The Web framework for perfectionists with deadlines.

Basic CRUD in Django

Low-Level Creation, Updating, and Deletion of Data

by Mike Cantelon
http://mikecantelon.com

Setting Up a Project

1. Change to the directory in which your project should be created
2. Issue the command `django-admin.py startproject linkdump`
3. This command will have created a directory, with the specified project name (in this case "linkdump"), containing base files for the project
4. Change to this directory
5. Issue the command `python manage.py runserver` to start the development server (http://127.0.0.1:8000)
6. If your project is to use a new, rather than existing, database then create it (unless using sqlite, which be automatically created)
7. Edit settings.py to configure access to your database
8. Issue the command `python manage.py syncdb` to build database tables needed by the administration interface (useful later). Follow the prompts.

Creating an Application

1. As a simple demonstration, we'll create an application to submit links to

2. To create the application's directory and base files, we issue the command `python manage.py startapp linktracker` ("linktracker" being the name of our application)

3. Next, we edit linktracker/models.py to define our model

```
class Link (models.Model):
    link_description = models.CharField(maxlength=200)
    link_url = models.CharField(maxlength=200)
```

4. To add our app to the project, we edit the project's settings.py to add our app (linkdump.linktracker) to the INSTALLED_APPS list in settings.py

5. To update the database, automatically creating the necessary tables, we issue the command `python manage.py syncdb`

Activating Administration

1. Add "django.contrib.admin" to the INSTALLED_APPS list in settings.py
2. Issue the command `python manage.py syncdb`
3. Edit urls.py to enable admin (the next slide will talk more about this file's functionality)
4. Issue the command `python manage.py runserver`
5. You can verify that admin is running by browsing http://127.0.0.1:8000/admin
6. Edit linktracker/models.py to turn on admin for links:

```
class Link(models.Model):
    # ...
    class Admin:
        pass
```

7. Edit linktrackers/models.py to enable default (naming) column for links

```
def __str__(self):
    return self.link_description
```

Designing URLs

- Now that we can enter data, adding a way for user to see this data would be useful
- To present data we need to write a "view" which is mapped to a URL pattern
- Editing urls.py allows URL patterns to be defined using regular expressions
- Add the below link to your urls.py file:
  `(r'^links/$', 'linkdump.linktracker.views.list')`
- The above line will, if someone browses /links/ on our site, try to display the results of the list view of the links application of the linkdump project... we have not yet created a list view, however, so we just get an error!

Enabling Templating

1. Views usually incorporate HTML templates so presentation is separated from logic

2. To enable templating, create a directory called “template” in which to store your templates then edit settings.py and add “template” to TEMPLATE_DIRS

3. Edit linktracker/views.py to create a list view for links that uses templating

```
from linkdump.linktracker.models import Link
from django.template import Context, loader
from django.shortcuts import render_to_response

def list(request):
    link_list = Link.objects.all()
    return render_to_response(
        'links/list.html',
        {'link_list': link_list}
    )
```

Creating A Template

1. We'll create a template to crudely display link items
2. Edit template/links/list.html:

```
{% if link_list %}
    <ul>
        {% for link in link_list %}
            <li><a href='{{ link.link_url }}'>
                {{link.link_description}}</a></li>
        {% endfor %}
    </ul>
{% else %}
    <p>No links found.</p>
{% endif %}
```

3. The /links/ URL pattern now actually does something!

Pagination (1 of 3)

- Use the admin interface to add a bunch of links then reload the /links/ URL
- An unlimited list is a problem so we need to add pagination

- We will have to add pagination support in three places: our URL patterns (urls.py), our list view (linktracker/views.py), and our template (template/links/list.html

- To add a URL pattern supporting pagination, add the below line to urls.py:

```
(r'^links/(?P<page>\d+)', 'linkdump.linktracker.views.list')
```

Pagination (2 of 3)

- To add pagination support to linktracker/views.py, first add the following line near the start:

```
from django.core.paginator import ObjectPaginator, InvalidPage
```

- Next, change the list function in linktracker/views.py to the code below:

```
def list(request, page = 0):
    page = int(page)
    link_list = ObjectPaginator(Link.objects.all(), 5)
    has_previous = link_list.has_previous_page(page)
    has_next = link_list.has_next_page(page)
    return render_to_response('links/list.html',
        {'link_list': link_list.get_page(page),
         'has_previous': has_previous,
         'previous_page': page - 1,
         'has_next': has_next,
         'next_page': page + 1}
    )
```

Pagination (3 of 3)

- Last, but not least, you'll want to change your template/links/list.html HTML template to support pagination... add the following lines before the line containing "else":

```
{% if has_previous %}
<a href='/links/{{ previous_page }}'>Previous</a>
    {% if has_next %} | {% endif %}
{% endif %}

{% if has_next %}
<a href='/links/{{ next_page }}'>Next</a>
{% endif %}
```

- Pagination is now a wonder to behold!

