djangoprojectscookbook

Release 2.0

Agiliq

Apr 24, 2018

Contents

1 Chapter 1: Introduction **5**
1.1 Virtual Environment (The Saviour) 5
1.2 Downloading and Installing 5
1.3 What is Django? 6
1.4 Django Features: 6
1.5 Inside Django 6

2 Chapter 2. Building a personal CD library. **9**
2.1 Starting a django project 9
2.2 Getting started with the App 12
2.3 An Introduction to the Django ORM 15

3 Chapter 3. Building a Pastebin. **21**
3.1 URL configuration - entry points 21
3.2 Templates - skeletons of our website: 22
3.3 Generic views - commonly used views: 25
3.4 Designing a pastebin app: 26
3.5 Sketch the models: 27
3.6 Configuring urls: 28
3.7 Writing custom management scripts: 36

4 Chapter 4. Building a Blog **39**
4.1 Topics in this chapter: 39
4.2 Our blog app: 41
4.3 Date based generic views: 46

5 Chapter 5. Building a Wiki **49**
5.1 A wiki application: 49
5.2 Article Management: 49

6 Chapter 6. Building a Quora like site **59**
6.1 Topics in this chapter: 59
6.2 Quora like Application: 59
6.3 Application Includes: 59
6.4 Django features to learn in this chapter: 59
6.5 Make custom user: 60
6.6 Class Based Views 61

6.7 Register Custom User 61
6.8 Basics of Django Testing: 68

7 Chapter 7. Building a Project management application **69**

8 Chapter 8. Building a Social news Site **71**

Django Projects Cookbook is a book for intermediate python programmers. It will take you from where Django tutorials left to a more advanced programmer.

UPDATED FOR
DJANGO 2.0
&
PYTHON 3.

DJANGO PROJECTS COOKBOOK

Learn advanced Django by building real projects

Contents:

CHAPTER 1

Chapter 1: Introduction

1.1 Virtual Environment (The Saviour)

The best time to learn about virtual environment is when you are a beginner in learning python/django as it will be very much helpful.

- **What is virtual environment ?**

  A virtual environment is a way for you to have multiple versions of python on your machine without them clashing with each other, each version can be considered as a development environment and you can have different versions of python libraries and modules all isolated from one another. For more information visit virtualenv

- **Importance of virtual environment.**

  Say, if you're working on an open source project that uses `django 1.7` but locally, you installed `django 2.0` for other project. It's almost impossible for you to contribute to open source because you'll get a lot of errors due to the difference in django versions. If you decide to downgrade to `django 1.7` then you can't work on your project anymore because that depend on `django 2.0`. Virtual environment lets you handle this situation by creating a separate virtual(development) environments that are not tied together and can be activated/deactivated easily whenever you want.

- **How does virtual environment work ?**

```
$ virtualenv venv
$ cd venv
$ source bin/activate
(venv)$                  // The current shell starts using the virtual environment.
(venv)$ deactivate       // virtual environment deactivated.
```

1.2 Downloading and Installing

Option 1

```
$ pip install Django==2.0.3
```

Option 2

```
$ git clone https://github.com/django/django.git // latest version of django
```

Note: Django 2.0+ versions are supported by Python 3+ versions. Django 1.11 LTS is the last version to be supported by Python 2.7.

1.3 What is Django?

- Django is a free open-source web framework, written in Python, which follows the model-view-template(MVT) architectural pattern. It is maintained by Django Software Foundation(DSF). Django's primary goal is to ease the creation of complex, database-driven websites.
- Django is a high-level Python Web framework that encourages rapid development and clean, pragmatic design. Built by experienced developers, it takes care of much of the hassle of Web development, so you can focus on writing your app without needing to reinvent the wheel.

1.4 Django Features:

1. Fast
2. Code Reusability
3. Security
4. Scalable
5. Versatile
6. Documentation
7. Community

1.5 Inside Django

- **Django Philosophy**

The web framework for perfectionists with deadlines.

- **Django Architecture**

Django follows the MVC pattern closely, however it does use its own logic in the implementation. Because the "C" is handled by the framework itself and most of the excitement in Django happens in models, templates and views, Django is often referred to as an MTV framework.

- **Django ORM**

ORMs provide a high-level abstraction upon a relational database that allows a developer to write Python code instead of SQL to create, read, update and delete data and schemas in their database. Developers can use the programming language they are comfortable with to work with a database instead of writing SQL statements or stored procedures.

- **DRY principle**

To help developers adhere to the DRY principle, Django forces users to use the MVC code structure. Django forces users to use this format by initially creating a views.py, models.py, and template files. By keeping the controller code separate from the views, it allows multiple controllers to use the same view.

- **Loose coupling**

A fundamental goal of Django's stack is loose coupling and tight cohesion. The various layers of the framework shouldn't "know" about each other unless absolutely necessary.

For example, the template system knows nothing about Web requests, the database layer knows nothing about data display and the view system doesn't care which template system a programmer uses.

- **Request-Response cycle**

Django uses request and response objects to pass state through the system.

When a page is requested, Django creates an HttpRequest object that contains metadata about the request. Then Django loads the appropriate view, passing the HttpRequest as the first argument to the view function. Each view is responsible for returning an HttpResponse object.

- **Middleware**

Middleware is a framework of hooks into Django's request/response processing. It's a light, low-level "plugin" system for globally altering Django's input or output.

- **Template tags**

Django's template language comes with a wide variety of built-in tags and filters designed to address the presentation logic needs of your application.

CHAPTER 2

Chapter 2. Building a personal CD library.

2.1 Starting a django project

Now that we have installed django, we are ready to start our project.

A project in django is analogous to a website. Once we have a project ready, we can host it using any wsgi supported server. More on deploying a django project later.

django-admin.py is a project utility that ships with django. In addition to the `startproject` subcommand, it also includes a lot of helper subcommands that can be useful while maintaining a django project.

Note: To get a full listing of available subcommands in django-admin.py, use

```
django-admin.py --help
```

To get help on each subcommand, use

```
django-admin.py help <subcommand>
```

For example, here's the result of `django-admin.py help startproject`

```
Creates a Django project directory structure for the given project name in the␣
↪current directory.
```

Lets create a project called djen_project:

```
django-admin.py startproject djen_project
```

We can see that the subcommand creates a folder and subfolder called djen_project in the working directory with the following files:

```
-- djen_project
- djen_project
```

```
    __init__.py
    settings.py
    urls.py
    wsgi.py
- manage.py
```

__init__.py is an empty file required to recognize this project as a python module.

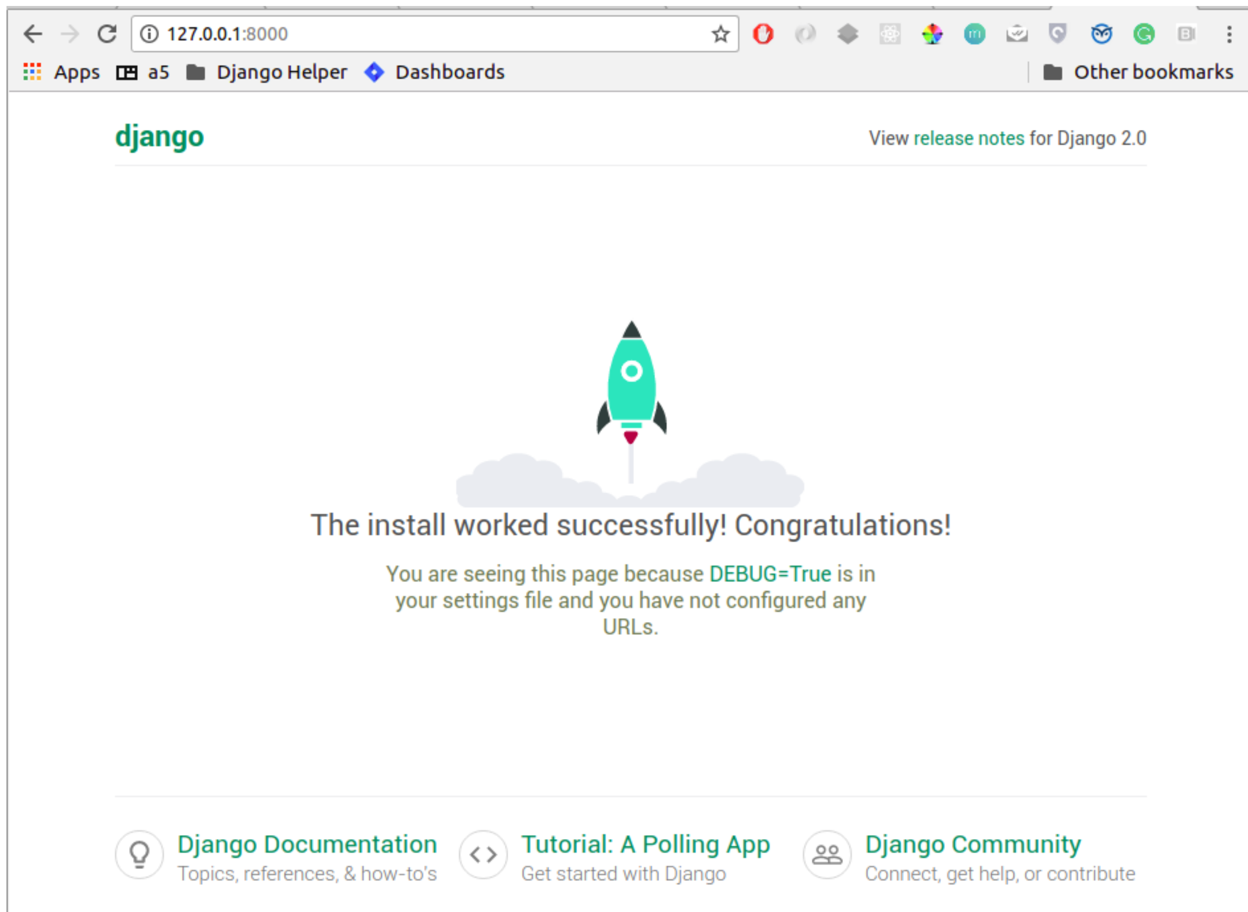
manage.py is a script that is similar to django-admin.py which allows you to manage this project. It has subcommands to start a development server, interact with database, backup/restore data etc.

