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.

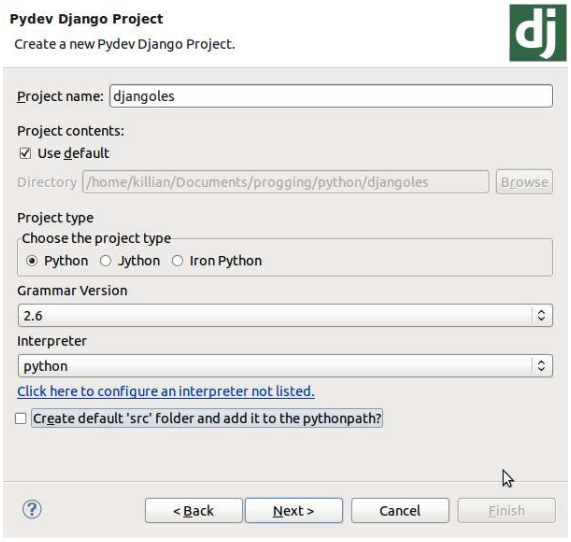

Figure 1: Example pydev project configuration

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.