Django, Part 1

Introduction

Prerequisites

In this tutorial, it is assume that you have Django v1.7 installed with Python3.

You'll also need the two CSS files provided with this tutorial. You can use a standard bootstrap CSS instead if you don't have them (http://getbootstrap.com/).

Goal of the tutorial

In this tutorial, we will:

- Start a project from scratch
- Generate a model, synchronize it with the DB provided by Django and interact with it through the default admin interface
- Generate another model, link it to the first and design a full CRUD (Create Read Update Delete) to interact with it as a user of the site, using Django's data validation and security tools
- Use templates with template inheritance and integrate a custom bootstrap CSS
- Set up the default user authentication system and an authentication wall to some part of the site

Description of the project

The project is an abstraction of the common kinds of site, it uses two types of object: storages that can only be created in the admin interface and boxes that can be manipulated by the users via a CRUD and that are linked to a storage in a many-to-one relationship.

The user stories are:

- As an admin, I can create users, boxes and storage from the admin interface
- As a non-authenticated user I can see the list of all boxes at the index view (/box) and log in
- As a log in user I can:
    - See all boxes at /box
    - Create a box choosing a tag and a storage from the list.
    - Read (= view) a box details at /box/{id} (link in the box # in the index view)
    - Update it from the read page, I can only change the storage
    - Delete it from the read page

Starting from scratch

Creating a new project

From the command line, go to the directory where you'd like to have your code. It doesn't have to be inside /var/www. Once there the following command will generate the core file of your Django project

```
$ django-admin startproject webtech_django1
```

This should create a folder named webtech_django1. Inside this directory you'll find:

- A script named manage.py offering a command-line interface to various Django's functionalities
- Another directory with the same name containing urls.py where you will configure the routing of your site and settings.py where you can configure numerous Django's core setting.

You can already test your project by running

```
$ python manage.py runserver
```

You should see a webpage at http://localhost:8000. However when you ran this command, Django informed you that the database isn't set up.

Setting up the database

In this tutorial we will use the default database provided by Django. To set it up we will use manage.py. Django uses migrations to ensure that the model in your code match the model in the database, migrations contains the queries requires to adapt the database to the model of the code.

During the project creation the migrations required to set up the database where automatically generated.

Run them using

```
$ python manage.py migrate
```

Creating the first app

Now that the project is set up, we can start with our first app. To quote from the official documentation:

What's the difference between a project and an app? An app is a Web application that does something – e.g., a Weblog system, a database of public records or a simple poll app. A project is a collection of configuration and apps for a particular Web site. A project can contain multiple apps. An app can be in multiple projects.

We will start with an app called storage. To create it we'll use manage.py:

```
$ python manage.py startapp storage
```

This should have created a directory named storage. This directory contains the code of this app.

But the project don't know yet that this app should be used, we'll have to edit settings.py.

In settings.py, add `'storage',` to the INSTALLED_APP list.

The storage model

Creating the model

Go inside the storage directory and open models.py. In every app, this is where you will define the model.

Let's write our model for the Storage class. The class has only one field called "name" that is a string (not too long). When converted to a string, a storage should give out its name. To be a model recognized by Django, the class needs to extend models.Model.

This gives us the following code:

```
class Storage(models.Model): #extend django's Model class

    #field in the database
    name = models.CharField(max_length = 100)

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

You could define as many classes in this models.py as required for your app but we'll only need this one.

Migration

We now have to synchronize this model with the one in the database, therefor we create a new migration. We use manage.py to do it by making a migration for the storage app:

```
$ python manage.py makemigrations storage
```

You should see that a migration was created. Now we can run it using:

```
$ python manage.py migrate
```

This will create a table storage_storage (appName_className) inside the database. The table contains an id column as primary key and the name column defined in models.py. The id will be managed by Django, you can however access it in your script using the field name 'id'.

The admin site

Setting up the admin superuser

Django can set up an admin site for your site. In this admin site you will be able to manage the user permissions and the database. To use the admin site you have to set up a superuser using manage.py:

```
$ python manage.py createsuperuser
```

Give a username, an email (optional) and a password.

Using the admin site

To access the admin site start the server using the runserver option for manage.py, then go to http://localhost:8000/admin and log in using the admin account you created.

There you see that you can manage users and groups but storage isn't there yet. You need to hook it to the admin site first.

To do that open the admin.py file in the storage directory and add the following lines:

```
from django.contrib import admin
from storage.models import Storage

admin.site.register(Storage) #Hook the models to the admin site
```

Refresh the admin site and you are now able to manage your storage from the admin site. You can add a few, edit them and delete them. Add at least 2-3 storages for the next part of the tutorial

The Box app, basic view

Creating a new app

We will now create the box application of our project. Since the storage application is pretty much empty you wouldn't normally make it a full-fledged application and instead incorporate it into the box application that will use it. But for the purpose of this tutorial let's assume that you will expand the storage app later and therefor require this to be outside the box app.

Create the new app:

```
$ python manage.py startapp box
```

Add it to the INSTALLED_APP in settings.py by adding `'box',` to the list

A box should have a tag and be linked to a storage. This box-to-storage relationship is a many-to-one relationship so it's easier to have the box model have the foreign key. For this kind of relationship Django will handle the abstraction, all you have to do is define a field as a Storage object and the database abstraction layer will convert it into an id foreign key. Therefor don't forget to import your Storage class from storage.models:

```
from django.db import models
from storage.models import Storage

class Box(models.Model): #extend django's Model class

    #field in the database
    tag = models.CharField(max_length = 100)
    storage = models.ForeignKey(Storage) #f.key for the many-to-one relationship

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

Now you can create a migration and migrate the new model into the database

```
$ python manage.py makemigrations box

$ python manage.py migrate
```

Finally hook the box's model to the admin site by editing the admin.py from the box directory

```
from django.contrib import admin
from box.models import Box

admin.site.register(Box) #Hook the models to the admin site
```

You can now use the admin site to create a few boxes. You should have a droplist to choose which one of your Storage objects is linked to the box you're creating.

A view for the app

We will now create views for our box model. Views allow users to interact with the model without the admin site and can go from basic HTML response (404 for example) to a full webpage.

We will create two views for our box. The first one will be our index and will show all the available boxes. The second one will show details about one box in particular.

First we need to implement the views in box/views.py. We will use basic view without templates first. Therefor we need to import from Django the HttpResponse object. We will also need to interact with the database using the models.Box we created.

To create a view, we define a function taking request as an argument and other eventual argument passed into the URL. We then interact with the database, produce a response string and return an HttpResponse (by default it will be an HTTP 200 with the given string as HTML response).

To fetch all boxes, we use the database API command Box.objects.all(). To get one particular we could use Box.objects.get(id=box_id) but this would raise an error if the id doesn't exist, the correct behavior would be to throw a HTTP 404 error and Django has a shortcut for it, get_object_or_404, that we need to import

```
from django.http import HttpResponse
from box.models import Box
from django.shortcuts import get_object_or_404

#List all the boxes
def index(request):
    box_list = Box.objects.all() #fetch all the boxes
    response = "Available boxes: <ul>"
    for b in box_list:
        response += "<li> #" + str(b.id) + " | Tag: " + b.tag + "</li>"
    response += "</ul>"
    return HttpResponse(response)

#Give details about one box
def read(request, box_id):
    b = get_object_or_404(Box, id=box_id) #fetch one box, 404 if box_id invalid
    return HttpResponse("Box #" + str(b.id) + "<br>Tag: " + b.tag + "<br>Storage: " + str(b.storage))
```

Now we need to link this function to an URL using Django's routing.

We could use the urls.py in the main configuration directory (the inner webtech_django1) but a good practice is to separate each routing to its own app. Therefor we will create an urls.py file inside the box directory and set it up correctly.

A simple URLconf is defined by a pattern (a regex), followed by a view and should have a name so that it can be dynamically generated in other views or templates using it name. This will allow use to change the URL of a route by editing only one file.