Note: manage.py also has –help switch and help with each subcommand similar to django-admin.py

You can quickly checkout the development server at this point by running:

```
python manage.py runserver
```

Now open http://127.0.0.1:8000 in your browser. you will see django powered page.



settings.py is a list of project wide settings with some default values. You will need to edit this often when installing new django applications, deployment etc.

You can change the DATABASES settings at this point to make sure your app can be sync'ed later. The easiest settings would look like:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3', # Add 'postgresql_psycopg2',
↪'postgresql', 'mysql', 'sqlite3' or 'oracle'.
        'NAME': 'os.path.join(BASE_DIR, 'db.sqlite3')',      # Or path to database␣
↪file if using sqlite3.
        'USER': '',                      # Not used with sqlite3.
        'PASSWORD': '',                  # Not used with sqlite3.
        'HOST': '',                      # Set to empty string for localhost. Not␣
↪used with sqlite3.
        'PORT': '',                      # Set to empty string for default. Not used␣
↪with sqlite3.
    }
}
```

or for a mysql:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql', # Add 'postgresql_psycopg2', 'postgresql
↪', 'mysql', 'sqlite3' or 'oracle'.
        'NAME': 'djen_database',                          # Or path to database file if␣
↪using sqlite3.
        'USER': 'root',                      # Not used with sqlite3.
        'PASSWORD': '****',                  # Not used with sqlite3.
        'HOST': '',                      # Set to empty string for localhost. Not␣
↪used with sqlite3.
        'PORT': '',                      # Set to empty string for default. Not used␣
↪with sqlite3.
    }
}
```

We will be using mysql database for examples in this book. Of course, you are free to change the settings to any other database you like. Just make sure the database exists.

Note: To verify your database settings run:

```
python manage.py check
```

Django will validate your settings and show you errors, if any. If you get this error:

```
_mysql_exceptions.OperationalError: (1049, "Unknown database...
```

make sure the database given in the settings exists.

Note: It is advised to have a local_settings.py file with exclusively server specific and sensitive settings like database username/password, API keys or Secret Key etc and have settings.py import all these values.

To do this, you would create local_settings.py and include `from local_settings import *` at the bottom of settings.py

urls.py is a 'table of contents' of our project (or website). It includes a list of the paths that are to be processed and responded to.

You are encouraged to go through settings.py and urls.py once to get an understanding of how settings and urls are defined.

2.2 Getting started with the App

Now that we have setup and understood the structure of our project, we can start our application.

To start an application, cd into the project directory and use

```
manage.py startapp cd_library
```

This will create a folder called cd_library with the following files:

```
__init__.py
admin.py
app.py
models.py
tests.py
views.py
```

__init__.py is again the file that allows this app to be considered a python module.

models.py will hold the Models of our application. A model is an object of our interest which we want to save to the database. If you are familiar with Model-View-Controller(MVC) architecture, you know what models are. If no, don't worry, we will see and use them in our application.

views.py has all the 'action' of our website. This is similar to the Controller of MVC architecture. Each 'view' function takes a `request` object and returns a `HttpResponse` object.

Note: It is recommended to have another urls.py (like the one in project) in the app and include them in the project urls. This reduces the clutter in the project urls and provides a namespace kind of resolution between urls. Also, it makes it easier to redistribute the app to other projects. As you would expect, reusable apps will depend on the project as little as possible.

Let us create our models first:

Open models.py and define our `CD` model which will hold all information related to a CD. You can see that models.py has:

```
from django.db import models
```

So we define the `CD` model as:

```
GENRE_CHOICES = (
                 ('R', 'Rock'),
                 ('B', 'Blues'),
                 ('J', 'Jazz'),
                 ('P', 'Pop'),
                )

class CD(models.Model):
    title = models.CharField(max_length=100)
    description = models.TextField(null=True, blank=True)
    artist = models.CharField(max_length=40)
    date = models.DateField()
    genre = models.CharField(max_length=1, choices=GENRE_CHOICES)
```

```
    def __unicode__(self):
        return "%s by %s, %s" %(self.title, self.artist, self.date.year)
```

A little explanation:

- All models should be a subclass of django.db.models.Model
- Each model has a list of fields which will define that model
- We have used CharField, TextField and DateField in this model.
- Each CharField requires a max_length argument which specifies the maximum length of the characters that the field can hold.
- A TextField can contain any number of characters and is suitable for fields such as description, summary, content etc.
- To make the description field optional, we pass the `null` and `blank` arguments as True
- DateField holds a date. If you need to store the time too, use DateTimeField instead.
- The `genre` field should be restricted to a group of values and that can be accomplished by passing an iterable of 2-tuples for the value and representation as the choices argument of the CharField.
- The __unicode__ property of the model defines it's string representation which will be used in the Admin interface, shell etc.

So far, we have defined the CD model, now we need to get it rolling in django:

First, let django know that `cd_library` is to be used in the project. To do this, edit the project settings.py and add:

```
'cd_library'
```

to the INSTALLED_APPS list so that your settings.py looks like this:

```
INSTALLED_APPS = [
'django.contrib.admin',
'django.contrib.auth',
'django.contrib.contenttypes',
'django.contrib.sessions',
'django.contrib.messages',
'django.contrib.staticfiles',
'cd_library',
]
```

Note: After modifying INSTALLED_APPS, it's always a good idea to run makmigrations and migrate:

```
$ python manage.py makemigrations

$ python manage.py migrate
```

This lets django keep the database and your project in sync. Since we have added an app, django will create that app's tables in the database. If an app is removed from the above list, django will ask you whether to remove the 'stale' tables.

Also, make sure you have DATABASES settings correctly pointed to the database before syncing.

Well, now that django knows about our app, let us add it to the Admin interface.

A little bit about the admin interface first:

- The admin interface is itself a django app.
- It is a contrib app, which means it is a community contributed app
- It is flexible enough to accommodate any other app's models and have admin actions for them.

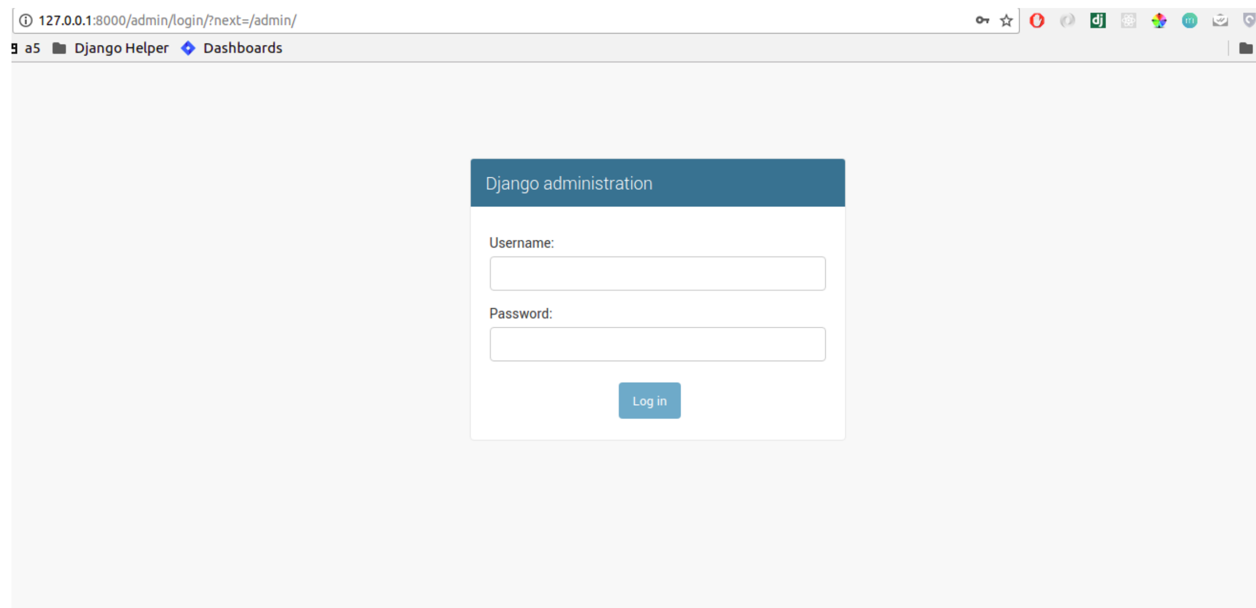
Note that the admin app uses a urls.py to keep its urls separate from the project (as discussed in the note above).