Prep for CRUD (1 of 2)

- Before getting into the nitty gritty of CRUD, we need a way to give users feedback for when they create, update, or delete data
- One simple way to do this is by passing a message via a GET parameter
- To allow this functionality we need to modify our list view to receive messages and modify the list.html template to display messages
- Change the first line of the list view in views.py, adding a parameter for messages:

```
def list(request, page = 0, message = ""):
```

- Also in views.py, add a line in the list view so the passed message is included in the list of parameters relayed to the list.html template:

```
'message': message,
```

Prep for CRUD (2 of 2)

- Now that we're preparing to add CRUD functionality, we need to update our list HTML template as shown below:

```
{% if message %}
 <b>{{ message }}</b>
 <p>
 {% endif %}
 {% if link_list %}
    <table>
    {% for link in link_list %}
        <tr bgcolor='{% cycle FFFFFF,EEEEEE as rowcolor %}'>
            <td><a href='{{ link.link_url }}'>{{ link.link_description }}</a></td>
            <td><a href='/links/edit/{{ link.id }}'>Edit</a></td>
            <td><a href='/links/delete/{{ link.id }}'>Delete</a></td>
        </tr>
    {% endfor %}
    </table>
    <p>
    {% if has_previous %}
        <a href='/links/{{ previous_page }}'>Previous</a>
        {% if has_next %}|{% endif %}
    {% endif %}

    {% if has_next %}
        <a href='/links/{{ next_page }}'>Next</a>
    {% endif %}
{% else %}
    <p>No links found.</p>
{% endif %}
<p>

<a href='/links/new'>Add Link</a>
```

Creating Data (1 of 2)

- Allowing users to submit data requires first displaying an HTML form then adding the submitted data
- To start, add a URL for form display to urls.py:

```
(r'^links/new', 'linkdump.linktracker.views.new')
```

- To deal with the submitted data, we add another URL to urls.py:

```
(r'^links/add/', 'linkdump.linktracker.views.add')
```

- Then, to display an HTML add form in linktracker/views.py, add the following block of code:

```
def new(request):
    return render_to_reponse(
        'links/form.html',
        {'action': 'add',
        'button': 'Add'}
    )
```

Creating Data (2 of 2)

- Next, we add the form HTML template (which, as written below, can be used for adding *or* editing, as we'll see later):

```
<form action='/links/{{ action }}/' method='post'>
    Description:
    <input name=link_description value="{{ link.link_description|escape}}"><br />
    URL:
    <input name=link_url value='{{ link.link_url|escape }}'><br />
    <input type=submit value='{{ button }}'>
</form>
```

- Finally, we add logic to linktracker/views.py to add the submitted data then return to the list view, passing a message:

```
def add(request):
    link_description = request.POST["link_description"]
    link_url = request.POST["link_url"]
    link = Link(
        link_description = link_description,
        link_url = link_url
    )
    link.save()
    return list(request, message="Link added!")
```

Updating Data (1 of 2)

- Allowing users to upate data requires first displaying an HTML form showing current data then, upon submission, doing the actual update
- To start, add a URL for form display to urls.py:

```
(r'^links/edit/(?P<id>\d+)', 'linkdump.linktracker.views.edit')
```

- Then, to display an HTML edit form in linktracker/views.py, add the following block of code:

```
def edit(request, id):
    link = Link.objects.get(id=id)
    return render_to_reponse(
        'links/form.html',
        {'action': 'update/' + id,
        'button': 'Update'}
    )
```

- **NOTE:** As the HTML form we added earlier can be used to edit, we're re-using it in the above code

Updating Data (2 of 2)

- To deal with submitted updates, we add another URL to urls.py:

```
(r'^links/update/(?P<id>\d+)', 'linkdump.linktracker.views.update')
```

- Finally, we add logic to linktracker/views.py to add the submitted data then return to the list view, passing a message:

```
def update(request, id):
    link = Link.objects.get(id=id)
    link.link_description = request.POST["link_description"]
    link.link_url = request.POST["link_url"]
    link.save()
    return list(request, message="Link updated!")
```

Deleting Data

- Allowing users to delete data requires first displaying an HTML form showing current data then, upon submission, doing the actual update

- To support deletion, add a new URL pattern to urls.py:

```
(r'^links/delete/(?P<id>\d+)', 'diglic.links.views.delete')
```

- Then, to display an HTML edit form in linktracker/views.py, add the following block of code:

```
def delete(request, id):
    Link.objects.get(id=id).delete()
    return list(request, message="Link deleted!")
```