Web Programming with Python (Django)

Sutee Sudprasert

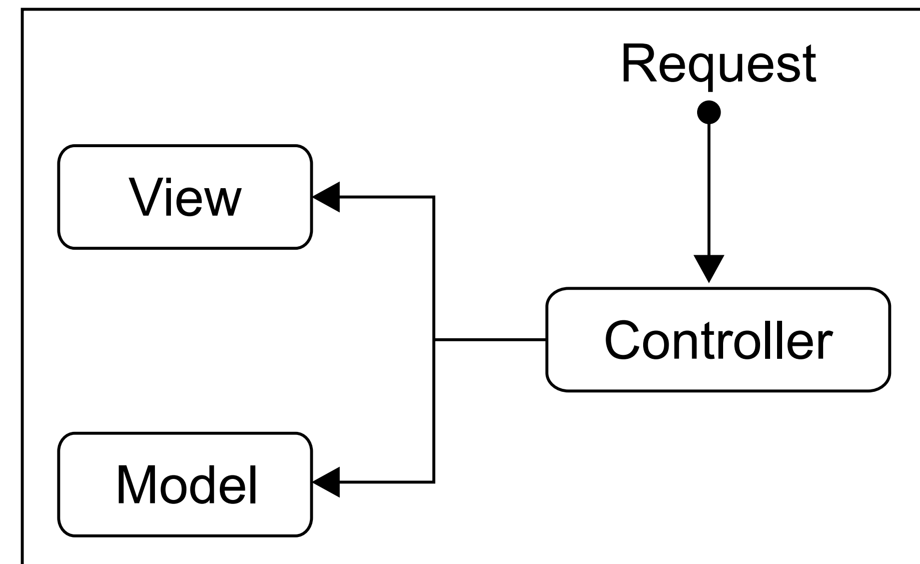
ลอกมาจากหนังสือ Django 1.0 Web Site Development - Ayman Hourieh

MVC pattern in web development

- Old and clumsy ways for web development
  - Common Gateway Interface
    - be difficult to work with
    - require a separate copy of the program to be launched for each request
  - Script languages such as Perl and PHP
    - lacked unified frameworks

MVC pattern in web development

- The better way
  - model-view-controller (MVC) pattern for web development
    - Model: data
    - User interface: view
    - Data-handling logic: controller



Why Python?

- A clean and elegant syntax
- A large standard library of modules that covers a wide range of tasks
- Extensive documentation
- A mature runtime environment
- Support for standard and proven technologies such as Linux and Apache

Why Django?

- Python web application frameworks
  - Turbogears
  - Grok
  - Pyjamas
  - and mores
    - http://en.wikipedia.org/wiki/Comparison_of_Web_application_frameworks#Python

Why Django?

- Tight integration between components
- Object-Relation Mapper
- Clean URL design
- Automatic administration interface
- Advanced development environment
- Multilingual support

History of Django

The Web framework for perfectionists with deadlines

Installing Django

- https://www.djangoproject.com/download/
  - extract and run: `python setup.py install`
- For Windows:
  - copy the `django-admin.py` file from `Django-x.xx/django/bin` to somewhere in your system path such as `c:\windows`
- You can test your installation by running this command:
  - `django-admin.py --version`

Setting up the database

- Edit `settings.py` file

```
DATABASE_ENGINE = 'django.db.backends.sqlite3'
DATABASE_NAME = 'bookmarks.db'
DATABASE_USER = ''
DATABASE_PASSWORD = ''
DATABASE_HOST = ''
DATABASE_PORT = ''
```

- Run `python manage.py syncdb`
    - Enter username, email, and password for the superuser account

Launching the development server

- Run `python manage.py runserver`
- open your browser, and navigate to http://localhost:8000/
- you can specify port in the command line
  - `python manage.py runserver <port number>`

Building a Social Bookmarking Application

Topics

- URLs and views: Creating the main page
- Models: Designing an initial database schema
- Templates: Creating a template for the main page
- Putting it all together: Generating user pages

MTV framework

- In Django
  - model = model
  - template = view
  - view = controller

Creating the main page view

- create a Django application inside our project
  - `python manage.py startapp bookmarks`
- it will create a folder named `bookmarks` with these files
  - `__init__.py`
  - `views.py`
  - `models.py`

Creating the main page view

- Open the file `bookmarks/views.py` and enter the following:

```
from django.http import HttpResponse
def main_page(request):
  output = u'''
    <html>
      <head><title>%s</title></head>
      <body>
        <h1>%s</h1><p>%s</p>
      </body>
    </html>
  ''' % (
      u'Django Bookmarks',
      u'Welcome to Django Bookmarks',
      u'Where you can store and share bookmarks!'
  )
  return HttpResponse(output)
```

Creating the main page URL

- Open file `urls.py` and add an entry for the main page

```
from django.conf.urls.defaults import *
from bookmarks.views import *

urlpatterns = patterns('',
  r '^$', main_page),
)
```

Models: designing an initial database schema

- Django abstracts access to database table through Python classes
- For our bookmarking application, we need to store three types of data in the database:
  - Users (ID, username, password, email)
  - Links (ID, URL)
  - Bookmarks (ID, title, user_id, link_id)