You should create a superuser, to login to django's inbuild admin panel.

```
$ python manage.py createsuperuser
Username (leave blank to use 'agiliq'):
Email address: user@agiliq.com
Password:                    # password won't be visible for security reasons.
Password (again):
Superuser created successfully.
```

Remember username and password for logging into the admin panel.

Just to confirm it, you can open http://127.0.0.1:8000/admin/ in your browser. You should see 'Site Administration' and actions for 'Authentication and Authorization' which are enabled by default.



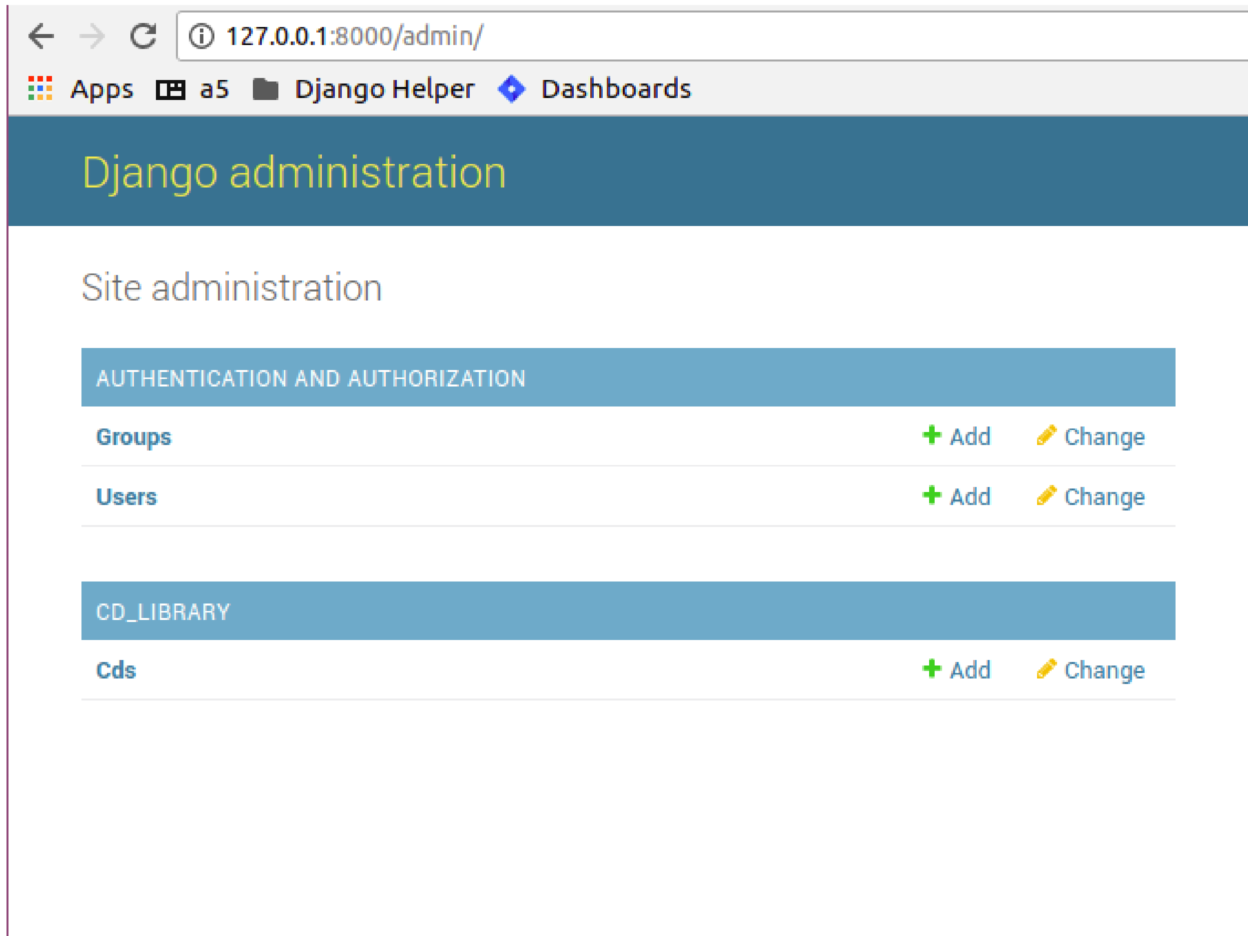
Now to enable our app's models:

Go to the app's directory i.e. cd_library and create a file called admin.py and add the following lines:

```
from django.contrib import admin
from .models import CD

admin.site.register(CD)
```

So, we have 'registered' our `CD` model with the admin interface.

If you refresh the admin page, you can see the 'Cd_library' header and 'Cds' under it. Yes this is our app's model and we can add/edit/delete any instances of our CD model through the admin interface. Try adding a few entries using the `Add` action. You can edit entries using the `Change` action which will take you to the change list page. Try editing and deleting entries.



Did you notice?:

- Django uses the models __unicode__ property to display the CD in the change list
- Django used our model field types (CharField, TextField, DateField) to create HTML widgets in the admin page
- Genre Field has a drop down field with the CHOICES attributes used to populate its key, value pairs
- DateField includes a handy calendar popup
- Description is optional, so it is not highlighted like the rest of the fields
- Django provides automatic form validation. Try entering blank values, or wrong dates and submitting the form
- In accordance with the DRY principle, models.py is the only place where you specified the fields

With this, we have built our own personal CD library.

2.3 An Introduction to the Django ORM

Now, lets take a look at the raw data that django stores for us.

We have configured the database django uses in 'DATABASES' attribute of settings.py. Notice that you can enter multiple database settings and use them by providing the `--database` switch to manage.py subcommands.

To go to the database shell and view the database, use:

```
python manage.py dbshell
```

'dbshell' is a handy manage.py subcommand that will give you access to the database using your DATABASES settings You can check the tables in the database by doing:

```
.tables for  sqlite
show tables for mysql
\dt for pgsql
```

Since we are using mysql for this example, the result is:

```
mysql> show tables;
+----------------------------+
| Tables_in_djen_database    |
+----------------------------+
| auth_group                 |
| auth_group_permissions     |
| auth_permission            |
| auth_user                  |
| auth_user_groups           |
| auth_user_user_permissions |
| cd_library_cd              |
| django_admin_log           |
| django_content_type        |
| django_migrations          |
| django_session             |
+----------------------------+
11 rows in set (0.00 sec)
```

Each table generally represents a model from an app. You can see that the `CD` model is saved as `cd_library_cd` table.

Well, lets look at the structure of this table:

```
mysql> desc cd_library_cd;
+-------------+--------------+------+-----+---------+----------------+
| Field       | Type         | Null | Key | Default | Extra          |
+-------------+--------------+------+-----+---------+----------------+
| id          | int(11)      | NO   | PRI | NULL    | auto_increment |
| title       | varchar(100) | NO   |     | NULL    |                |
| description | longtext     | YES  |     | NULL    |                |
| artist      | varchar(40)  | NO   |     | NULL    |                |
| date        | date         | NO   |     | NULL    |                |
| genre       | varchar(1)   | NO   |     | NULL    |                |
+-------------+--------------+------+-----+---------+----------------+
6 rows in set (0.00 sec)
```

I have added a few entries to the CD model, so lets see if they are here:

```
mysql> SELECT * FROM cd_library_cd;
+----+-------+-------------+-----------+------------+-------+
| id | title | description | artist    | date       | genre |
+----+-------+-------------+-----------+------------+-------+
|  1 | Kid A |             | Radiohead | 2010-01-01 | R     |
+----+-------+-------------+-----------+------------+-------+
1 row in set (0.01 sec)
```

Note: Primary key field for an object (id in this case) is autogenerated by django. If you need a custom primary key, pass primary_key=True in the field.

Django's Object Relational Mapper (ORM) worked behind the scenes to create the tables, sync them with the models, and add/edit/delete entries to the tables.

