Django tutorial

October 22, 2010

1 Introduction

In this tutorial we will make a sample Django application that uses all the basic Django components: Models, Views, Templates, Urls, Forms and the Admin site. This tutorial will make use of the eclipse IDE, with the pydev extension installed. For a complete installation of python, django, sqlite and eclipse I gladly refer to google, covering this for each possible platform is out of scope. A basic installation with everything available is provided on the computers.

2 Setup environment

To start we will setup a complete django environment in eclipse.

2.1 Create pydev project

1. Open Eclipse
2. Select `File > New > Project...`
3. Select `Pydev > Pydev Django Project` and click `next`
4. Call the project `djangolesson`, assistance will be provided to fill out the python information. Make sure creation of 'src' folder is unchecked, See figure 1 for an example. Click `next` again.
5. Fill out the last form, make sure `django 1.2` and `database sqlite3` are chosen, for the other fields default values should suffice. Finally, click finish.

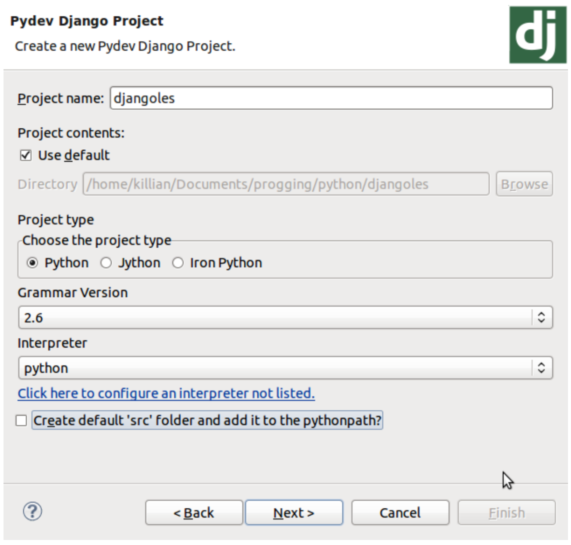
2.2 Run the project

1. Right click on the project (not subfolders).
2. `Run As > Pydev:Django`
3. Open a browser and go to `http://localhost:8000/`

Congratulations, you've started your first django site. Notice a console is opened inside eclipse that shows some basic information of the development server. In this console there are buttons provided to stop or restart the server.

2.3 Django management commands and shell

The eclipse interface did some work for us behind the screens, it invoked many django manage commands which you would otherwise have to execute manually. While this suffices for this simple example, deeper knowledge of django manage commands is essential for larger projects. The official django tutorial on `www.djangoproject.com` does cover these commands and also shows how to use the django shell. Note however that all these commands can also be called through the django submenu of the eclipse project.


Pydev Django Project

Create a new Pydev Django Project.

Project name: djangoles

Project contents:

☑ Use default

Directory /home/killian/Documents/progging/python/djangoles

Project type

Choose the project type

● Python ○ Jython ○ Iron Python

Grammar Version: 2.6

Interpreter: python

Click here to configure an interpreter not listed.

☐ Create default 'src' folder and add it to the pythonpath?

Figure 1: Example pydev project configuration

3 A first view

Here we will create a very simple view that will display only the classic `"Hello world!"` message.

3.1 Create new application

First we start by creating a new application inside the django project. Note that after creating the application we need to add it to the project settings, this is necessary to use it in the database, which will be covered further in this tutorial.

1. Right click on the project and select `Django > Create Application`,
2. Call the application `members`. Click `OK`.
3. Open `settings.py`
4. Add `djangolesson.members` to the `INSTALLED_APPS` list so it looks like this:

```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.messages',
    'djangolesson.members',
)
```

You can see eclipse automatically created some files for us. Only the `test.py` file will not be covered in this tutorial, in it you can write tests that verify the functionality of your django application. While testing is an essential part of modern high-quality applications, it is a topic on itself that is not covered in this tutorial.

3.2 Create view method

Open the views.py file in the `members` application and modify it to look like this:

```
from django.http import HttpResponse

def myFirstView(request):
    return HttpResponse("Hello world!")
```

As you can see, a view is a simple python function that takes at least one variable, the request context, which contains some extra data like the URL, form input, logged in user, ... The view should always return an HTTP compatible message, the easiest way is to use `HttpResponse` which just sends some text back that will be displayed in the browser.