The link data model

- Open the `bookmarks/models.py` file and type the following code:

```
from django.db import models

class Link(models.Model):
  url = models.URLField(unique=True)
```

A partial table of the field types

Field type	Description
`IntegerField`	An integer
`TextField`	A large text field
`DateTimeField`	A date and time field
`EmailField`	An email field with 75 characters max.
`URLField`	A URL field with 200 characters max.
`FileField`	A file-upload field

http://docs.djangoproject.com/en/dev/ref/models/fields/

The link data model

- การที่จะเริ่มใช้ model ของ application ที่เพิ่มเข้าไปใหม่ จำเป็น activate ตัว application ก่อน โดยการเพิ่มในไฟล์ settings.py

```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'django_bookmarks.bookmarks',
)
```

- จากนั้นจึงสั่ง

```
python manage.py syncdb
```

- สามารถดู SQL ที่ถูกสร้างได้จาก

```
python manage.py sql bookmarks
```

The link data model

- To explore the API, we will use the Python interactive console. To launch the console, use the following command:

```
python manage.py shell
```

- Using the Link model:

```
>>> from bookmarks.models import *
>>>
>>> link1 = Link(url=u'http://www.packtpub.com/')
>>> link1.save()
>>> link2 = Link(url=u'http://www.example.com/')
>>> link2.save()
>>> link2.url
>>>
>>> Link.objects.all()
>>> Link.objects.get(id=1)
>>>
>>> link2.delete()
>>> Link.objects.count()
```

The link data model

- Django provides the convenient approach to manage the model
    - You don't have to learn another language to access the database.
    - Django transparently handles the conversion between Python objects and table rows.
    - You don't have to worry about any special SQL syntax for different database engines.

The user data model

- Fortunately for us, management of user accounts is so common that Django comes with a user model ready for us to use.

```
>>> from django.contrib.auth.models import User
>>> User.objects.all()
>>> user = User.objects.get(1)
>>>
>>> dir(user)
```

- We can directly use the **User** model without any extra code.

The bookmark data model

- The Bookmark model has a many-to-many relationship between users and links
  - these is the one-to-many relationship between the user and their bookmarks
  - these is the one-to-many relationship between a link and its bookmarks

```
from django.contrib.auth.models import User

class Bookmark(models.Model):
  title = models.CharField(max_length=200)
  user = models.ForeignKey(User)
  link = models.ForeignKey(Link)
```

Templates: creating a template for the main page

- We embed the HTML code of the page into the view's code. This approach has many disadvantages even for a basic view:
  - Good software engineering practices always emphasize the separation between UI and logic
  - Editing HTML embedded within Python requires Python knowledge
  - Handling HTML code within the Python code is a tedious and error-prone task
- Therefore, we'd better separate Django views from HTML code generation

Templates: creating a template for the main page

- This template may contain placeholders for dynamic sections that are generated in the view.
    - the view loads the template and passes dynamic values to it
    - the template replaces the placeholders with these values and generates the page
- First, we need to inform Django of our templates folder. Open `settings.py`

```
import os.path, sys

TEMPLATE_DIRS = (
  os.path.join(sys.path[0], 'templates'),
)
```

Templates: creating a template for the main page

- Next, create a file called main_page.html in the templates folder with the following content:

```
<html>
  <head>
    <title>{{ head_title }}</title>
  </head>
  <body>
    <h1>{{ page_title }}</h1>
    <p>{{ page_body }}</p>
  </body>
</html>
```

Templates: creating a template for the main page

- Edit the `bookmarks/views.py` file and replace its contents with the following code:

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

def main_page(request):
  template = get_template('main_page.html')
  variables = Context({
    'head_title': u'Django Bookmarks',
    'page_title': u'Welcome to Django Bookmarks',
    'page_body': u'Where you can store and share bookmarks!'
  })
  output = template.render(variables)
  return HttpResponse(output)
```

Putting it all together: generating user pages

- Creating the URL
    - The URL of this view will have the form `user/username`, where *username* is the owner of the bookmarks that we want to see.
    - This URL is different from the first URL that we added because it contains a dynamic portion.

```
urlpatterns = patterns('',
   (r'^$', main_page),
   (r'^user/(\w+)/$', user_page),
)
```

Putting it all together: generating user pages

- Open the `bookmarks/views.py` file and enter the following code:

```
from django.http import HttpResponse, Http404
from django.contrib.auth.models import User

def user_page(request, username):
  try:
    user = User.objects.get(username=username)
  except User.DoesNotExist:
    raise Http404(u'Requested user not found.')

  bookmarks = user.bookmark_set.all()
  template = get_template('user_page.html')
  variables = Context({
    'username': username,
    'bookmarks': bookmarks
  })
  output = template.render(variables)
  return HttpResponse(output)
```

Putting it all together: generating user pages

- Create a file called `user_page.html` in the templates directory