Now lets try out the ORM first hand. Use the `shell` subcommand of `manage.py`:

```
python manage.py shell
```

Note: use ipython shell for tab-completion, reverse history search and more. django will automatically use ipython shell if available

This will take you to the python shell, but within the django environment. So now you can interact with your project

A few examples:

```
from cd_library.models import CD
```

retrieve all cds:

```
cds = CD.objects.all()
```

loop through the cds and print their names:

```
for cd in cds:
    print cd
```

add a new CD:

```
new_cd = CD()
new_cd.title = "OK Computer"
new_cd.artist = "Radiohead"
new_cd.date = "2000-01-01"
new_cd.genre = "R"
new_cd.save()
```

Note: a model is never saved to the database until the save method is explicitly called

Whats all this:

- Our CD model is mapped to a database table
- A default primary key is used since we have not primary_key on any of the fields
- The default primary key is of the type int and is autoincremented
- The table fields are selected automatically based on model fields

That is really the core of the work of the ORM: mapping classes (or models) to tables. While doing so, django takes care of the conversion of model fields to database columns, type conversions, primary keys, constraints and all of that. Thanks to the ORM, you don't have to deal with the databases directly. In fact, if you were to switch the underlying database by modifying DATABASES in settings, your application would be least affected by it.

Now that you know a little bit about the ORM, lets see some more utilities it provides:

the object manager:

```
CD.objects
```

objects refers to the default object manager. A manager provides the way of dealing with the database. Custom managers can be used to provide different 'views' of the model. More on that later.

the get method - to get a single object:

```
CD.objects.get(pk=1)
```

returns:

```
<CD: OK Computer by Radiohead, 2000>
```

that is, a single instance of our CD model. The arguments **must** return a unique object or else this method will raise `MultipleObjectsReturned` error.

Note: arguments to the manager methods include pk for primary key, all model fields and some operators called lookups

---

Use the get method on when you want to retrieve one record based on the given criteria.

the filter method - to filter the list using given criteria:

```
CD.objects.filter(artist='Radiohead')
```

to get all CDs by Radiohead, returns:

```
[<CD: OK Computer by Radiohead, 2000>, <CD: Kid A by Radiohead, 2010>]
```

which is a list of model instances

the exclude method - equal to all-filter:

```
CD.objects.exclude(title='OK Computer')
```

returns:

```
[<CD: Kid A by Radiohead, 2010>]
```

Now on to the lookups:

How do we get all CDs of the year 2000?

The object manager methods have some special arguments to operate on the fields

to pass date.year as the argument:

```
CD.objects.filter(date__year='2000')
```

Or, get the CDs in genres 'Rock' and 'Pop':

```
CD.objects.filter(genre__in=['R', 'P'])
```

A few other useful lookups:

```
title__startswith
title__endswith
date__lte
date__gte
title__contains
```

Use shell to experiment with object manager methods and lookups.

CHAPTER 3

Chapter 3. Building a Pastebin.

3.1 URL configuration - entry points

We have already noticed urls.py in our project. This controls our website's points of entry. All incoming urls will be matched with the regexes in the `urlpatterns` and the view corresponding to the first match will get to handle the request. A request url that does not match any urlconf entry will be 404'ed.

Note: brush up regexes in python from python docs or diveintopython

As an example from our previous app:

```
from django.contrib import admin
from django.urls import path

urlpatterns = [
    path('admin/', admin.site.urls),
]
```

Now when we call http://127.0.0.1:8000/admin/ django matches that to the url entries. This urlconf has included `admin.urls` which means that all further paths matches will be done with the `admin.urls` module. Once again, the first match will get to handle the request. You can think of this as 'mounting' the admin app at `/admin/`. You are of course free to change the 'mount point' to anything else you like.

A typical urlconf entry looks like this:

```
(r'<regex>', <view_function>, <arg_dict>),
```

`regex` is any valid python regex that has to be processed. This would be absolute in the project urls.py and relative to the mount point in an app's urls.py

`view_function` is a function that corresponds to this url. The function **must** return a `HttpResponse` object. Usually, shortcuts such as `render`, are used though. More about views later.

`arg_dict` is an optional dict of arguments that will be passed to the `view_function`. In addition, options can be declared from the url regex too. For example:

```
path('object/<int:id>/', views.get_object)
```

will match all urls having an integer after `object/`. Also, the value will be passed as `object_id` to the `get_object` function.

3.1.1 Named urls:

Usually, we would want an easier way to remember the urls so that we could refer them in views or templates. We could *name* our urls by using the `path` constructor. For example:

```
path(r'^welcome/$', 'app.views.welcome', name='welcome'),
```

This line is similar to the previous urls, but we have an option of passing a `name` argument.

To get back the url from its name, django provides:

- `django.url.reverse` function for use in views
- `url` templatetag for use in templates

We will see how to use the templatetag in our templates.

Note: Also see http://agiliq.com/books/djangodesignpatterns/urls.html#naming-urls

3.2 Templates - skeletons of our website:

You must be wondering where all those pages came from, since we have not touched any html yet. Well, since we used the admin app, we were able to rely on the admin templates supplied with django.

A template is a structure of webpage that will be *rendered* using a *context* and returned as response if you want it to. A `django.template.Template` object can be rendered using the `render` method.

Normally templates are html files with some extra django content, such as templatetags and variables. Note that our templates need not be publicly accessible(in fact they shouldn't be) from a webserver. They are not meant to be displayed directly; django will process them based on the request, context etc and respond with the rendered templates.

In case you want a template to be directly accessible (e.g. static html files), you could use the `django.views.generic.TemplateView` generic view.

3.2.1 Template Loaders:

By default, Django uses a filesystem-based template loader, but Django comes with a few other template loaders, which know how to load templates from other sources.

Some of these other loaders are disabled by default, but you can activate them by adding a 'loaders' option to your DjangoTemplates backend in the TEMPLATES setting or passing a loaders argument to Engine. loaders should be a list of strings or tuples, where each represents a template loader class. Here are the template loaders that come with Django:

django.template.loaders.filesystem.Loader

```
TEMPLATES = [{
    'BACKEND': 'django.template.backends.django.DjangoTemplates',
    'OPTIONS': {
        'loaders': [
            (
                'django.template.loaders.filesystem.Loader',
                [os.path.join(BASE_DIR, 'templates')],
            ),
        ],
    },
}]
```

django.template.loaders.app_directories.Loader

Loads templates from Django apps on the filesystem. For each app in INSTALLED_APPS, the loader looks for a templates subdirectory. If the directory exists, Django looks for templates in there.

This means you can store templates with your individual apps. This also makes it easy to distribute Django apps with default templates.

For example, for this setting:

```
INSTALLED_APPS = ['cd_library', 'pastebin']
```

You can enable this loader simply by setting APP_DIRS to True:

```
TEMPLATES = [{
    'BACKEND': 'django.template.backends.django.DjangoTemplates',
    'APP_DIRS': True,
}]
```

3.2.2 Context:

A context is a dict that will be used to render a page from a template. All context keys are valid template variables.

To display a user name in your template, suppose you provide the `username` in your context, you could do:

```
Hello {{ username }}
```

When this template is rendered using (e.g. using `render`), `username` will be replaced with its value

You can pass any variable to the context, so you can call a dict's key, or an objects property. However you cannot pass any arguments to the property.

For example:

```
Hello {{ user.username }}
```

can be used to get `user['username']` or `user.username`

Similarly:

```
<a hef="{{ user.get_absolute_url }}">{{ user.username }}</a>
```

can be used to get `user.get_absolute_url()`

3.2.3 Templatetags:

Templatetags are helpers to the template. Suppose you have an `iterable` with a list of objects in your context:

```
{% for object in objects %}
    {{ object }}
{% endfor %}
```

would render them. If this is a html template, we would prefer:

```
{% if objects %}
<ul>
    {% for object in objects %}
        <li>
            {{ object }}
        </li>
    {% endfor %}
</ul>
{% endif %}
```

which would render the objects in html unordered list.

Note that `{% if %}` `{% for %}` `{% endif %}` `{% endfor %}` are all built-in templatetags. If and for behave very much like their python counterparts.

3.2.4 Common templatetags and template inheritance:

Some templatetags we will use in our application:

- `url`

This templatetag takes a named url or view function and renders the url as found by `reverse`

For example:

```
<a href="{% url 'pastebin_paste_list' %}">View All</a>
```

would output

```
<a href="/pastebin/pastes/">View All</a>
```

It also takes arguments:

```
<a href="{% url pastebin_paste_detail paste.id  %}">{{ paste }}</a>
```

would output

```
<a href="/pastebin/paste/9">Sample Paste</a>
```

Note: You must make sure the correct urlconf entry for the give url exists. If the url entry does not exist, or the number of arguments does not match, this templatetag will raise a `NoReverseMatch` exception.

- `csrf_token`

  This is a security related tag used in forms to prevent cross site request forgery.

- `include <template>`

  This will simply include any file that can be found by the `TEMPLATE_LOADERS` where it is called

- `extends <template>`

  This will extend another template and provides template inheritance. You can have a `base` template and have other specific template extend the `base` template.