Here the path should be /box to view the index and /box/{box_id} to view the details about the box with id = box_id

```
from django.conf.urls import patterns, url

from box import views

urlpatterns = patterns('',
    url(r'^$', views.index, name='index'),
    url(r'^(?P<box_id>\d+)/$', views.read, name='read'),
)
```

Now we just need to add this file to the global URLconf in webtech_django1/urls.py, we will also use route namespacing in case we'll later have other apps using the same names for their routes. Add the following to the pattern list:

```
url(r'^box/', include('box.urls', namespace='box')),
```

You can now test that http://localhost:8000/box lists all your boxes and that using an available id, you can see a box detail at /box/id

Templates

A basic template

We now have views but they aren't pretty and wouldn't be easy to extend. Django offers a solution with templates. To use it we will use the default template loader which require to have the template for the box app located inside box/templates. You can then use subdirectories if you want to classify your templates.

The syntax used by the template engine is fairly simple and provide some basic looping and branching:

- {{variable}} to print a variable
- {% if bool %} … {% else %} … {% endif %} for branching
- {% for a in a_set %} … {% endfor %} to iterate over a set
- {% function %} to execute a function
- {% block BlockName %} … {% endblock %} to define blocks (see template inheritance)

You can find more in the official documentation.

Here is a basic template for our index page.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Webtech Django 1</title>
  </head>
  <body>
    {% if box_list %}
      <table>
        <tr>
          <th>#</th>
          <th>Tag</th>
          <th>Storage</th>
        </tr>
        {% for box in box_list %}
        <tr>
          <td><a href={% url 'box:read' box.id%}>{{ box.id }}</a></td>
          <td>{{box.tag}}</td>
          <td>{{box.storage}}</td>
        </tr>
        {% endfor %}
      </table>
    {% else %}
      <p>No boxes are available.</p>
    {% endif %}
  </body>
</html>
```

All information are displayed in a tab and we use the {% url 'box:read' box.id %} function to generate the URL for the details of a box, it will use the routing that we defined in urls.py. If no box are available we display a simple text.

To use this template instead of the HttpResponse, we have to edit the index function in box/views.py. We'll use the shortcut render to render the template, the render shortcut takes 3 arguments: the request object, the template path and the template argument.

Here we have to pass the list of all boxes as box_list to the template engine:

```
from django.shortcuts import get_object_or_404, render

#List all the boxes
def index(request):
    box_list = Box.objects.all() #fetch all the boxes
    return render(request, 'index.html', {'box_list': box_list})
```

Notice that the code for the view is now much shorter as all the data formatting for the display is handle inside the template.

Template inheritance

We could simply do the same for the read view. But that would create duplicate code. Django's solution is template inheritance. We will therefor design a base template that define the general aspect of our site and then make the specific view inherit from it.

Because this base template will be for our whole project and not just the box app, we want to have it somewhere outside of the app directory. To do that we will use the filesystem template loader and configure it to look at a directory /templates at the root of the project with the settings.py by adding:

```
TEMPLATE_DIRS = (
    os.path.join(BASE_DIR, 'templates'),
)
```

We can now create a directory named templates at the root folder of the project and all templates inside will be loaded by the template engine.

To use template inheritance, the template engine use blocks. A block is a part of a template that can be inherited as it or overloaded by the child template using a block with the same name.

Create a template name base.html in the new templates folder:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        {% block head_title %}
            <title>Django 1</title>
        {% endblock %}
    </head>
    <body>
        {% block content %}
        {% endblock %}
    </body>
</html>
```

Then edit the index.html template to extend base.html

```
{% extends 'base.html' %}

{% block content %}
    {% if box_list %}
        [...]
    {% endif %}
{% endblock %}
```

Static files (CSS)

Now that we have a base template, we can use it to include some global CSS. We will use the bootstrap CSS and its customization main.css provided with the tutorial but you can also download bootstrap from its website.

If we'd dynamically linked the CSS to base.html, the path would be wrong for all view inheriting it located in another folder. We could use hard coded full path, or Django's static files system, the latter being the best practice.

Static files are located inside a directory located in one of the place defined by the STATICFILES_DIRS variable in settings.py and will be seen as if their path started with the path stored inside STATIC_URL.

By adding the following STATICFILES_DIRS to settings.py:

```
STATICFILES_DIRS = (
   os.path.join(BASE_DIR, "static"),
)
```

We can create a directory named static in the root directory of your project and put the CSS files there.