```
<html>
  <head>
    <title>Django Bookmarks - User: {{ username }}</title>
  </head>
  <body>
    <h1>Bookmarks for {{ username }}</h1>
    {% if bookmarks %}
    <ul>
      {% for bookmark in bookmarks %}
        <li>
         <a href="{{ bookmark.link.url }}">{{ bookmark.title }}</a>
        </li>
      {% endfor %}
    </ul>
    {% else %}
      <p>No bookmarks found.</p>
    {% endif %}
  </body>
</html>
```

Putting it all together: generating user pages

- Populating the model with data

```
>>> from django.contrib.auth.models import User
>>> from bookmarks.models import *
>>> user = User.objects.get(id=1)
>>> link = Link.objects.get(id=1)

>>> bookmark = Bookmark(
...   title=u'Packt Publishing',
...   user=user,
...   link=link
... )
>>> bookmark.save()

>>> user.bookmark_set.all()
```

User Registration and Management

Topics

- Session authentication
- Improving template structures
- User registration
- Account management

Session authentication

- The Django authentication system is available in the **django.contrib.auth** package that is installed by default.

```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'django_bookmarks.bookmarks',
)
```

Session authentication

- Features of the auth package
  - **Users**: A comprehensive user data model with fields commonly required by web applications
  - **Permissions**: Yes/No flags that indicate whether a user may access a certain feature or not
  - **Groups**: A data model for grouping more than one user together and applying the same set of permissions to them
  - **Messages**: Provides the functionality for displaying information and error messages to the user

Creating the login page

- Those who have worked on programming a session management system in a low-level web framework (such as PHP and its library) will know that this task is not straightforward.
- Fortunately, Django developers have carefully implemented a session management system for us and activating it only requires exposing certain views to the user.

Creating the login page

- First of all, you need to add a new URL entry to the `urls.py` file.

```
urlpatterns = patterns('',
    (r'^$', main_page),
    (r'^user/(\w+)/$',user_page),
    (r'^login/$', 'django.contrib.auth.views.login'),
)
```

- The module `django.contrib.auth.views` contains a number of views related to session management. We have only exposed the login view for now.

Creating the login page

- The login view requires the availability of a template called `registration/login.html` in the templates directory.
- The login view passes passes an object that represents the login form to the template.
  - `form.username` generates HTML code for username
  - `form.password` generates HTML code for password
  - `form.has_errors` is set to true if logging in fails after submitting the form

Creating the login page

- Make a directory named `registration` within the `templates` directory, and create a file called `login.html` in it.

```
<html>
  <head> <title>Django Bookmarks - User Login</title> </head>
  <body>
    <h1>User Login</h1>
    {% if form.errors %}
    <p>Your username and password didn't match.
      Please try again.</p>
    {% endif %}
    <form method="post" action="."> {% csrf_token %}
      <p> <label for="id_username">Username:</label>
        {{ form.username }} </p>
      <p> <label for="id_password">Password:</label>
        {{ form.password }} </p>
      <input type="hidden" name="next" value="/" />
      <input type="submit" value="login" />
    </form>
  </body>
</html>
```

Creating the login page

- It is a good idea to make the main page indicate whether you are logged in or not. Rewrite the `templates/main_page.html` file:

```
<html>
  <head>
    <title>Django Bookmarks</title>
  </head>
  <body>
    <h1>Welcome to Django Bookmarks</h1>
    {% if user.username %}
      <p>Welcome {{ user.username }}!
        Here you can store and share bookmarks!</p>
    {% else %}
      <p>Welcome anonymous user!
        You need to <a href="/login/">login</a>
        before you can store and share bookmarks.</p>
    {% endif %}
  </body>
</html>
```

Creating the login page

- Open `bookmarks/views.py` and change the view as follows:

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

def main_page(request):
  template = get_template('main_page.html')
  variables = Context({'user': request.user})
  output = template.render(variables)
  return HttpResponse(output)
```

Creating the login page

- You may have noticed by now that loading a template, passing variables to it, and rendering the page is a very common task.

- Indeed, it is so common that Django provides a shortcut for it.

```
from django.shortcuts import render_to_response

def main_page(request):
    return render_to_response(
        'main_page.html',
        {'user': request.user}
    )
```

- Using the `render_to_response` method from the `django.shortcuts` package, we have reduced the view to one statement.

Creating the login page

- The user object available at request.user is the same type of User object as we have dealt with before.
  - `is_authenticated()` returns a Boolean value indicating whether the user is logged in or not
  - `get_full_name()` returns the first name and the last name of the user, with a space in between
  - `email_user(subject, message, from_email=None)` sends an email to the user
  - `set_password(raw_password)` sets the user password to the passed value
  - `check_password(raw_password)` returns a Boolean value indicating whether the passed password matches the user password

Creating the login page

- "Why is there a set_password method when one can just as easily set the password attribute of the user object?"

```
>>> from django.contrib.auth.models import User
>>> user = User.objects.get(id=1)
>>> user.password
'sha1$e1f02$bc3c0ef7d3e5e405cbaac0a44cb007c3d34c372c'
```

Enabling logout functionality

- When the user hits the URL `logout/`, we will log the user out and redirect him or her back to the main page.
- To do this, create a new view in the `bookmarks/views.py` file.

```
from django.http
import HttpResponseRedirect
from django.contrib.auth import logout

def logout_page(request):
    logout(request)
    return HttpResponseRedirect('/')
```