- `block` and `endblock`

  `blocks` are used to customize the `base` page from a `child` page. If the `base` page defines a block called
  `head`, the child page can override that block with its own contents.

- `load`

  This is used to load custom templatetags. More about writing and using custom templatetags later.

We will see later how to add custom templatetags.

3.2.5 Filters:

Filters are simple functions which operate on a template variable and manipulate them.

For example in our previous template:

```
Hello {{ username|capfirst }}
```

Here `capfirst` is a filter that will capitalize the first char our `username`

Note: Reference of built-in templatetags and filters: https://docs.djangoproject.com/en/2.0/ref/templates/builtins/

3.2.6 Templates are not meant for programming:

One of the core django philosophy is that templates are meant for rendering the context and optionally making a few aesthetic changes only. Templates should not be used for handling complex queries or operations. This is also useful to keep the programming and designing aspects of the website separate. Template language should be easy enough to be written by designers.

3.3 Generic views - commonly used views:

3.3.1 Views:

Views are just functions which take the `HttpRequest` object, and some optional arguments, then do some work and return a `HttpResponse` page. Use `HttpResponseRedirect` to redirect to some other `url` or `HttpResponseForbidden` to return a `403 Forbidden` response.

By convention, all of an app's views would be written in <app>/views.py

A simple example to return "Hello World!" string response:

```
from django.http import HttpResponse

def hello_world(request):
    return HttpResponse("Hello World!")
```

To render a template to response one would do:

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

def hello_world(request):
    template = loader.get_template("hello_world.html")
    context = {"username": "Monty Python"}
    return HttpResponse(template.render(context))
```

But there's a simpler way:

```
from django.shortcuts import renders

def hello_world(request):
        return render(request,"hello_world.html", {"username": "Monty Python"})
```

3.3.2 Generic Views:

Django's generic views were developed to ease that pain.They take certain common idioms and patterns found in view development and abstract them so that you can quickly write common views of data without having to write too much code.

3.3.3 Extending Generic Views

There's no question that using generic views can speed up development substantially. In most projects, however, there comes a moment when the generic views no longer suffice. Indeed, the most common question asked by new Django developers is how to make generic views handle a wider array of situations.

This is one of the reasons generic views were redesigned for the 1.3 release - previously, they were just view functions with a bewildering array of options; now, rather than passing in a large amount of configuration in the URLconf, the recommended way to extend generic views :Qis to subclass them, and override their attributes or methods.

Note: reference: https://docs.djangoproject.com/en/2.0/topics/class-based-views/generic-display/

3.4 Designing a pastebin app:

In this chapter we will be designing a simple pastebin. Our pastebin will be able to

- Allow users to paste some text
- Allow users to edit or delete the text
- Allow users to view all texts
- Clean up texts older than a day

Some 'views' that the user will see are

- A list view of all recent texts
- A detail view of any selected text
- An entry/edit form for a text

- A view to delete a text

Our work flow for this app would be

- sketch the models
- route urls to generic views
- use generic views with our models
- write the templates to use generic views

So let's dive in:

3.5 Sketch the models:

We have only one object to store to the database which is the text pasted by the user. Let's call this Paste.

Some things our Paste model would need to handle are

- Text pasted by the user
- Optional file name
- Created time
- Updated time

The time fields would be useful for getting 'latest' or 'recently updated' pastes.

So let's get started:

```
python manage.py startapp pastebin
```

In pastebin/models.py

```
from django.db import models

# Create your models here.
class Paste(models.Model):
    text = models.TextField()
    name = models.CharField(max_length=40, null=True, blank=True)
    created_on = models.DateTimeField(auto_now_add=True)
    updated_on = models.DateTimeField(auto_now=True)

    def __unicode__(self):
        return self.name or str(self.id)
```

Note:

- auto_now_add automatically adds current time to the created_on field when an object is added.
- auto_now is similar to the above, but it adds the current time to the updated_on field each time an object is saved.
- the id field is primary key which is autocreated by django. Since name is optional, we fall back to the id which is guaranteed.

Adding our app to the project

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'cd_library',
    'pastebin',
]
```

Makemigrations and Migrate:

```
$ python manage.py makemigrations
Migrations for 'pastebin':
  pastebin/migrations/0001_initial.py
    - Create model Paste
$ python manage.py migrate
Operations to perform:
    Apply all migrations: admin, auth, cd_library, contenttypes, pastebin, sessions
Running migrations:
    Applying pastebin.0001_initial... OK
```

There, we have our pastebin models ready.

3.6 Configuring urls:

We have already seen how to include the admin urls in urls.py. But now, we want to have our app take control of the urls and direct them to generic views. Here's how

Let's create urls.py in our app. Now our pastebin/urls.py should look like

```
from django.urls import path
from .views import PasteCreate

urlpatterns = [
    path(r'', PasteCreate.as_view(), name='create'),
]
```

Notes:

- Each urlpatterns line is a mapping of urls to views

  `path(r'', PasteCreate.as_view(), name='create'),`

- Here the url is `''` will be matched with the incoming request. If a match is found, the request is forwarded to the corresponding view.
- The scope goes to class based generic view, which is written in our `views.py`.

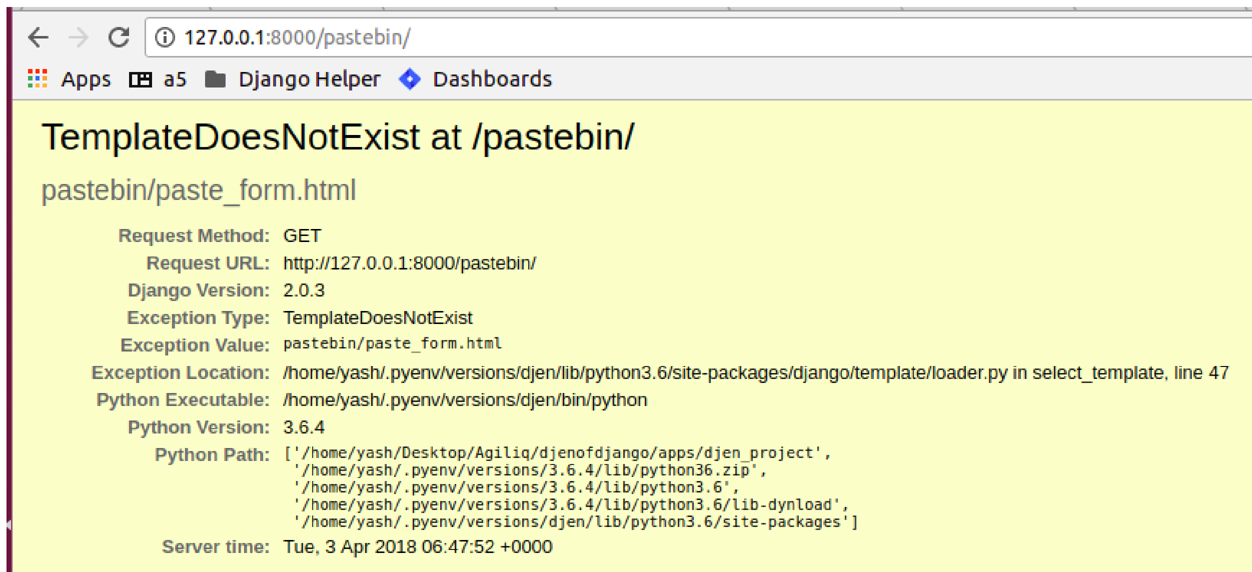
```
from django.views.generic.edit import CreateView

class PasteCreate(CreateView):
    model = Paste
    fields = ['text','name']
```

Let's tell the project to include our app's urls

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('pastebin/', include('pastebin.urls')),
]
```

Now django knows to forward urls starting with `/pastebin` to the pastebin app. All urls relative to this url will be handled by the pastebin app. That's great for reusability.

If you try to open http://127.0.0.1/pastebin at this point, you will be greeted with a TemplateDoesNotExist error. If you observe, the error message says that django cannot find `pastebin/paste_form.html`. Usually getting this error means that django was not able to find that file.



The default template used by CreateView is '<app>/<model>_form.html'. In our case this would be `pastebin/paste_form.html`.

Let's create this template. In `templates/pastebin/paste_form.html`:

```
<h1>Create new Paste</h1>
<form action="" method="POST">
    {% csrf_token %}
    <table>
        {{ form.as_table }}
    </table>
    <input type="submit" name="create" value="Create">
</form>

<a href="{% url 'pastebin_paste_list' %}">View All</a>
```

Just after adding the template we an refresh the page. We will se our webpage as.

Observe that:

- the form has been autogenerated by django's forms library by using the `Paste` model
- to display the form, all you have to do is render the `form` variable
- form has a method `as_table` that will render it as table, other options are `as_p`, `as_ul` for enclosing the form in `<p>` and `<ul>` tags respectively
- form does not output the form tags or the submit button, so we will have to write them down in the template
- you need to include `csrf_token` tag in every form posted to a local view. Django uses this to prevent cross site request forgery
- the generated form includes validation based on the model fields

Now, we need a page to redirect successful submissions to. We can use the detail view page of a paste here.

For this, we will use the `django.views.generic.detail.DetailView`

