./configure.apxs
$ su
(enter root password)
$ make install
```

You can consult the `mod_fcgid` documentation if you want details on the above commands.

With this module installed we can now configure our server. For this we need to edit the Apache configuration file. Linux distributions do not agree on a standard location and name for this file, so you will need to figure out where this file is. On CentOS, the config file is located at `/etc/httpd/conf/httpd.conf`.

The changes that we need to make are simply to define our server. At the bottom of the configuration file we add the following (from the root account, since this file is not writable to regular users):

```
FcgidIPCDir /tmp
AddHandler fcgid-script .fcgi
<VirtualHost *:80>
    DocumentRoot /home/apps/microblog/app/static
    Alias /static /home/apps/microblog/app/static
    ScriptAlias / /home/apps/microblog/runp-mysql.fcgi/
</VirtualHost>
```

The `FcgidIPCDir` sets the directory where the file sockets will be created. I've found that on CentOS this was going by default to a directory where the `apache` user did not have write permissions, so I'm putting all these files on `/tmp` instead.

Next we use `AddHandler` to tell apache that any files that have a `.fcgi` extension should be treated as FastCGI files and routed through the `mod_fcgid` module that we just installed. Remember the new `runp-mysql.fcgi` file that we have in our root folder? This file uses the `flipflop` Python module as an adapter so that our application can speak the FastCGI protocol. Let's have a quick look at this file:

```
#!flask/bin/python
import os
os.environ['DATABASE_URL'] = 'mysql://apps:apps@localhost/apps'

from flipflop import WSGIServer
from app import app

if __name__ == '__main__':
    WSGIServer(app).run()
```

Apache will execute this script to start our application. Note how we insert the MySQL database name in the environment, so that Apache can see it.

The `<VirtualHost>` section defines the host that will run our web server. The `*:80` denomination indicates that any requests that arrive at the server for any hostnames on port 80 will be handled by this virtual server. The configuration can have multiple `<VirtualHost>` sections, each defining a different server. To differentiate the servers you can use different IP addresses, domain or sub-domain names or ports.

The definition of our virtual host is pretty simple. The `DocumentRoot` statement tells Apache where to look for static files. Any requests for files will be served our of this folder, so for example, when browsers ask for a `/favicon.ico` file to show the little icon next to the URL Apache will try to find it here. Unfortunately all the static files that we use in our application have a `/static` prefix, so to prevent Apache from looking for another `static` folder we use the `Alias` statement, which just says that anything that starts with `/static` should go directly to our folder. Finally, the `ScriptAlias` statement tells Apache that when a request that starts with `/` arrives (basically all requests that are not for static files) then use the script indicated as second argument to handle it, and that is our `.fcgi` script.

The Apache config file supports many more options that I will not mention here. I recommend that you review the Apache documentation to decide what options make sense for your server.

To activate the changes we made we need to restart the apache server, again from the root account:

```
$ service httpd restart
```

And now from the browser on your PC you should be able to access microblog at the address `http://<your-server>`.

Installing application updates

The last remaining thing we are going to look at for this server is how to roll out an update of the application.

Let's assume we have deployed the application and it has been running happily for a while on our server. Now it's time to push an update, which could fix some bugs we found, or just add features.

We will use `git` and our own tools for this. If we login to the web server using the `apps` account we can upgrade to the latest release from the git repository by running:

```
$ cd microblog
$ git pull
$ DATABASE_URL=mysql://apps:apps@localhost/apps ./db_upgrade.py
$ ./tr_compile.py
```

The `git pull` operation downloads any new or updated files from our git server. We then upgrade our database and recompile our translation files, in case any of those need updating. That's it!

Once we have the updates in place we need to tell apache to restart the FastCGI processes, and this gets a bit tricky, because we are logged in as user `apps`, an unprivileged account, while the FastCGI processes are owned by the `apache` user.

For this we are going to use `sudo`. The `sudo` tool allows users to selectively run applications as other users. We don't really want to give our `apps` account a lot of power, we will limit it to just be able to send kill signals to the FastCGI processes started by the apache user.

The `killpython` script that we have in the `microblog` directory does this work:

```
killall /home/apps/microblog/flask/bin/python
```

The problem is, if we run this under the `apps` account we won't have permission to kill the processes owned by the `apache` user. To enable `apps` to kill these processes we have to make a small change in file `/etc/sudoers`:

```
apps    ALL=(apache) NOPASSWD:/home/apps/microblog/killpython
Defaults: apps    !requiretty
```

Note you will need to make this change as user `root`, since `/etc/sudoers` is a system file.

The first cryptic command gives the `apps` user permission to run the `killpython` command as user `apache` without having to provide a password. The second line allows `sudo` commands for user `apps` to be issued from a script in addition to doing it from an interactive console. You can read the man page for `sudoers` for detailed information on the syntax of `sudo` configuration file.

Now when logged in as `apps` we can kill the python jobs as follows:

```
$ sudo -u apache ./killpython
```

If the FastCGI processes are killed Apache will restart them the next time a request comes for

`microblog`.

To make upgrading our server a bit more simple we can write a client side script that upgrades and restarts all together. The script would run the following:

```
ssh apps@<your-server> "cd microblog;git pull;sudo -u apache
./killpython;DATABASE_URL=mysql://apps:apps@localhost/apps
./db_upgrade.py;./tr_compile.py"
```

If you store the above line in a script then you can roll out an application upgrade with a single command!

What's left to do

I will not discuss a few more mundane operations that are recommended to make a server that is open to the hostile Internet world more secure. Things such as:

- Not allowing `root` remote logins
- Disabling any services that are not used, like FTP, CIFS, etc.
- Setting up a firewall
- Keep the server up to date on security fixes.
- Etc.

I will leave these as an exercise to the interested reader, since they are largely unrelated to this tutorial.

Hosting on the Raspberry Pi

The Raspberry Pi is a revolutionary little Linux computer that costs $35. It has very low power consumption so it is the perfect device to host a home based web application that can be online 24/7 without tying up a full blown computer.

There are several Linux distributions that run on the Raspberry Pi. We'll set up our application on Raspbian, the official distribution.

As a side benefit, note that Raspbian is a derivative of Debian, so the instructions below will apply with little or no modification for Debian/Ubuntu based servers.

We will now follow the same steps we followed for CentOS to get the RPi server setup.

Client side setup

A Raspberry Pi is a pretty decent computer to work in. You can connect an HDMI monitor, keyboard and mouse and work directly on it to configure our web server. Also like we did for CentOS or actually any other Linux distribution you can have the Raspberry Pi just connected to the network and then access it remotely from a PC over `ssh`. See the client side setup section above for CentOS for instructions on how to setup a client PC with `ssh`.

Note that I will not cover how to install and setup the Raspberry Pi here, there are plenty of articles

elsewhere for that. I will assume the Raspberry Pi is already running and connected to the network.

Installing software packages

Since the Pi is a limited power machine we will not do a full blown Apache/MySQL installation like we did for CentOS. Instead, we will take a lightweight approach. For web server we will use Lighttpd (pronounced "Lighty"), a small web server with a very good and efficient FastCGI implementation. For our database we will stay with sqlite.

With CentOS we used `yum` to install packages. In Debian derivatives the package manager is called `apt-get`:

```
$ sudo apt-get update
$ sudo apt-get upgrade
$ sudo apt-get install python python-dev python-virtualenv lighttpd git
```

The `update` line downloads an updated package list to your Pi. The `upgrade` line upgrades all installed packages to their latest versions. It is recommended that you issue these two commands regularly, to keep your Raspberry Pi's software up to date. Finally the `install` statement installs the packages that we need for our task.

Creating a user account to host the application

We said that we will assume our Raspberry Pi server will be in a trusted network. Since security is not a major concern we will not bother creating a dedicated account and instead will install the app directly on the default `pi` account.

If your Raspberry Pi will be connected to the internet then you should follow the approach we used for CentOS and make a different account that has no privileges, as this ensure that if an intruder manages to access the account the damage that he or she can cause will be limited.

Password-less logins

Having a public key authentication procedure is also less important if we assume the application will run on a small intranet, but should you need such functionality the procedure is identical to what we did for the CentOS system.

Installing the application

The application installs very easily, again using `git`:

```
$ git clone git://github.com/miguelgrinberg/microblog.git
$ cd microblog
```

Initializing the application

Since we will be using a sqlite database in this host we can go ahead and initialize `microblog` as we have done in past articles for our development server:

```
$ virtualenv flask
$ flask/bin/pip install -r requirements.txt
% flask/bin/pip install mysql-python
$ ./db_create.py
$ ./tr_compile.py
```

Note that the `setup.py` script will try to install the MySQL support and that will fail, but this is all right since we will not use it.

The files were all checked out under the `pi` user ownership. But as we have seen for CentOS, the web server on the Pi will run under a different account. Since security isn't a concern for this server, we will take a much simpler approach:

```
$ chmod -R 777 *
```

The above `chmod` statement makes all the files writable by all users. Again, this would be a really bad thing to do on an open server, but for a server where you have full control it saves you from the trouble of dealing with user permissions.

I want to make it clear that the Raspberry Pi is perfectly capable of supporting a configuration based on user groups and user permissions similar to what we created above for the CentOS box. We are choosing this simplified setup because we will be using this server in a controlled environment.

Setting up Lighttpd

Lighttpd comes with native support for FastCGI, so we do not need to worry about installing a separate module for it.

All we need to do is append the definition of our website to the configuration file, which is located at `/etc/lighttpd/lighttpd.conf`:

```
fastcgi.server = ("/microblog" =>
    ((
        "socket" => "/tmp/microblog-fcgi.sock",
        "bin-path" => "/home/pi/microblog/runp-sqlite.fcgi",
        "check-local" => "disable",
        "max-procs" => 1
    ))
)

alias.url = (
    "/microblog/static/" => "/home/pi/microblog/app/static/",
)
```

And with this our application should be online at `http://aaa.bbb.ccc.ddd/microblog`, with `aaa.bbb.ccc.ddd` being the IP address of your Raspberry Pi.

But there are a few things to note in the above configuration, and also a couple of things that didn't initially work and required some changes in the application side.

The `fastcgi.server` statement is the one that defines the behavior of our FastCGI server, which

we are exposing under the `/microblog` URL. The reason we are not exposing the application at the root URL is simple, we may want to host more than one application, so putting all the URLs under a `/microblog` root effectively works as a namespace.

Inside the FastCGI definition we provide a socket name in the `/tmp` directory and the path to our `runp-sqlite.fcgi` file, which Lighttpd will execute to start up FastCGI processes. The `check-local` option tells Lighttpd to send requests to the FastCGI server even if the request path matches a file on disk. The `max-procs` limits the number of FastCGI processes to 1, which for a small server is enough and avoids potential problems with multiple concurrent writers to the sqlite database.

The `alias.url` section provides the mapping that enables Lighttpd to server our static files. The alias maps any requests for `/microblog/static/...` to the correct location on disk where we store our static files.

The `runp-sqlite.fcgi` script is pretty much the same as the one we used for CentOS, but without overriding the database setting:

```
from flipflop import WSGIServer
from app import app

if __name__ == '__main__':
    WSGIServer(app).run()
```

Some necessary fixes

Testing this setup reveals a couple of bugs in our application.

First, our javascript and css static files are not being served. The problem becomes evident after inspecting the HTML source of the login page. The `base.html` template references these files via hardcoded paths that begin with `/static`, but we said are putting this application inside a `/microblog` prefix. For example, see how we had the link to the CSS file:

```
<link href="/static/css/bootstrap.min.css" rel="stylesheet" media="screen">
```

Here we really want `/microblog/static/bootstrap.min.css`, but of course we can't add `/microblog` to the `base.html` file because that will break the development server that runs on our PC. The solution is to let Flask generate these URLs using our old friend `url_for`:

```
<link href="{{ url_for('.static', filename='css/bootstrap.min.css') }}"
rel="stylesheet" media="screen">
```

After updating all the js and css files as above the site serves the login page with all the auxiliary static files.

But trying to login reveals another problem, Right after logging in we get a 500 error code. Luckily for us we have logging, so a quick look at `/home/pi/microblog/tmp/microblog.log` shows the following error:

```
ProgrammingError: (ProgrammingError) SQLite objects created in a thread can only be
used in that same thread.The object was created in thread id -1260325776 and this
is thread id -1243548560
```

What is this? We are not running multiple threads ourselves but obviously there are multiple threads in our application. The only difference between our development server and this one is that we are now using `flipflop` for the FastCGI server. Looking at the code in this module we can easily find that by default `flipflop` runs a multithreaded web server.

A possible solution would be to use the single-threaded FastCGI server from `flipflop` but that could affect the performance if the server will have multiple concurrent users. Another, more interesting way to handle this problem is to enable threading in `sqlite` (which appears to be fully supported). Multithreading can be enabled in SQLAlchemy by setting the `check_same_thread` option to `False` in our `config.py` configuration file:

```
SQLALCHEMY_DATABASE_URI = 'sqlite:///' + os.path.join(basedir, 'app.db') + '?
check_same_thread=False'
```

With this change we are allowing multiple threads to make changes to the database, the sqlite library takes care of synchronizing multiple accesses.

And with these small changes we have a fully enabled web application running on our Raspberry Pi!

Installing application updates

To deploy application updates we can just login to the Raspberry Pi and issue the commands that update the source files, database and translation files:

```
$ cd microblog
$ git pull
$ ./db_upgrade.py
$ ./tr_compile.py
```

If you setup password-less login to your Pi then of course you can write a script wrapper that does all this in a single command.

Final words

The updated application is available, as always, on my github page. Alternatively you can download it as a zip file below:

Download microblog 0.17.

In the next article we will be looking at deployment on cloud services as an alternative to the traditional options we explored today.

I will see you then. Thank you!

Miguel

Part XVIII: Deployment on the Heroku Cloud

In the previous article we explored traditional hosting options. We've looked at two actual examples of deployment to Linux servers, first to a CentOS system and later to the Raspberry Pi credit card sized computer. Those that are not used to administer a Linux system probably thought the amount of effort we had to put into the task was huge, and that surely there must be an easier way.

Today we will try to see if deploying to the *cloud* is the answer to the complexity problem.

But what does it mean to "deploy to the cloud"?

A cloud hosting provider offers a platform on which an application can run. All the developer needs to provide is the application, because the rest, which includes the hardware, operating system, scripting language interpreters and database, is managed by the service.

Sounds too good to be true, right?

We'll look at deploying to Heroku, one of the most popular cloud hosting services. I picked Heroku not only because it is popular, but also because it has a free service level, so we get to host our application without having to spend any money. If you want to find information about this type of services and what other providers are out there you can consult the Wikipedia page on platform as a service.

Hosting on Heroku

Heroku was one of the first platform as a service providers. It started as a hosting option for Ruby based applications, but then grew to support many other languages like Java, Node.js and our favorite, Python.

In essence, deploying a web application to Heroku requires just uploading the application using `git` (you'll see how that works in a moment). Heroku looks for a file called `Procfile` in the application's root directory for instructions on how to execute the application. For Python projects Heroku also expects a `requirements.txt` file that lists all the module dependencies that need to be installed.

After the application is uploaded you are essentially done. Heroku will do its magic and the application will be online within seconds. The amount of money you pay directly determines how much computing power you get for your application, so as your application gets more users you will need to buy more units of computing, which Heroku calls "dynos", and that is how you keep up with the load.

Ready to try Heroku? Let's get started!

Creating Heroku account

Before we can deploy to Heroku we need to have an account with them. So head over to heroku.com and create an account.

Once you are logged in you have access to a dashboard, where all your apps can be managed. We will

not be using the dashboard much though, but it provides a nice view of your account.

Installing the Heroku client

Even though it is possible to manage applications from the Heroku web site to some extent, there are some things that can only be done from the command line, so we'll just do everything there.

Heroku offers a tool called the "Heroku client" that we'll use to create and manage our application. This tool is available for Windows, Mac OS X and Linux. If there is a Heroku toolbelt download for your platform then that's the easiest way to get the Heroku client tool installed.

The first thing we should do with the client tool is to login to our account:

```
$ heroku login
```

Heroku will prompt for your email address and your account password. The first time you login it will send your public SSH key to the Heroku servers.

Your authenticated status will be remembered in subsequent commands.

Git setup

The `git` tool is core to the deployment of apps to Heroku, so it must also be available. If you installed the Heroku toolbelt then you already have it as part of that installation.

To deploy to Heroku the application must be in a local `git` repository, first so let's get one set up:

```
$ git clone -b version-0.18 git://github.com/miguelgrinberg/microblog.git
$ cd microblog
```

Note that we are choosing a specific branch to be checked out, this is the branch that has the Heroku integration.

Creating a Heroku app

To create a new Heroku app you just use the `create` command from the root directory of the application:

```
$ heroku apps:create flask-microblog
Creating flask-microblog... done, stack is cedar
http://flask-microblog.herokuapp.com/ | git@heroku.com:flask-microblog.git
```

In addition to setting up a URL this command adds a git remote to our `git` repository that we will soon use to upload the application.

Of course the name `flask-microblog` is now taken by me, so make sure you use a different app name if you are doing this along.

Eliminating local file storage

Several of the functions of our application rely on writing data to disk files.

Unfortunately we have a tricky problem with this. Applications that run on Heroku are not supposed to write permanent files to disk, because Heroku uses a virtualized platform that does not remember data files, the file system is reset to a clean state that just contains the application script files each time a virtual worker is started. Essentially this means that the application can write temporary files to disk, but should be able to regenerate those files should they disappear. Also when two or more workers (dynos) are in use each gets its own virtual file system, so it is not possible to share files among them.

This is really bad news for us. For starters, it means we cannot use sqlite as a database, and our Whoosh full text search database will also fail to work, since it writes all its data to files. We also have the compiled translation files for Flask-Babel, which are generated when running the `tr_compile.py` script. And yet another area where there is problem is logging, we used to write our logfile.to the `tmp` folder and that is also not going to work when running on Heroku.

We have identified four major problems for which we need to try to find solutions.

For our first problem, the database, we'll migrate to Heroku's own database offering, which is based on PostgreSQL.

For the full text search functionality we don't have a readily available alternative. We could re-implement full text searches using PostgreSQL functionality, but that would require several changes to our application. It is a pity, but solving this problem now would be a huge distraction, so for now we'll disable full text searches when running under Heroku.

To support translations we are going to include the compiled translation files in the git repository, that way these files will be persistant in the file system.

Finally, since we can't write our own log file, we'll add our logs to the logger that Heroku uses, which is actually simple, since Heroku will add to its log anything that goes to `stdout`.

Creating a Heroku database

To create a database we use the Heroku client:

```
$ heroku addons:add heroku-postgresql:dev
Adding heroku-postgresql:dev on flask-microblog... done, v3 (free)
Attached as HEROKU_POSTGRESQL_ORANGE_URL
Database has been created and is available
 ! This database is empty. If upgrading, you can transfer
 ! data from another database with pgbackups:restore.
Use `heroku addons:docs heroku-postgresql:dev` to view documentation.
$ heroku pg:promote HEROKU_POSTGRESQL_ORANGE_URL
Promoting HEROKU_POSTGRESQL_ORANGE_URL to DATABASE_URL... done
```

Note that we are adding a development database, because that is the only database offering that is free. A production web server would need one of the production database options.

And how does our application know the details to connect to this database? Heroku publishes the URI to the database in the `$DATABASE_URL` environment variable. If you recall, we have modified our configuration to look for this variable in the previous hosting article, so the changes are already in place

to connect with this database.

Disabling full text searches

To disable full text searches we need our application to be able to know if it is running under Heroku or not. For this we will add a custom environment variable, again using the Heroku client tool:

```
heroku config:set HEROKU=1
```

The `HEROKU` environment variable will now be set to `1` when our application runs inside the Heroku virtual platform.

Now it is easy to disable the full text search index. First we add a configuration variable (file `config.py`):

```
# Whoosh does not work on Heroku
WHOOSH_ENABLED = os.environ.get('HEROKU') is None
```

Then we suppress the creation of the full text database instance (file `app/models.py`):

```
from config import WHOOSH_ENABLED

enable_search = WHOOSH_ENABLED
if enable_search:
    import flask.ext.whooshalchemy as whooshalchemy

# ...
if enable_search:
    whooshalchemy.whoosh_index(app, Post)
```

Precompiled Tranlations

This one is pretty easy. After running `tr_compile.py` we end up with a `<language>.mo` file for each `<language>.po` source file. All we need to do is add the `mo` files to the git repository, and then in the future we'll have to remember to update them. The `mo` file for Spanish is included in the branch of the git repository dedicated to this article.

Fixing the logging

Under Heroku, anything that is written to `stdout` is added to the Heroku application log. But logs written to a disk file will not be accessible. So on this platform we will suppress the file log and instead use a log that writes to `stdout` (file `app/__init__.py`):