3.3 configure urls

Before we can actually see the view we need to configure urls, we will do this in two steps, first open `urls.py` in the main project and change `urlpatterns` to this:

```
urlpatterns = patterns('',
    (r'^members/', include("members.urls")),
)
```

This makes sure that each url starting with `members/` will further be processed in the `urls.py` file of the members application. Now create this file by right clicking on the `members` application and selecting `New > Pydev Module`. Name it urls and leave the other values as they are. Fill the file with:

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

urlpatterns = patterns('members.views',
    (r'', 'myFirstView'),
)
```

This file looks just like the first urls.py. Now however we used the first argument of the patterns command to tell we will use views located in `members/views.py`. The second line states that we will use the function `myFirstView` for every URL. Remember we already defined that the url has to start with `members/` in the project `urls.py` file, so this will be used for urls of the type `members/*`, where `*` is a wildcard.

Now start the django project as before and go to `http://localhost:8000/members/` and see your first real django webpage. Notice that the domain name (http://localhost:8000/) is stripped for url lookups, this makes it easy to deploy django on different domains, for example localhost for development and the real server for release.

3.4 How does URL matching work?

Up until now we didn't explain the exact details of how url lookups work. Actually the first part of each url definition is a regular expression. These can become quite complex but the basics are simple. Without any modifier the url will be used whenever that text is encountered in the url. so `r'test'` will resolve any url containing test, because of this `r''` will simply match any url. When you add `^` at the beginning you specify that the url should start with this text, similar when you add `$` add the end you specify that the url should end with this text. So `r'^members/'` will match both `members/` and `members/test` but not `test/test`. `r'test$'` will match `members/test` and `test/test` but not `members/`. Finally `r'^members/$` will only match `members/`.

4 Models and the django admin

In this section we will define two simple models to communicate with the database and we will show how to use these models in the built-in django database administration application. The models define a member and possible studies. The `Member` model contains some basic information like name and address and points to one `Study` object, which contains the name of the study and corresponding year. This way, when a study name changes we can easily change it for all members in this study, we do not have to update each member, just the study object. This process is called normalization and is very essential for modern database design. As a simple rule, when some attributes are common for multiple model objects, it's often a better choice to place them in their own model and point to these.

4.1 Creating the models

In the members application, open the models.py file and add the following two models to it.

```
class Study(models.Model):
    name = models.CharField(max_length=255)
    year = models.SmallIntegerField()

    def __unicode__(self):
        return self.name

class Member(models.Model):
    first_name = models.CharField(max_length=255)
    last_name = models.CharField(max_length=255)
    street = models.CharField(max_length=255)
    nr = models.IntegerField()
    postal_code = models.IntegerField()
    city = models.CharField(max_length=255)
    study = models.ForeignKey(Study)

    def __unicode__(self):
        return "%s, %s" %(self.last_name, self.first_name)
```

Each model is a class derived from the django base class `django.db.models.Model`. It contains some class variables that define which fields define the model. Each field defines a type that will be mapped to a corresponding type in the used database. Some types require extra arguments, e.g. to specify a maximum length. Most types also accept a lot of optional arguments that can be used to constrain the data, e.g. to specify a minimum or maximum value. When a Model object is created (an instance), all of these fields are also available as object variables (through the `self` keyword), in this case they specify the values rather than the fields.

Additionally, it is good practice to provide each model with a `__unicode__` method, which will be called when a textual representation is required, this is for example used in the built-in admin interface. In the code listing you can see how the `self` keyword is used to access object variables. Also, in the `Member` example it is shown how to do python string formatting, always use this method rather than pasting strings together with `+`.

4.2 Creating the database

While we now have a representation of our models in django, they still do not exist in the database. Django provides an easy command to create these straight from the models code. This command is called syncdb. You can call it by right clicking on the project and selecting `Django > Sync DB`. Note that if everything goes successful this will ask to create a superuser, type yes here and fill it out to your liking, we will need the account later on.

You can now see that a file `sqlite.db` is added to the project, this contains the database. Remember we used `sqlite3` as an engine, which stores databases as simple files. If something goes wrong and you want to start from scratch, just delete this file and run `syncdb` again.

4.3 Starting the django admin

Now that we have a some models we will show the easiest way to populate them with data, using the built-in django admin interface. To do this we must follow some simple steps. First we have to add our models to the admin system, therefore we create a `admin.py` file in the members application (similar to creating the `urls.py` file). The contents of these files look like this:

```
from django.contrib import admin
from models import Member, Study

admin.site.register(Member)
admin.site.register(Study)
```

This very basic admin script just adds each model to the default admin site. There are many more complex possibilities to customize the admin site to your preferences, which is the main reason this is placed in a separate file.

Finally, we need to activate the admin application and set up the urls to find it.

1. Open `settings.py`
2. Add `'django.contrib.admin',` to the `INSTALLED_APPS` list, just as when adding a new application.
3. Close this file and open `urls.py` in the main project.
4. Change it to look like this:

   ```
   from django.conf.urls.defaults import *
   from django.contrib import admin
   admin.autodiscover()

   urlpatterns = patterns('',
       (r'^members/', include("members.urls")),
       (r'^admin/', include(admin.site.urls)),
   )
   ```
5. Run syncdb again (the admin application added some new models).
6. Re-launch the django project as before

You can now surf to `http://localhost:8000/admin/` to start using the admin site with the superuser account that you created earlier. Add some members with corresponding studies, we will use those later.

5 Using models in views

In this section we will show how to use models in views, we will create a new function in the `views.py` file of the members project. This will display a list of members, so we will need to import the `Member` model, we do this by adding the following to the top of the file.

```
from models import Member
```

Next, we create the new function as follows.

```
def membersList(request):
    result = ""
    for m in Member.objects.all():
        result += "%s --- study: %s<br />" %(m,m.study)
    return HttpResponse(result)
```

This function simply loops over all member objects and adds the string representation (remember the `__unicode__` method) to the string result. Afterwards a normal `HttpResponse` object containing this string is returned.

Finally, add a new entry to the `urls.py` (in `members`), you can choose how you call it. Now try and surf to the url and if everything went well you'll see a list of all the members you've added through the admin.

6 Introducing templates

It may be clear from the previous part that it is not really useful to always define the string you want to return manually. Certainly when you want to display a real webpage this will contain a lot of HTML/CSS/JS/... code which you prefer to keep separated from the django code. This is solved by using templates. Essentially views will pass data to templates which will place it inside of html content using a very simple template language that can be easily learned by designers.

6.1 Setting up the template environment

We start by creating separate template folders in the `djangolesson` module (note: this is not the eclipse project but the python module with the same name right beneath it). Right click it and select `New > Folder`, name it templates. Now add another folder called members to the newly created templates folder. It is good practice to use the same structure for your templates folder as for your applications.

We also need to add this template directory to the `TEMPLATE_DIRS` variable in `settings.py`, to find out the whole folder, take a look at the database configuration at the beginning of this file. The result should look like this, make note this should contain the whole path, it can not be relative to the project directory.

```
TEMPLATE_DIRS = (
    '/home/killian/Documents/progging/python/djangolesson/djangolesson/templates',
)
```

6.2 Creating the template file

In the `templates/members` folder create a normal file and name it `list.html`. Note that eclipse will open this by default with a built-in browser. To edit the file right click on it and select `Open With > Text Editor`. Now enter following html code:

```
<!doctype html>
<html>
  <head>
    <title>Members list</title>
  </head>
  <body>
    {% for m in members %}
      {{ m }} --- study: {{ m.study }} <br />
    {% endfor %}
  </body>
</html>
```

As you can see the code consists of standard html, with some template-specific commands added. In general you should know that `{% command %}` executes a specific command and `{{ variable }}` displays a variable. A full list of built-in template tags and variables can be found in the django documentation[1].

In this example we use the `for` tag to loop over a list `members` which we received from the view, similar to a python for loop. Each item of this list will be a member object which we can handle in almost exactly the same way as in normal python code. We display the list very similar to the original view.

6.3 Using the template from a view

To use templates in views, most often the method `render_to_response` is used, this is a django function that does some complexer template processing behind the screens. It accepts 2 arguments, the template name and a python dictionary with the values it should pass. To use it, import it at the top of `views.py` as follows:

```
from django.shortcuts import render_to_response
```

The second argument passed is a python dict, this is a structure that has a key value relationship and is defined as follows:

```
values = {"members": Member.objects.all()}
```

In this simple example we define a dictionary `values` where we assign the key `"members"` to the value `Member.objects.all()` (a list of all members). In normal python code we could just call `values["members"]` to use it. Putting everything together we can change the view from the previous section to this.