```
from django.views.generic.detail import DetailView
from .models import Paste
from django.views.generic.edit import CreateView

class PasteCreate(CreateView):
    model = Paste
    fields = ['text','name']
```

```python
class PasteDetail(DetailView):
    model = Paste
    template_name = "pastebin/paste_detail.html"
```

Related urls:

```python
from django.urls import path
from .views import PasteList, PasteDetail, PasteCreate

urlpatterns = [
    path('', PasteCreate.as_view(), name='create'),
    path('paste/<int:pk>', PasteDetail.as_view(), name='pastebin_paste_detail'),
]
```

Using this generic view we will be able to display the details about the paste object with a given id. Note that:

- model and template_name are the arguments passed to DetailView. (ProjectDetailView)
- we are naming this view using the url constructor and passing the `name` argument. This name can be referred to from views or templates and helps in keeping this DRY.
- the DetailView view will render the `pastebin/paste_detail.html` template. We need to write down this template for this view to work.

In `templates/pastebin/paste_detail.html`:

```html
<label>Paste Details: </label>
<p>
    <div>
        <label>ID</label>
        <span>{{ object.id }}</span>
    </div>
    <div>
        <label>Name</label>
        <span>{{ object.name }}</span>
    </div>
    <div>
        <label>Text</label>
        <span>{{ object.text }}</span>
    </div>
    <div>
        <label>Created</label>
        <span>{{ object.created_on }}</span>
    </div>
    <div>
        <label>Modified</label>
        <span>{{ object.updated_on }}</span>
    </div>
</p>
```

Now, that we have a create view and a detail view, we just need to glue them together. We can do this in two ways:

- pass the `post_save_redirect` argument in `create_object` view
- set the `get_absolute_url` property of our Paste model to its detail view. `create_object` view will call the object's `get_absolute_url` by default

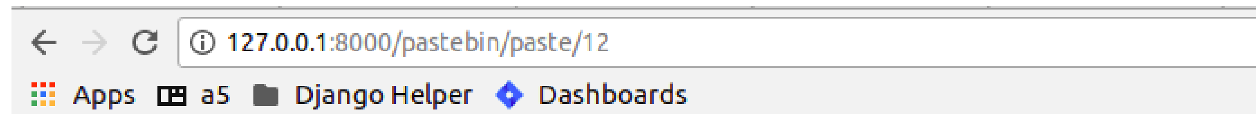
I would choose the latter because it is more general. To do this, change your Paste model and add the get_absolute_url property:

```
from django.db import models


class Paste(models.Model):
    text = models.TextField()
    name = models.CharField(max_length=40, null=True, blank=True)
    created_on = models.DateTimeField(auto_now_add=True)
    updated_on = models.DateTimeField(auto_now=True)

    def __unicode__(self):
        return self.name or str(self.id)

    @models.permalink
    def get_absolute_url(self):
        return ('pastebin_paste_detail', [self.id])
```

Note that:

- We could have returned `'/pastebin/paste/%s' %(self.id)'` but it would mean defining the same url twice and it violates the DRY principle. Using the `models.permalink` decorator, we can tell django to call the url named `pastebin_paste_detail` with the parameter `id`

And so, we are ready with the create object and object detail views. Try submitting any pastes and you should be redirected to the details of your paste.



Paste Details:

ID 12
Name Creating New Paste.
Text
```
This Post is for demo purpose, which will help you
to check the functionality of the detail view.
```
Created April 3, 2018, 7:10 a.m.
Modified April 3, 2018, 7:10 a.m.

Now, on to our next generic view, which is ListView:

```
from django.urls import path
from .views import PasteList, PasteDetail, PasteCreate

urlpatterns = [
    path('', PasteCreate.as_view(), name='create'),
    path('pastes/', PasteList.as_view(), name='pastebin_paste_list'),
    path('paste/<int:pk>', PasteDetail.as_view(), name='pastebin_paste_detail'),
```

```
]
```

This is simpler than the detail view, since it does not take any arguments in the url. The default template for this view is `pastebin/paste_list.html` so let's fill that up with:

```
{% if object_list %}
    <h1>Recent Pastes:</h1>
<ul>
    {% for paste in object_list %}
    <li>
        <a href="{% url 'pastebin_paste_detail' paste.id  %}">{{ paste }}</a>
    </li>
    {% endfor %}
</ul>
{% else %}
    <h1>No recent pastes</h1>
{% endif %}
```

Note that

- We have used the `url` template tag and passed our named view i.e. `pastebin_paste_detail` to get the url to a specific paste

Similarly, our update and delete generic views would look like

```
from django.urls import path
from .views import PasteList, PasteDetail, PasteDelete, PasteUpdate, PasteCreate

urlpatterns = [
    path('', PasteCreate.as_view(), name='create'),
    path('pastes/', PasteList.as_view(), name='pastebin_paste_list'),
    path('paste/<int:pk>', PasteDetail.as_view(), name='pastebin_paste_detail'),
    path('paste/delete/<int:pk>', PasteDelete.as_view(), name='pastebin_paste_delete
↪'),
    path('paste/edit/<int:pk>', PasteUpdate.as_view(), name='pastebin_paste_edit'),
]
```

Note that the `delete_object` generic view requires an argument called `post_delete_redirect` which will be used to redirect the user after deleting the object.

We have used update_object, delete_object for the update/delete views respectively. Let's link these urls from the detail page:

```
{% if messages %}
    <div class="messages">
    <ul>
    {% for message in messages %}
        <li class="{{ message.tag }}">
            {{ message }}
        </li>
        {% endfor %}
    </ul>
    </div>
{% endif %}

<h1>Paste Details: </h1>
<p>
    <div>
        <label>ID</label>
```

```
        <span>{{ object.id }}</span>
    </div>
    <div>
        <label>Name</label>
        <span>{{ object.name }}</span>
    </div>
    <div>
        <label>Text</label>
        <textarea rows="10" cols="50" OnClick="this.select();" readonly="true">{{
→object.text }}</textarea>
    </div>
    <div>
        <label>Created</label>
        <span>{{ object.created_on }}</span>
    </div>
    <div>
        <label>Modified</label>
        <span>{{ object.updated_on }}</span>
    </div>
</p>

<h2>Actions</h2>
    <ul>
        <li>
            <a href="{% url 'pastebin_paste_edit' object.id %}">Edit this paste</a>
        </li>
        <li>
            <a href="{% url 'pastebin_paste_delete' object.id %}">Delete this paste</
→a>
        </li>
    </ul>

<a href="{% url 'pastebin_paste_list' %}">View All</a>
```

Our `views.py` for complete pastebin looks like

```
from django.urls import reverse_lazy
from django.views.generic import DeleteView
from django.views.generic.edit import CreateView, UpdateView
from django.views.generic.detail import DetailView
from django.views.generic.list import ListView
from .models import Paste

class PasteCreate(CreateView):
    model = Paste
    fields = ['text','name']

class PasteList(ListView):
    model = Paste
    template_name = "pastebin/paste_list.html"
    queryset = Paste.objects.all()
    context_object_name = 'queryset'

class PasteDetail(DetailView):
    model = Paste
    template_name = "pastebin/paste_detail.html"

class PasteDelete(DeleteView):
```

```
    model = Paste
    success_url = reverse_lazy('pastebin_paste_list')

class PasteUpdate(UpdateView):
    model = Paste
    fields = ['text', 'name']
```

Note that the delete view redirects to a confirmation page whose template name is `paste_confirm_delete.html` if called using GET method. Once in the confirmation page, we need need to call the same view with a POST method. The view will delete the object and pass a message using the messages framework.

```
<h1>Really delete paste {{ object }}?</h1>
<h2>This action cannot be undone</h2>
<form action="{% url 'pastebin_paste_delete' object.id %}" method="POST">
    {% csrf_token %}
    <input type="submit" value="Delete">
</form>
```

Let's handle the message and display it in the redirected page.