```
if not app.debug and os.environ.get('HEROKU') is None:
    import logging
    from logging.handlers import RotatingFileHandler
    file_handler = RotatingFileHandler('tmp/microblog.log', 'a', 1 * 1024 * 1024,
10)
    file_handler.setLevel(logging.INFO)
    file_handler.setFormatter(logging.Formatter('%(asctime)s %(levelname)s: %
(message)s [in %(pathname)s:%(lineno)d]'))
    app.logger.addHandler(file_handler)
```

```
    app.logger.setLevel(logging.INFO)
    app.logger.info('microblog startup')

if os.environ.get('HEROKU') is not None:
    import logging
    stream_handler = logging.StreamHandler()
    app.logger.addHandler(stream_handler)
    app.logger.setLevel(logging.INFO)
    app.logger.info('microblog startup')
```

The web server

Heroku does not provide a web server. Instead, it expects the application to start its own server on the port number given in environment variable `$PORT`.

We know the Flask web server is not good for production use because it is single process and single threaded, so we need a better server. The Heroku tutorial for Python suggests gunicorn, a pre-fork style web server written in Python, so that's the one we'll use.

For our local environment `gunicorn` installs as a regular python module into our virtual environment:

```
$ flask/bin/pip install gunicorn
```

To start this browser we need to provide a single argument that names the Python module that defines the application and the application object, both separated by a colon. Now for example, if we wanted to start a local `gunicorn` server with this module we would issue the following command:

```
$ flask/bin/gunicorn --log-file - app:app
2013-04-24 08:42:34 [31296] [INFO] Starting gunicorn 19.1.1
2013-04-24 08:42:34 [31296] [INFO] Listening at: http://127.0.0.1:8000 (31296)
2013-04-24 08:42:34 [31296] [INFO] Using worker: sync
2013-04-24 08:42:34 [31301] [INFO] Booting worker with pid: 31301
```

The requirements file

Soon we will be uploading our application to Heroku, but before we can do that we have to inform the server what dependencies the application needs to run. We created a `requirements.txt` file in the previous chapter, to simplify the installation of dependencies in a dedicated server, and the good news is that Heroku also imports dependencies from a requirements file.

The `gunicorn` web server needs to be added to the list, and so is the `psycopg2` driver, which is required by SQLAlchemy to connect to PostgreSQL databases. The final `requirements.txt` file looks like this:

```
Babel==1.3
Flask==0.10.1
Flask-Babel==0.9
Flask-Login==0.2.11
Flask-Mail==0.9.0
Flask-OpenID==1.2.1
```

```
Flask-SQLAlchemy==2.0
Flask-WTF==0.10.2
Flask-WhooshAlchemy==0.56
Jinja2==2.7.3
MarkupSafe==0.23
SQLAlchemy==0.9.7
Tempita==0.5.2
WTForms==2.0.1
Werkzeug==0.9.6
Whoosh==2.6.0
blinker==1.3
coverage==3.7.1
decorator==3.4.0
flipflop==1.0
guess-language==0.2
gunicorn==19.1.1
itsdangerous==0.24
pbr==0.10.0
psycopg2==2.5.4
python-openid==2.2.5
pytz==2014.7
six==1.8.0
speaklater==1.3
sqlalchemy-migrate==0.9.2
sqlparse==0.1.11
```

Some of these modules will not be needed in the Heroku version of our application, but it really doesn't hurt to have extra stuff, to me it seems better to have a complete requirements list.

The Procfile

The last requirement is to tell Heroku how to run the application. For this Heroku requires a file called `Procfile` in the root folder of the application.

This file is extremely simple, it just defines process names and the commands associated with them (file `Procfile`):

```
web: gunicorn app:app
init: python db_create.py
upgrade: python db_upgrade.py
```

The `web` label is associated with the web server. Heroku expects this task and will use it to start our application.

The other two tasks, named `init` and `upgrade` are custom tasks that we will use to work with our application. The init task initializes our application by creating the database. The upgrade task is similar, but instead of creating the database from scratch it upgrades it to the latest migration.

Deploying the application

And now we have reached the most interesting part, where we push the application to our Heroku hosting account. This is actually pretty simple, we just use `git` to push the application:

```
$ git push heroku master
Counting objects: 307, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (168/168), done.
Writing objects: 100% (307/307), 165.57 KiB, done.
Total 307 (delta 142), reused 272 (delta 122)

-----> Python app detected
-----> No runtime.txt provided; assuming python-2.7.4.
-----> Preparing Python runtime (python-2.7.4)
-----> Installing Distribute (0.6.36)
-----> Installing Pip (1.3.1)
-----> Installing dependencies using Pip (1.3.1)
...
-----> Discovering process types
       Procfile declares types -> init, upgrade, web

-----> Compiled slug size: 29.6MB
-----> Launching... done, v6
       http://flask-microblog.herokuapp.com deployed to Heroku

To git@heroku.com:flask-microblog.git
 * [new branch]      master -> master
```

The label `heroku` that we used in the `git push` command was automatically registered with our `git` repository when we created our application with `heroku create`. To see how this remote repository is configured you can run `git remote -v` in the application folder.

The first time we push the application to Heroku we need to initialize the database and the translation files, and for that we can execute the `init` task that we included in our `Procfile`:

```
$ heroku run init
Running `init` attached to terminal... up, run.7671
/app/.heroku/python/lib/python2.7/site-packages/sqlalchemy/engine/url.py:105:
SADeprecationWarning: The SQLAlchemy PostgreSQL dialect has been renamed from
'postgres' to 'postgresql'. The new URL format is
postgresql[+driver]://<user>:<pass>@<host>/<dbname>
  module = __import__('sqlalchemy.dialects.%s' % (dialect, )).dialects
```

The deprecation warning comes from SQLAlchemy, because it does not like that the URI starts with `postgres://` instead of `postgresql://`. This URI comes from Heroku via the `$DATABASE_URL` environment variable, so we really don't have any control over this. Let's hope this continues to work for a long time.

Believe it or not, now the application is online. In my case, the application can be accessed at http://flask-microblog.herokuapp.com. For example, you can become my follower from my profile page. I'm not sure how long I'll leave it there, but feel free to give it a try if you can connect to it!

Updating the application

The time will come when an update needs to be deployed. This works in a similar way to the initial deployment. First the application is pushed from `git`:

```
$ git push heroku master
```

Then the upgrade script is executed:

```
$ heroku run upgrade
```

Logging

If a problem occurs then it may be useful to see the logs. Recall that for the Heroku hosted version we are writing our logs to `stdout` which Heroku collects into its own logs.

To see the logs we use the Heroku client:

```
$ heroku logs
```

The above command will show all the logs, including Heroku ones. To only see application logs we can use this command:

```
$ heroku logs --source app
```

Things like stack traces and other application errors will appear in these app logs.

Is it worth it?

We've now seen what it takes to deploy to a cloud hosting service so we can now compare against the traditional hosting.

The simplicity aspect is easily won by cloud. At least for Heroku the deployment process was extremely simple. When deploying to a dedicated server or VPS there are a lot of administrative tasks that need to be done to prepare the system. Heroku takes care of all that and allows us to concentrate on our application.

The price is where it is harder to come to a conclusion. Cloud offerings are more expensive than dedicated servers, since you are not only paying for the server but also for the admin work. A pretty basic production service with Heroku that includes two dynos and the least expensive production database costs $85 per month at the time I'm writing this. On the other side, if you look hard you can find well provisioned VPS servers for abour $40 per year.

In the end, I think it all comes down to what is most important to you, time or money.

The End?

The updated application is available, as always, on my github page. Alternatively you can download it as a zip file below:

Download microblog 0.18.

With our application deployed in every possible way it feels like we are reaching the end of this journey.

I hope these articles were a useful introduction to the development of a real world web application project, and that the knowledge dump I've made over these eighteen articles motivates you to start your own project.

I'm not closing the door to more `microblog` articles. If and when an interesting topic comes to mind I will write more, but I expect the rate of updates from now on will slow down a bit. From time to time I may make small updates to the application that don't deserve a blog post, so you may want to watch the project on github to catch these.

I will continue blogging about topics related to web development and software in general, so I invite you to connect via Twitter or Facebook if you haven't done it yet, so that you find my future articles.

Thank you, again, for being a loyal reader.

Miguel