Enabling logout functionality

- Now we need to add a URL entry for this view. Open `urls.py` and create an entry as follows:

```
urlpatterns = patterns('',
     (r'^$', main_page),
     (r'^user/(\w+)/$',user_page),
     (r'^login/$', 'django.contrib.auth.views.login'),
     (r'^logout/$', logout_page),
)
```

Improving template structure

- We have created three templates so far. They all share the same general structure, and only differ in the title and main content.
- The Django template system already provides such a feature-template inheritance.
  - create **base** template
  - declare certain **blocks** of the base template that can be modified by **child** template
  - create a child template that **extends** the base template and modifies its blocks

Improving template structure

- Apply this feature to our project by creating a file called `base.html` in the `templates` directory with the following content:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
  <head>
    <title>
      Django Bookmarks | {% block title %}{% endblock %}
    </title>
  </head>
  <body>
    <h1>{% block head %}{% endblock %}</h1>
    {% block content %}{% endblock %}
  </body>
</html>
```

Improving template structure

- The `block` tag is used to define sections that are modifiable by child templates.

- To modify these blocks using a child template, edit the `templates/main_page.html` file and replace its content with the following:

```
{% extends "base.html" %}
{% block title %}Welcome to Django Bookmarks{% endblock %}
{% block head %}Welcome to Django Bookmarks{% endblock %}
{% block content %}
  {% if user.username %}
    <p>Welcome {{ user.username }}!
      Here you can store and share bookmarks!</p>
  {% else %}
    <p>Welcome anonymous user!
      You need to <a href="/login/">login</a>
      before you can store and share bookmarks.</p>
  {% endif %}
{% endblock %}
```

Improving template structure

- Next, let's restructure the templates/user_page.html file to make use of the new base template:

```
{% extends "base.html" %}
{% block title %}{{ username }}{% endblock %}
{% block head %}Bookmarks for {{ username }}{% endblock %}
{% block content %}
  {% if bookmarks %}
    <ul>
      {% for bookmark in bookmarks %}
      <li>
        <a href="{{ bookmark.link.url }}">{{ bookmark.title }}</a>
      </li>
      {% endfor %}
    </ul>
  {% else %}
    <p>No bookmarks found.</p>
  {% endif %}
{% endblock %}
```

Improving template structure

- Finally, let's see how to convert the templates/registration/login.html file:

```
{% extends "base.html" %}
{% block title %}User Login{% endblock %}
{% block head %}User Login{% endblock %}
{% block content %}
  {% if form.errors %}
    <p>Your username and password didn't match.
      Please try again.</p>
  {% endif %}
  <form method="post" action="."> {% csrf_token %}
    <p><label for="id_username">Username:</label>
      {{ form.username }}</p>
    <p><label for="id_password">Password:</label>
      {{ form.password }}</p>
    <input type="submit" value="login" />
    <input type="hidden" name="next" value="/" />
  </form>
{% endblock %}
```

Adding a CSS stylesheet to the project

- Stylesheets and images are static files and Django **does not serve** them.
  - we will use a workaround to make it serve static content.
- Open `urls.py` and update it.

```
import os.path
site_media = os.path.join(
  os.path.dirname(__file__), 'site_media'
)
urlpatterns = patterns('',
   ...
   (r'^site_media/(?P<path>.*)$',
'django.views.static.serve',
     {'document_root': site_media}),
)
```

Linking the stylesheet to the template

- Edit the `templates/base.html` file

```
<head>
  <title>
    Django Bookmarks | {% block title %}{% endblock %}
  </title>
  <link rel="stylesheet" href="/site_media/style.css"
        type="text/css"/>
</head>
```

Adding a navigation menu

- Edit the `templates/base.html` file

```
<body>
  <div id="nav">
    <a href="/">home</a> |
    {% if user.is_authenticated %}
      welcome {{ user.username }}
      (<a href="/logout">logout</a>)
    {% else %}
      <a href="/login/">login</a>
    {% endif %}
  </div>
  <h1>{% block head %}{% endblock %}</h1>
  {% block content %}{% endblock %}
</body>
```

Positioning the navigation menu on the page

- Edit the newly created stylesheet to add the following code:

```
#nav {
  float: right;
}
```

Correcting `user_page` view

- The navigation menu works well on the main page, but not the user page. To overcome this problem, we have two options:
  - Edit our user_page view and pass the user object to the template
  - Use a RequestContext object
- We will modify both `main_page` and `user_page` to use the `RequestContext` objects.

Correcting `user_page` view

- main_page

```
def main_page(request):
    return render_to_response(
        'main_page.html', RequestContext(request))
```

- user_page

```
def user_page(request, username):
  ...
  variables = RequestContext(request, {
    'username': username,
    'bookmarks': bookmarks
  })
  return render_to_response('user_page.html', variables)
```

User registration

- Site visitors need a method to create accounts on the site.
- User registration is a basic feature found in all social networking sites these days.
    - We will create a user registration form and learn about the Django library that handles form generation and processing

Django forms

- Web applications receive input and collect data from users by means of web forms.
- The Django form's library handles three common tasks:
  - HTML form generation
  - Server-side validation of user input
  - HTML form redisplay in case of input errors
- This library works in a way which is similar to the working of Django's data models.

Django forms

- Start by defining a class that represents the form
  - This class must be derived from the `forms.Form` base class
  - Attributes in this class represent form fields
- There are methods for **HTML code generation**, methods to **access the input data**, and methods to **validate the form**.

Designing the user registration form

- Create `forms.py` file in the `bookmarks` application folder.