```
{% if messages %}
    <div class="messages">
    <ul>
    {% for message in messages %}
        <li class="{{ message.tag }}">
            {{ message }}
        </li>
        {% endfor %}
    </ul>
    </div>
{% endif %}

{% if object_list %}
    <h1>Recent Pastes:</h1>
<ul>
    {% for paste in object_list %}
    <li>
        <a href="{% url 'pastebin_paste_detail' paste.id  %}">{{ paste }}</a>
    </li>
    {% endfor %}
</ul>
{% else %}
    <h1>No recent pastes</h1>
{% endif %}

<a href="{% url 'pastebin_paste_create' %}">Create new</a>
```

While we are at it, Let's also include the messages in paste detail page, where create/update view sends the messages:

```
{% if messages %}
    <div class="messages">
    <ul>
    {% for message in messages %}
        <li class="{{ message.tag }}">
            {{ message }}
        </li>
        {% endfor %}
    </ul>
```

```
    </div>
{% endif %}

<h1>Paste Details: </h1>
<p>
    <div>
        <label>ID</label>
        <span>{{ object.id }}</span>
    </div>
    <div>
        <label>Name</label>
        <span>{{ object.name }}</span>
    </div>
    <div>
        <label>Text</label>
        <textarea rows="10" cols="50" OnClick="this.select();" readonly="true">{{␣
↪object.text }}</textarea>
    </div>
    <div>
        <label>Created</label>
        <span>{{ object.created_on }}</span>
    </div>
    <div>
        <label>Modified</label>
        <span>{{ object.updated_on }}</span>
    </div>
</p>

<h2>Actions</h2>
    <ul>
        <li>
            <a href="{% url 'pastebin_paste_edit' object.id %}">Edit this paste</a>
        </li>
        <li>
            <a href="{% url 'pastebin_paste_delete' object.id %}">Delete this paste</
↪a>
        </li>
    </ul>

<a href="{% url 'pastebin_paste_list' %}">View All</a>
```

So we now have pages to create, update, delete and view all pastes.

Now, for better maintenance, we would like to delete all pastes that have not been updated in a day using an script. We will use django's custom management scripts for this.

3.7 Writing custom management scripts:

Just like other manage.py subcommands such as `migrations`, `shell`, `startapp` and `runserver`, we can have custom subcommands to help us maintain the app.

For our subcommand to be registered with manage.py, we need the following structure in our app:

```
.
|-- __init__.py
|-- management
```

```
|   |-- commands
|   |   `-- __init__.py
|   `-- __init__.py
|-- models.py
|-- tests.py
|-- urls.py
`-- views.py
```

All scripts inside `management/commands/` will be used as custom subcommands. Let's create `delete_old.py` subcommand:

```
import datetime

from django.core.management.base import BaseCommand

from pastebin.models import Paste

class Command(BaseCommand):
    help = """
            deletes pastes not updated in last 24 hrs

            Use this subcommand in a cron job
            to clear older pastes
           """

    def handle(self, **options):
        now = datetime.datetime.now()
        yesterday = now - datetime.timedelta(1)
        old_pastes = Paste.objects.filter(updated_on__lte=yesterday)
        old_pastes.delete()
```

Here:

- We subclass either of the `BaseCommand`, `LabelCommand` or `AppCommand` from `django.core.management.base`. `BaseCommand` suits our need because we dont need to pass any arguments to this subcommand.
- `handle` will be called when the script runs. This would be `handle` for other Command types.
- We have used the `lte` lookup on `updated_on` field to get all posts older than a day. Then we delete them using `delete` method on the queryset.

You can test if the subcommand works by doing:

```
python manage.py delete_old
```

Now we can configure this script to run daily using cronjob or something similar.

CHAPTER 4

Chapter 4. Building a Blog

4.1 Topics in this chapter:

So far, we have seen how to use django's admin interface, and generic views. In this chapter we will write custom views and blog entry admin page, store some user details in the session and use date based generic views for our blog archives.

4.1.1 Models with ForeignKeys:

To link two Models, we use foreign keys. Since we may have many comments for a blog post, we would like to have a relation from the comment model to the post model. This can be done by using a model field of the type 'ForeignKey'. It requires a string or class representing the Model to which the ForeignKey is to be created. We would also need ForeignKey to link each blog post to it's owner, since we want only the admin to be able to create a post.

As we shall see, django simplifies access to foreign keys by automatically creating a related object set on the linked Model.

```
class Post(models.Model):
    text = models.TextField()
    ...
    ...

class Comment(models.Model):
    text = models.CharField(max_length=100)
    post = models.ForeignKey(Post)
    ...
    ...
```

would relate each `Post` to multiple `Comment`

Now, to fetch all comments related to a Post, we can use

```
post = Post.objects.get(pk=1)
comments = post.comment_set.all()
```

So, post gets a <ForeignKeyModel>_set property which is actually the Manager for Comments related to that post.

4.1.2 ModelForms:

We have already seen how the admin app creates the form for our model by inspecting its fields. If we want to customize the autogenerated forms, we can use ModelForms. In this chapter we see how to use ModelForms to create and customize the forms for post and comment models. By convention, all custom forms should be in <app>/forms.py

The simplest way to use ModelForms would be:

```
class PostForm(ModelForm):
    class Meta:
        model = Post
```

The autogenerated fields can be overridden by explicitly specifying the field. To change the html widget and label used by text field of Post model, one would do:

```
class PostForm(ModelForm):
    text = forms.CharField(widget=forms.TextArea, label='Entry')
    class Meta:
        model = Post
```

4.1.3 Custom views with forms and decorators:

We will write our custom views in this chapter. We have already introduced views in the previous chapter, so we will see how to use forms in our views and how to limit access to certain views using decorators.

```
form = PostForm(request.POST or None)
```

Here we are handling GET and POST in the same view. If this is a GET request, the form would be empty else it would be filled with the POST contents.

```
form.is_valid
```

validates the form and returns True or False

We will use these two together to save a valid form or display empty form.

To restrict views to a condition, we can use the user_passes_test decorator from contrib.auth. The decorator takes a callable which should perform the test on the user argument and return True or False. The view is called only when the user passes the test. If the user is not logged in or does not pass the test, it redirects to LOGIN_URL of settings. By default this is /accounts/login and we will handle this url from urls.py

Some other useful decorators are:

- django.contrib.admin.views.decorators import staff_member_required

  Restricts view to staff members only.

- django.contrib.auth.decorators.login_required

  Restricts view to logged in users only

4.2 Our blog app:

Let's list out the features we would want to see in our blog:

- Create/Edit blog post (restricted to admin)
- View blog post (public)
- Comment on a blog post (anonymous)
- Store anonymous user details in session
- Show month based blog archives
- Generate RSS feeds

We have two models here: `Post` and `Comment`. The data we would like store are:

For Post:

- Title
- Text Content
- Slug
- Created Date
- Author

For Comment:

- Name
- Website
- Email
- Text Content
- Post related to this comment
- Created Date

Note: Since we want anonymous to be able to comment on a post, we are not relating the comment poster to a registered user.

We want the `author` field of the post to be mapped to a registered user and the `post` field to be mapped to a valid `Post`. As we shall see, we will ForeignKeys to the appropriate models to manage these.

4.2.1 Models:

We have already seen how to create and integrate an app into our project, so I will start with the models

```
from django.db import models
from django.template.defaultfilters import slugify

from django.contrib.auth.models import User

class Post(models.Model):
    title = models.CharField(max_length=100)
    slug = models.SlugField(unique=True)
```

```
    text = models.TextField()
    created_on = models.DateTimeField(auto_now_add=True)
    author = models.ForeignKey(User, on_delete=models.CASCADE)

    def __unicode__(self):
        return self.title

    @models.permalink
    def get_absolute_url(self):
        return ('blog_post_detail', (),
                {
                    'slug' :self.slug,
                })

    def save(self, *args, **kwargs):
        if not self.slug:
            self.slug = slugify(self.title)
        super(Post, self).save(*args, **kwargs)

class Comment(models.Model):
    name = models.CharField(max_length=42)
    email = models.EmailField(max_length=75)
    website = models.URLField(max_length=200, null=True, blank=True)
    text = models.TextField()
    post = models.ForeignKey(Post, on_delete=models.CASCADE)
    created_on = models.DateTimeField(auto_now_add=True)

    def __unicode__(self):
        return self.text
```

Quite a few new things here, let's analyze them:

- slug field - it is used for storing slugs (e.g. this-is-a-slug). SEO or something.
- We will be using slugs in the url to fetch a blog post, so this must be unique.
- `slugify` is a helper function to get slug from a string. We won't need to get the slug from the form, we will generate it ourself using `slugify`
- To autogenerate the slug, we override the `model` save method, whose signature is `save(self, *args, **kwargs)` We set `self.slug` to the slug generated by `slugify` and call the parent `save` method.
- This ensures that every time a model is saved, it will have a slug field.
- The `get_absolute_url` of the `Post` model points to the `blog_post_detail` which takes a `slug` parameter. This is the `Post` detail view, and it fetches the post based on the `slug`. We will soon see how this is implemented.
- `model.ForeignKey` is a ForeignKey field which can be used to link this model to any other model. Here we want to link the `author` field to a `User`, which is django's model for a user. It comes from `django.contrib.auth` app, which is another useful package shipped with django.
- Similarly to link a `Comment` to a `Post` we have a ForeignKey from in the `post` field of the comment.
- We won't need the `author` field from the `Post` form either, but we will fill it up in the view, where we have access to the logged in user details

4.2.2 Views:

The views we would need are:

- Admin should be able to login
- Add/Edit a post - restricted to admin
- View a blog post
- Comment on a blog post

We need to customize our forms to only display fields which need user input, because we will take care of the rest. For example, we have already seen how to autofill slug field. Next, we would like to autofill `post` for `Comment` and `author` for `Post` in the view. Heres our `blog/forms.py`

```
from django import forms

from .models import Post, Comment

class PostForm(forms.ModelForm):
    class Meta:
        model = Post
        exclude = ['author', 'slug']

class CommentForm(forms.ModelForm):
    class Meta:
        model = Comment
        exclude = ['post']
```

For login, we will use `django.contrib.auth.views.login` view which is included in the `contrib.auth` app. It expects a `registration/login.html` which we will steal from `django/contrib/admin/templates/admin/login.html`. We will include the login url in the project urls.