```
def membersList(request):
    variables = {"members": Member.objects.all()}
    return render_to_response("members/list.html",variables)
```

When you restart the django server and go to the url you specified for this view, you can now see it is rendering HTML by inspecting the source of the webpage (for firefox: press ctrl-u).

7 Forms

While you already have all the building blocks for a basic website, we will go deeper in one last very useful and very powerful system that django provides: forms. Forms help taking responsibilities away from views, making the code lighter and easier to maintain and reuse. The main function of a Django form is to validate input and decide which action to take with that input. An additional feature is that django forms allow to easily generate HTML forms. Sadly enough, many people use forms only for this target, which results once again in long views to handle input processing.

To keep things simple we will create a basic form that asks for someone's full name, checks if it is longer than 3 characters and displays it if it's correct.

7.1 Creating the form

In the `members` application, create a new module called `forms.py`. The form should look like this.

```
from django import forms

class NameForm(forms.Form):
    name = forms.CharField()
```

[1] http://docs.djangoproject.com/en/dev/ref/templates/builtins/

```
    def getName(self):
        return self.cleaned_data["name"]

    def clean_name(self):
        name = self.cleaned_data.get("name","")
        if len(name) < 3:
            raise forms.ValidationError("Name should be at least 3 characters long")
        return name
```

Defining form fields is very similar to defining model fields. Note however that in forms the fields will not be available in an object, so you can not call `self.name` or `someform.name`. Therefore we added a method to get the name. A second method `clean_name` is added to validate the name field. A validator method for a specific field will always be called `clean_fieldname` We search for the `"name"` attribute in the `self.cleaned_data` dictionary, which is a dictionary containing the input the form received. It is possible that this field does not exist (when no input is given), so we use the `get` method of the dictionary to pass a default value, in this case an empty string. After getting the name we check if the length is smaller than 3 and if so we raise an errormessage which django will translate into an error message in the form itself. Please note that this is an illustration, the CharField offers a `min_length` attribute that does this validation for you.

7.2 Using the form

We create a new view that looks like this, note you should import the `NameForm` class and of course define an url for this view, the templates are covered later on.

```
def formView(request):
    f = NameForm(request.GET)
    if f.is_valid():
        return render_to_response("members/welcome.html", {"name": f.getName()})
    return render_to_response("members/form.html", {"form": f})
```

This very simple example creates the form with whichever input `request.GET` contains (can be nothing). GET is one of the possible methods HTTP provides to send input, a better method is POST but this requires a complexer setup in django due to built in security measures. After the creation of the form we check if the form is valid, this will run the `clean_*` methods. Depending on this result we choose between two templates to be rendered. Note that if no input is given, `request.GET` will be empty and the form will not validate, which results in rendering the form instead of the welcome message.

The `form.html` template is as follows, we only show the part between the `<body></body>` tags for simplicity.

```
    <form method="get" action=".">
     {{ form.as_p }}
     <input type="submit" value="enter" />
    </form>
```

We still have to define our own form HTML tag, which specifies the input method and target URL (`"."` meaning the same as the current URL). We also need to provide some way to submit the form, the easiest way is to include a standard submit button. Finally we can simply render a django form as an html form, in this case we use the `as_p` method which will use `<p>` tags to render the form.

Finally, the content of `welcome.html` is simply `Welcome, {{ name }}`, of course again surrounded with some standard HTML.

When you now go to the url you defined for `formView` you should be able to test your newly created form. Don't forget to add a trailing slash to the URL when you surf to it.

8 Conclusion

After completing this tutorial you should be familiar with the way django powered websites work. To further practice your skills I have two suggestions (order does not matter)

First, try to add an `activity` application to the `djangolesson` project where users can subscribe themselves too. This will help you understand the basics learned here even more and will require you too try some new things which you can't simply type over. As a hint, you'll probably need 2 models (Activity and Subscription). To create a subscription form, search for `ModelForm` in the django documentation, it will show you how you can easily generate django forms from models.

Second, follow the django tutorial on the official website `www.djangoproject.com`. While there is much overlap, the official tutorial is longer and goes into more detail about some aspects, it will give you more insight in how everything works.

After this, go experimenting, try to build your own simple site or join a team working on one. One clue: always keep the official django documentation and google close. When you encounter a problem that you think will take you many hours of coding, the odds are high it's already provided in django itself or by an external django application.