Now edit the base.html template to load the static files with the following in the header:

```
<head>
   [...]
   {% load staticfiles %}
   <link href="{% static 'bootstrap.min.css' %}" rel="stylesheet">
   <link href="{% static 'main.css' %}" rel="stylesheet">
</head>
```

To improve our basic layout, let's use the new bootstrap features and add a jumbotron and a footer:

```
<body>
   <div class="container">
      <div class="jumbotron">
         {% block jumbotron %}
            <h1>Default Jumbotron</h1>
         {% endblock %}
      </div>
         <div class="row maincontent">
         <div class="col-md-10 col-md-offset-1">
            {% block content %}
            {% endblock %}
         </div>
         </div>
      <div class="footer">
         {% block footer %}
            <p>Default footer</p>
         {% endblock %}
      </div>
   </div>
</body>
```

Now we can edit index.html to include our new feature:

```
{% extends "base.html" %}

{% block jumbotron %}
   <h1>All boxes</h1>
{% endblock %}

{% block content %}
   {% if box_list %}
      <table class="table table-striped">
         <tr>
            <th>#</th>
            <th>Tag</th>
```

```
            <th>Storage</th>
          </tr>
        {% for box in box_list %}
          <tr>
            <td><a href={% url 'box:read' box.id%}>{{ box.id }}</a></td>
            <td>{{box.tag}}</td>
            <td>{{box.storage}}</td>
          </tr>
        {% endfor %}
        </table>
    {% else %}
        <p>No boxes are available.</p>
    {% endif %}
{% endblock %}
```

You can now create a read.html template for the read view of a box's detail using the layout of base.html and overriding the jumbotron and the content. Don't forget to change the return of the read view in views.py to a render.

CRUD

ModelForm and Create

Now that we have templates and models ready, we can write a CRUD for our box app. However these usually require a lot of data validations and forms. Django provides the ModelForm object to handle all these step itself. A ModelForm defines what fields of an object have to be in a form in a template and once synchronized with a POST request it can validate the data and update the database.

The create view will first bind the ModelForm form to the request. If the form isn't valid (for example because the user did a GET request), we render the template create.html (extends base.html) with the form to create a box. Else we save the box and redirect the user to the index, to do that we can use the shortcut redirect that we have to import.

```
from django.shortcuts import get_object_or_404, render, redirect
from django.forms import ModelForm

#ModelForm to create a Box
class BoxCreateForm(ModelForm):
    class Meta:
        model = Box
        fields = ['tag', 'storage']

#Create a Box
def create(request):
    form = BoxCreateForm(request.POST or None)
    if form.is_valid():
        form.save()
        return redirect('box:index')
    return render(request, 'create.html', {'form':form})
```

The ModelForm provides all the input field of the form for the template engine:

```
{% block content %}
<p>
    <form method="post">
        {% csrf_token %}
        {{ form.as_p }}
    <input role="button" class="btn btn-lg btn-primary btn-block" type="submit" value="Create" />
    </form>
</p>
{% endblock %}
```

The {{form.as_p}} will render the form. Notice the {% csrf_token %} that you should include to all form. It provides protection from CSRF attacks using the CSRF middleware activated by default in settings.py.

Finally don't forget to add a route to the view in box/urls.py

```
url(r'^create/$', views.create, name='create'),
```

You can add a link to create from the index by adding the following to index.html:

```
<p><a role="button" class="btn btn-lg btn-primary btn-block" href="{% url 'box:create' %}">Create a box</a></p>
```

ModelForm and Update

We could try to use the same ModelForm for the Update part of the CRUD but our project's specifications say that the tag of a box shouldn't be modifiable. Therefor we'll use another ModelForm:

```
#ModelForm to update a box, only the storage can change
class BoxUpdateForm(ModelForm):
    class Meta:
        model = Box
        fields = ['storage']

#Update a box
def update(request, box_id):
    box= get_object_or_404(Box, id=box_id)
    form = BoxUpdateForm(request.POST or None, instance=box, initial={'storage': box.storage})
    if form.is_valid():
        form.save()
        return redirect('box:read', box_id)
    return render(request, 'update.html', {'form':form})
```

The template update.html will be almost the same as create.html. Don't forget the {% csrf_token %}. You then have to create a route to update, as you can see the function take a box_id argument like read so the pattern in urls.py will have a regex:

```
url(r'^update/(?P<box_id>\d+)/$', views.update, name='update'),
```

You can then add a link to update from read by adding the following to read.html

```
<p><a role="button" class="btn btn-lg btn-success btn-block" href="{% url 'box:update' box.id %}">Update</a></p>
```

Delete

To delete a box, no need for a specific view. However it is better to use a form to protect your users from CSRF attacks.