```
from django import forms

class RegistrationForm(forms.Form):
  username = forms.CharField(label=u'Username', max_length=30)
  email = forms.EmailField(label=u'Email')
  password1 = forms.CharField(
    label=u'Password', widget=forms.PasswordInput()
  )
  password2 = forms.CharField(
    label=u'Password (Again)', widget=forms.PasswordInput()
  )
```

Designing the user registration form

- Several parameters are listed below, which can be passed to the constructor of any field type.
  - `label`: This parameter creates the label of the field when HTML code is generated.
  - `required`: This parameter is set to true by default whether the user enters a value or not. To change it, pass `required=False` to the constructor.
  - `widget`: This parameter lets you control how the field is rendered in HTML. We used it earlier to make the `CharField` of the password a password input field.
  - `help_text`: This parameter displays description of the field when the form is rendered.

Commonly used field types

Field type	Description
`CharField`	Returns a string
`IntegerField`	Returns an integer
`DateField`	Returns a Python `datetime.date` object
`DateTimeField`	Returns a Python `datetime.datetime` object
`EmailField`	Returns a valid email address as a string
`URLField`	Returns a valid URL as a string

Partial list of available form widgets

Widget type	Description
`PasswordInput`	A password text field
`HiddenInput`	A hidden input field
`Textarea`	A text area that enables text entry on multiple lines
`FileInput`	A file upload field

Learn more about the form’s API via console

```
>>> from bookmarks.forms import *
>>> form = RegistrationForm()
>>> print form.as_table()

>>> print form[‘username’]

>>> form = RegistrationForm({
...     ‘username’: ‘test’,
...     ‘email’: ‘test@example.com’,
...     ‘password1’: ‘test’,
...     ‘password2’: ‘test’})

>>> print form.is_bound
>>> print form.is_valid() // True
>>> print form.cleaned_data
>>> print form.errors
```

Customize validations

- The form in its current state detects missing fields and invalid email addresses, but we still need to do the following:
  - Prevent the user from entering an invalid username or a username that's already in use
  - Make sure that the two password fields match

Customize validations

- Append the following method to the RegistrationForm class:

```
def clean_password2(self):
    if 'password1' in self.cleaned_data:
      password1 = self.cleaned_data['password1']
      password2 = self.cleaned_data['password2']
      if password1 == password2:
        return password2
    raise forms.ValidationError('Password do not match.')
```

Customize validations

- Append the following method to the RegistrationForm class:

```
import re
from django.contrib.auth.models import User

  ...

  def clean_username(self):
    username = self.cleaned_data['username']
    if not re.search(r'^\w+$', username):
      raise forms.ValidationError('Username can only contain '
        'alphanumeric characters and the underscore.')
    try:
      User.objects.get(username=username)
    except User.DoesNotExist:
      return username
    raise forms.ValidationError('Username is already taken.')
```

The registration page view

- Open `bookmarks/views.py` and insert the following code:

```
from bookmarks.forms import *
def register_page(request):
  if request.method == 'POST':
    form = RegistrationForm(request.POST)
    if form.is_valid():
      user = User.objects.create_user(
        username=form.cleaned_data['username'],
        password=form.cleaned_data['password1'],
        email=form.cleaned_data['email'])
      return HttpResponseRedirect('/')
  else:
    form = RegistrationForm()
  variables = RequestContext(
    request,
    { 'form': form })
  return render_to_response(
    'registration/register.html',
    variables)
```

The registration page template

- Create a new file called `templates/registration/register.html` and add the following code to it:

```
{% extends "base.html" %}
{% block title %}User Registration{% endblock %}
{% block head %}User Registration{% endblock %}
{% block content %}
<form method="post" action=".">{% csrf_token %}
  {{ form.as_p }}
  <input type="submit" value="register" />
</form>
{% endblock %}
```

- We need to add a URL entry for it

```
(r'^register/$', register_page),
```

- Edit the stylesheet

```
input { display: block; }
```

Adding the registration page to the menu

- Open templates/base.html and modify the navigation menu

```
<body>
  <div id="nav">
    <a href="/">home</a> |
    {% if user.is_authenticated %}
      welcome {{ user.username }}
      (<a href="/logout">logout</a>)
    {% else %}
      <a href="/login/">login</a> |
      <a href="/register/">register</a>
    {% endif %}
  </div>
  <h1>{% block head %}{% endblock %}</h1>
  {% block content %}{% endblock %}
</body>
```

Successful registration page

- It is better if we displayed a success message after the user completes the registration process.
  - It **does not need** to generate dynamic content or process input.
  - Django already provides a view named `direct_to_template` in the `django.views.generic.simple` package for such a task

Successful registration page

- Create a template, `register_success.html`, for the successful registration page at `templates/registration` with the following content:

```
{% extends "base.html" %}
{% block title %}Registration Successful{% endblock %}
{% block head %}
  Registration Completed Successfully
{% endblock %}
{% block content %}
  Thank you for registering. Your information has been
  saved in the database. Now you can either
  <a href="/login/">login</a> or go back to the
  <a href="/">main page</a>.
{% endblock %}
```

Successful registration page

- To directly link this template to a URL, first add this import statement at the beginning of `urls.py`:

```
from django.views.generic.simple import direct_to_template
```

- Next, add the following entry to the URL table:

```
(r'^register/success/$', direct_to_template,
  {'template': 'registration/register_success.html'}),
```

- Finally, modify the `register_page` view in `bookmarks/views.py`

```
return HttpResponseRedirect('/register/success/')
```

Account management

- If we need to add more features about account management such as updating the password or email address. We can do one of the two things:
  - use the views that Django provides for common account management tasks (as we did while creating the login form)
  - design our own form and process its input data (as we did with the registration form)

Account management

- Each views in the `django.contrib.auth` application expects a certain template name to be available and passes some variables to this template.

- All views that are available in the `django.contrib.auth.views` package:

  `logout`: Logs a user out and displays a template when done

  `logout_then_login`: Logs a user out and redirects to the login page

  `password_change`: Enables the user to change the password

  `password_change_done`: Is shown after the password is changed

  `password_reset`: Enables the user to reset the password and receive a new password via email

  `password_reset_done`: Is shown after the password is reset

  `redirect_to_login`: Redirects to the login page

Summary

- The current user (User object) is accessible from the `request.user` attribute of the `HttpRequest` object passed to the view.

- Django provides a shortcut for loading a template, rendering it, and wrapping it in an `HttpResponse` object that is called render_to_response (from the `django.shortcuts` package).

- Don't access the `user.password` directly, use the `user.set_password` method instead. Because it takes care of password hashing for you.

Summary

- A form object can be rendered by using `as_table`, `as_p`, or `as_ul` method on it.
- Binding a from to user input can be done by passing user input as a dictionary to the form's. `is_valid()` is used to validate the input and `errors` attribute will contain found errors in the form.
- Input data and clean data are accessible through `form.data` and `form.cleaned_data` attributes respectively.

Introduction Tags

Topics

- Design a tag data model
- Build a bookmark submission form
- Create pages for listing bookmarks under a certain tag
- Build a tag cloud
- Restrict access to some pages
- Protect against malicious data input by users

The tag data model

- The relationship between tags and bookmarks is a many-to-many relationship, and is represented in Django models using `models.ManyToManyField`.

```
class Tag(models.Model):
    name = models.CharField(max_length=64, unique=True)
    bookmarks = models.ManyToManyField(Bookmark)
    def __unicode__(self):
        return self.name
```

The tag data model

```
$ python manage.py sql bookmarks
```

```
CREATE TABLE "bookmarks_tag_bookmarks" (
    "id" integer NOT NULL PRIMARY KEY,
    "tag_id" integer NOT NULL,
    "bookmark_id" integer NOT NULL REFERENCES "bookmarks_bookmark" ("id"),
    UNIQUE ("tag_id", "bookmark_id")
);

CREATE TABLE "bookmarks_tag" (
    "id" integer NOT NULL PRIMARY KEY,
    "name" varchar(64) NOT NULL UNIQUE
);
```

Creating the bookmark submission form

open the `bookmarks/forms.py` file and add the following class to it:

```
class BookmarkSaveForm(forms.Form):
    url = forms.URLField(
        label=u'URL', widget=forms.TextInput(attrs={'size': 64})
    )
    title = forms.CharField(
        label=u'Title', widget=forms.TextInput(attrs={'size': 64})
    )
    tags = forms.CharField(
        label=u'Tags', required=False,
        widget=forms.TextInput(attrs={'size': 64})
    )
```

Creating the bookmark submission form

open the `bookmarks/views.py` file and add the following class to it:

```
def bookmark_save_page(request):
    if request.method == 'POST':
        form = BookmarkSaveForm(request.POST)
        if form.is_valid():
            # Create or get link
            link, dummy = Link.objects.get_or_create(url=form.cleaned_data['url'])
            # Create or get bookmark
            bookmark, created = Bookmark.objects.get_or_create(user=request.user,link=link)
            # Update bookmark title.
            bookmark.title = form.cleaned_data['title']
            # if the bookmark is being updated, clear old tag list.
            if not created:
                bookmark.tag_set.clear()
            tag_names = form.cleaned_data['tags'].split() # Create new tag list.
            for tag_name in tag_names:
                tag, dummy = Tag.objects.get_or_create(name=tag_name)
                bookmark.tag_set.add(tag)
            bookmark.save() # Save bookmark to database.
            return HttpResponseRedirect('/user/%s/' % request.user.username)
    else:
        form = BookmarkSaveForm()
    variables = RequestContext(request, {'form': form})
    return render_to_response('bookmark_save.html', variables)
```

Creating the bookmark submission form

Create a file called bookmark_save.html in the templates folder and insert the following code into it:

```
{% extends "base.html" %}
{% block title %}Save Bookmark{% endblock %}
{% block head %}Save Bookmark{% endblock %}
{% block content %}
 <form method="post" action=".">
  {% csrf_token %}
  {{ form.as_p }}
  <input type="submit" value="save" />
 </form>
{% endblock %}
```

Creating the bookmark submission form

We are almost there. Open the `urls.py` file and insert the following entry in it:

```
(r'^save/$', bookmark_save_page),
```

Restricting access to logged-in users

Let's add a link to the bookmark submission form in the navigation menu and restructure the menu a little. Open the `templates/base.html` file and update it so that it looks as follows

```
<div id="nav">
  <a href="/">home</a> |
  {% if user.is_authenticated %}
    <a href="/save/">submit</a> |
    <a href="/user/{{ user.username }}/">{{ user.username }}</a> |
    <a href="/logout/">logout</a>
  {% else %}
    <a href="/login/">login</a> |
    <a href="/register/">register</a>
  {% endif %}
</div>
```

How do we make sure that anonymous users cannot submit links?

Restricting access to logged-in users

We can see whether the user is logged in or not:

```
if request.user.is_authenticated():
    # Process form.
else:
    # Redirect to log-in page.
```

However, the task of limiting certain pages to logged-in users is so common that Django provides a shortcut for it,

```
from django.contrib.auth.decorators import login_required

@login_required
def bookmark_save_page(request):
```

Restricting access to logged-in users

- How will login_required know our login URL?
  - By default, it assumes that the page is located at /accounts/login/.
- If we want to change this, we can set the login URL in variable called `LOGIN_URL` that resides in the `settings.py` file.
- So, Add the following code at the end of the `settings.py` file:

```
LOGIN_URL = '/login/'
```

Methods for browsing bookmarks

- Browsing bookmarks lies at the heart of our application.
  - Therefore, it is vital to provide a variety of ways for the user to explore available bookmarks and share them with others.
- Although we intend to provide several ways to browse bookmarks, the technique used to generate bookmark listings will remain the same.
  - First, we build a list of bookmarks using the Django model API.
  - Next, we present the list of bookmarks using a template.

Methods for browsing bookmarks

- We can present each bookmark as a link, with a tag and user information below it. It would be a good idea if we could write one template and reuse it across all the pages.

- The Django template system provides another powerful mechanism that is called the `include` template tag.

    - The concept of the `include` tag is simple. It lets you include the contents of one template file in another.

Methods for browsing bookmarks

- Create a new file called `bookmark_list.html` in the templates directory and enter the following code into it:

```
{% if bookmarks %}
 <ul class="bookmarks">
  {% for bookmark in bookmarks %}
   <li>
    <a href="{{ bookmark.link.url }}" class="title">
      {{ bookmark.title }}</a><br />
    {% if show_tags %}
     Tags:
     {% if bookmark.tag_set.all %}
      <ul class="tags">
       {% for tag in bookmark.tag_set.all %}
        <li>{{ tag.name }}</li>
       {% endfor %}
      </ul>
     {% else %}
      None.
     {% endif %}
     <br />
    {% endif %}
...
```

```
...
{% if show_user %}
     Posted by:
     <a href="/user/{{ bookmark.user.username }}/"
       class="username">{{ bookmark.user.username }}</a>
    {% endif %}
   </li>
  {% endfor %}
 </ul>
{% else %}
 <p>No bookmarks found.</p>
{% endif %}
```

Improving the user page

- Now to make use of this template snippet on the user page, we need to include it from within the `user_page.html` template.
- So open the templates/user_page.html file and modify it to look like the following:

```
{% extends "base.html" %}
{% block title %}{{ username }}{% endblock %}
{% block head %}Bookmarks for {{ username }}{% endblock %}
{% block content %}
  {% include "bookmark_list.html" %}
{% endblock %}
```

Improving the user page

- Before we can see the new template in action, we need to modify the user view and our stylesheet a little.
- Open `bookmarks/views.py` and change the view as follows:

```
from django.shortcuts import get_object_or_404
def user_page(request, username):
    user = get_object_or_404(User, username=username)
    bookmarks = user.bookmark_set.order_by('-id')
    variables = RequestContext(request, {
        'username': username,
        'bookmarks': bookmarks,
        'show_tags': True
    })
    return render_to_response('user_page.html', variables)
```

Improving the user page

- To improve the look of the tag list and avoid nested lists, open the `site_media/style.css` file and insert the following:

```
ul.tags, ul.tags li {
  display: inline;
  margin: 0;
  padding: 0;
}
```

Creating a tag page

- Next, we will create a similar bookmark listing for tags. For this task, we won't write any new code.

- Let's start by adding a URL entry for the tag page. Open the urls.py file and insert the following entry

```
(r'^tag/([^\s]+)/$', tag_page),
```

Creating a tag page

- Next, we will create the tag_page view. Open `bookmarks/views.py` and insert the following code:

```
def tag_page(request, tag_name):
    tag = get_object_or_404(Tag, name=tag_name)
    bookmarks = tag.bookmarks.order_by('-id')
    variables = RequestContext(request, {
        'bookmarks': bookmarks,
        'tag_name': tag_name,
        'show_tags': True,
        'show_user': True
    })
    return render_to_response('tag_page.html', variables)
```

Creating a tag page

- Lastly, we need to create a template for the tag page. Create a file called `templates/tag_page.html` with the following contents:

```
{% extends "base.html" %}
{% block title %}Tag: {{ tag_name }}{% endblock %}
{% block head %}Bookmarks for tag: {{ tag_name }}{% endblock %}
{% block content %}
 {% include "bookmark_list.html" %}
{% endblock %}
```

Creating a tag page

- Before we try out the new tag page, we will link tag names to their respective tag pages.
- To do this, open `bookmark_list.html` and modify the section that generates tag lists as follows:

```
<ul class="tags">
 {% for tag in bookmark.tag_set.all %}
   <li><a href="/tag/{{ tag.name }}/">{{ tag.name }}</a></li>
 {% endfor %}
</ul>
```

Building a tag cloud

- A tag cloud is a visual representation of the tags available in a system and of how often they are used.

- The size of a tag name in the cloud corresponds to the number of items under this tag.

    - The more the items under a certain tag, the larger the font size used to represent the tag.

Building a tag cloud

- Open the `bookmarks/views.py` file and create a new view for the tag cloud page:

```
def tag_cloud_page(request):
    MAX_WEIGHT = 5
    tags = Tag.objects.order_by('name')

    # Calculate tag, min and max counts.
    min_count = max_count = tags[0].bookmarks.count()
    for tag in tags:
        tag.count = tag.bookmarks.count()
        min_count = tag.count if tag.count < min_count else min_count
        max_count = tag.count if max_count < tag.count else max_count

    # Calculate count range. Avoid dividing by zero.
    range = float(max_count-min_count) if max_count != min_count else 1.0

    # Calculate tag weights.
    for tag in tags:
        tag.weight = int(MAX_WEIGHT*(tag.count-min_count)/range)

    variables = RequestContext(request, {'tags': tags})
    return render_to_response('tag_cloud_page.html', variables)
```

Building a tag cloud

- Create a file called `tag_cloud_page.html` in the templates directory with the following content:

```
{% extends "base.html" %}
{% block title %}Tag Cloud{% endblock %}
{% block head %}Tag Cloud{% endblock %}
{% block content %}
  <div id="tag-cloud">
    {% for tag in tags %}
      <a href="/tag/{{ tag.name }}/"
        class="tag-cloud-{{ tag.weight }}">{{ tag.name }}</a>
    {% endfor %}
  </div>
{% endblock %}
```

Building a tag cloud

- Next, we will write CSS code to style the tag cloud. Open the site_media/ style. css file and insert the following code:

```
#tag-cloud {
  text-align: center;
}
#tag-cloud a {
  margin: 0 0.2em;
}
.tag-cloud-0 { font-size: 100%; }
.tag-cloud-1 { font-size: 120%; }
.tag-cloud-2 { font-size: 140%; }
.tag-cloud-3 { font-size: 160%; }
.tag-cloud-4 { font-size: 180%; }
.tag-cloud-5 { font-size: 200%; }
```

Building a tag cloud

- Finally, add an entry to the `urls.py` file. We will map tag_cloud_page to the URL `/tag/` (without a tag name after it):

```
(r'^tag/$', tag_cloud_page),
```

A word on security

- The golden rule in web development is "**Do not trust user input, ever.**"
  - You must always validate and sanitize user input before saving it to the database and presenting it in HTML pages.
- Two common vulnerabilities in web applications:
  - SQL injection
  - Cross-Site Scripting (XSS)

SQL injection

- SQL injection vulnerabilities happen when the developer uses input to construct SQL queries without escaping special characters in it.
- As we are using the Django model API to store and retrieve data, we are safe from these types of attacks.
  - The model API automatically escapes input before using it to build queries.
  - So **we do not need to do anything special** to protect our application from SQL injections.

Cross-Site Scripting (XSS)

- A malicious user supplies JavaScript code within input. When this input is rendered into an HTML page, the JavaScript code is executed to take control of the page and steal information such as cookies.

- Again, Django automatically does this for us by escaping template variables before printing them to a page.

- For example, a user may submit the following string as a bookmark title:

    - `<script>alert("Test.");</script>`

Template filters

- There are times when you want to disable auto-escaping.
    - a piece of HTML-formatted text that has already been sanitized
- Django provides a feature called **template filters** to process variables before printing them in a template.
    - `safe` filter : disables auto-escaping and prints the template variable as is
    - `urlencode` filter : escapes a string for use in a URL

Template filters

- Let's use the `urlencode` filter in the `templates/bookmark_list.html` file.

```
<ul class="tags">
  {% for tag in bookmark.tag_set.all %}
    <li><a href="/tag/{{ tag.namelencodeurl }}/">{{ tag.name }}</a></li>
  {% endfor %}
</ul>
```

- You will need to do a similar change to `templates/tag_cloud_page.html` file:

```
<div id="tag-cloud">
  {% for tag in tags %}
    <a href="/tag/{{ tag.namelencodeurl }}/"
      class="tag-cloud-{{ tag.weight }}">{{ tag.name }}</a>
  {% endfor %}
</div>
```