You can add a route called 'delete' to urls.py and a hidden form (just the submit button visible) in read.html:

```
<form action="{% url 'box:delete' %}" method="post">
    {% csrf_token %} <!-- no CSRF attack is possible to delete a box using an external form -->
    <input hidden name="box_id" value={{box.id}} />
    <input role="button" class="btn btn-lg btn-danger btn-block" type="submit" value="Delete" />
</form>
```

Then add a delete view that will only work using a POST Request in views.py:

```
def delete(request):
    b = get_object_or_404(Box, id=request.POST['box_id']).delete()
    return redirect('box:index')
```

User authentication

Login and logout page

In the MIDDLEWARE_CLASSES, you might have seen two middleware named SessionAuthentication and Authentication. This is the user authentication feature provided by Django and we will use it to create an authentication wall.

First create some users using the admin site, you don't need to do more than setting up a username and a password. Note that the admin you used to log in into the admin site is also a valid user.

Now to use the provided feature, we need at least to set up routes to a login page, we will also set up a logout page.

Start by adding routes in the main urls.py:

```
url('^login/','django.contrib.auth.views.login', {'template_name': 'login.html'}, name='login'),
url('^logout/','django.contrib.auth.views.logout', {'template_name': 'logout.html'}, name='logout'),
```

Then we need a login.html and a lougout.html template. As they belong to no app, we'll add them to our global template folder.

Both will have a form object containing all the required field. You'll need however to make a hidden field called 'next' to set the redirection after the login/logout

Here is for example the login template content block (make it extend base.html)

```
{% block content %}
<p>
   <form method="post">
      {% csrf_token %}
      {{ form.as_p }}
   <input role="button" class="btn btn-lg btn-success btn-block" type="submit" value="Login" />
   <input type="hidden" name="next" value="{% url 'box:index' %}" />
   </form>
</p>
{% endblock %}
```

Finally you have to ensure that the setting's login path is the same as the one defined in urls.py. You can hardcode it but it is better to use reverse_lazy in settings.py to match the one in urls.py:

```
#Login url
from django.core.urlresolvers import reverse_lazy
LOGIN_URL = reverse_lazy('login')
```

You should now be able to login at http://localhost:8000/login

You can also modify the footer of base.html so that it link to the login page if you aren't connected and to the logout page if you are by using the user.is_authenticated test in the template.

```
<div class="footer">
   {% block footer %}
      {% if user.is_authenticated %}
         <p>Hello {{user.username}} | <a href="{% url 'logout' %}">Logout</a></p>
      {% else %}
         <a href="{% url 'login' %}">Login</a>
      {% endif %}
   {% endblock %}
</div>
```

You can also access many information about the user using the user object in a template, this object is always transmitted to the template engine.

Authentication wall

Only authenticated user should be able to access the box CRUD. The request contains the user information at request.user. You could check every time if request.user.is_authenticated() and redirect to the login page if not but there is an easier way using a Decorator pattern that will handle it for you.

Use the decorator @login_required by importing it and adding it in front of every function requiring a login in views.py:

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

#Create a box
@login_required
def create(request):
[...]
```

You can now test that if you aren't connected, you will be redirected to the login page when attempting to use the CRUD but not when seeing the index if you didn't set up a @login_required before the index function

Going further

This is the end of this tutorial. You now know how to quickly build the layout and key features of a website using Django. The basic website we build could easily be extended into a complex fully-fledged one with only the simple to use features that this tutorial exposed.

However this isn't the end for you. If you liked this first experience with Django, you can join its thriving community and learn the full extent of its possibility. Django has a lot more to offer and is very well documented. A good start would be how to modify and adapt to your specific needs the default feature we exposed.