```
from django.contrib import admin
from django.urls import path, include
from django.contrib.auth.views import login

urlpatterns = [
    path('accounts/login/', login),
    path('admin/', admin.site.urls),
    path('pastebin/', include('pastebin.urls')),
    path('blog/', include('blog.urls')),
    ]
```

In `templates/registration/login.html`, copy contents from `django/contrib/admin/templates/admin/login.html`

For the others, we will write custom views in `blog/views.py`.

```
from django.contrib.auth.decorators import user_passes_test
from django.shortcuts import redirect, render_to_response, get_object_or_404, render

from .models import Post
from .forms import PostForm, CommentForm

@user_passes_test(lambda u: u.is_superuser)
def add_post(request):
    form = PostForm(request.POST or None)
    if form.is_valid():
```

```
        post = form.save(commit=False)
        post.author = request.user
        post.save()
        return redirect(post)
    return render(request, 'blog/add_post.html',{ 'form': form })

def view_post(request, slug):
    post = get_object_or_404(Post, slug=slug)
    form = CommentForm(request.POST or None)
    if form.is_valid():
        comment = form.save(commit=False)
        comment.post = post
        comment.save()
        return redirect(request.path)
    return render(request, 'blog/blog_post.html',{'post': post,'form': form,})
```

Note:

- The `user_passes_test` decorator whether the user is admin or not. If not, it will redirect the user to login page.
- We are using the `ModelForms` defined in `forms.py` to autogenerate forms from our Models.
- `ModelForm` includes a `save` method (just like a `Models` save method) which saves the model data to the database.
- `commit=False` on a form save gives us the temporary `Model` object so that we can modify it and save permanently. Here, we have used it to autofill the `author` of `Post` and `post` of `Comment`
- `redirect` is a shortcut that redirects using `HttpResponseRedirect` to another url or a model's `get_absolute_url` property.

4.2.3 Templates:

The corresponding templates for these views would look like:

`blog/templates/blog/add_post.html`:

```
<h2>Hello {{ user.username }}</h2>
<br />
<h2>Add new post</h2>
<form action="" method="POST">
    {% csrf_token %}
    <table>
        {{ form.as_table }}
    </table>
    <input type="submit" name="add" value="Add" />
</form>
```

`blog/templates/blog/blog_post.html`:

```
<h2>{{ post.title }}</h2>
<div class="content">
    <p>
        {{ post.text }}
    </p>
    <span>
        Written by {{ post.author }}  on {{ post.created_on }}
```

```
    </span>
</div>

{% if post.comment_set.all %}
<h2>Comments</h2>
<div class="comments">
    {% for comment in post.comment_set.all %}
        <span>
            <a href="{{ comment.website }}">{{ comment.name }}</a> said on {{ comment.
↪created_on }}
        </span>
        <p>
            {{ comment.text }}
        </p>
    {% endfor %}
</div>
{% endif %}

<br />

<h2>Add Comment</h2>

<form action="" method="POST">
    {% csrf_token %}
    <table>
        {{ form.as_table }}
    </table>
    <input type="submit" name="submit" value="Submit" />
</form>
```

Note: Since `Comment` has a ForeignKey to `Post`, each `Post` object automatically gets a `comment_set` property which provides an interface to that particular `Post`'s comments.

4.2.4 Sessions:

So far we have most of the blog actions covered. Next, let's look into sessions:

Suppose we want to store the commenter's details in the session so that he/she does not have to fill them again.

```
from django.contrib.auth.decorators import user_passes_test
from django.shortcuts import redirect, render_to_response, get_object_or_404, render

from .models import Post
from .forms import PostForm, CommentForm

@user_passes_test(lambda u: u.is_superuser)
def add_post(request):
    form = PostForm(request.POST or None)
    if form.is_valid():
        post = form.save(commit=False)
        post.author = request.user
        post.save()
        return redirect(post)
    return render(request, 'blog/add_post.html',{ 'form': form })
```

```
def view_post(request, slug):
    post = get_object_or_404(Post, slug=slug)
    form = CommentForm(request.POST or None)
    if form.is_valid():
        comment = form.save(commit=False)
        comment.post = post
        comment.save()
        request.session["name"] = comment.name
        request.session["email"] = comment.email
        request.session["website"] = comment.website
        return redirect(request.path)
    form.initial['name'] = request.session.get('name')
    form.initial['email'] = request.session.get('email')
    form.initial['website'] = request.session.get('website')
    return render(request, 'blog/blog_post.html',{'post': post,'form': form,})
```

Note that the `form.initial` attribute is a `dict` that holds initial data of the form. A session lasts until the user logs out or clears the cookies (e.g. by closing the browser). django identifies the session using `sessionid` cookie.

The default session backend is `django.contrib.sessions.backends.db` i.e. database backend, but it can be configured to `file` or `cache` backend as well.

4.3 Date based generic views:

Note: reference: https://docs.djangoproject.com/en/2.0/ref/class-based-views/generic-date-based/

Add code of date based views to the `blog/views.py` so as to make it work.

```
from django.contrib.auth.decorators import user_passes_test
from django.shortcuts import redirect, render_to_response, get_object_or_404, render
from django.views.generic.dates import MonthArchiveView, WeekArchiveView

from .models import Post
from .forms import PostForm, CommentForm

@user_passes_test(lambda u: u.is_superuser)
def add_post(request):
    form = PostForm(request.POST or None)
    if form.is_valid():
        post = form.save(commit=False)
        post.author = request.user
        post.save()
        return redirect(post)
    return render(request, 'blog/add_post.html',{ 'form': form })

def view_post(request, slug):
    post = get_object_or_404(Post, slug=slug)
    form = CommentForm(request.POST or None)
    if form.is_valid():
        comment = form.save(commit=False)
        comment.post = post
        comment.save()
        request.session["name"] = comment.name
        request.session["email"] = comment.email
```

```
            request.session["website"] = comment.website
            return redirect(request.path)
    form.initial['name'] = request.session.get('name')
    form.initial['email'] = request.session.get('email')
    form.initial['website'] = request.session.get('website')
    return render(request, 'blog/blog_post.html',{'post': post,'form': form,})

class PostMonthArchiveView(MonthArchiveView):
    queryset = Post.objects.all()
    date_field = "created_on"
    allow_future = True

class PostWeekArchiveView(WeekArchiveView):
    queryset = Post.objects.all()
    date_field = "created_on"
    week_format = "%W"
    allow_future = True
```

We will use date based generic views to get weekly/monthly archives for our blog posts:

```
from django.urls import path, include
from .views import view_post, add_post, PostMonthArchiveView, PostWeekArchiveView


urlpatterns = [
    path('post/<str:slug>', view_post, name='blog_post_detail'),
    path('add/post', add_post, name='blog_add_post'),
    path('archive/<int:year>/month/<int:month>', PostMonthArchiveView.as_view(month_
↪format='%m'), name='blog_archive_month',),
    path('archive/<int:year>/week/<int:week>', PostWeekArchiveView.as_view(), name=
↪'blog_archive_week'),
        ]
```

`PostMonthArchiveView` generic class based views outputs to `post_archive_month.html` and `PostWeekArchiveView` to `post_archive_week.html`

```
<h2>Post archives for {{ month|date:"F" }}, {{ month|date:"Y" }}</h2>

<ul>
    {% for post in object_list %}
        <li>
        <a href="{% url 'blog_post_detail' post.slug %}">{{ post.title }}</a>
        </li>
    {% endfor %}
</ul>
```

```
<h2>Post archives for week {{ week|date:"W" }}, {{ week|date:"Y" }}</h2>

<ul>
    {% for post in object_list %}
        <li>
        <a href="{% url 'blog_post_detail' post.slug %}">{{ post.title }}</a>
        </li>
    {% endfor %}
</ul>
```

Now, blog archives should be accessible from `/blog/archive/2018/month/03` or `/blog/archive/2018/week/16`

CHAPTER 5

Chapter 5. Building a Wiki

5.1 A wiki application:

In this chapter, we will build a wiki from scratch. Basic functionality includes:

- Article Management (CRUD) with ReST support
- Audit trail for articles
- Revision history

5.2 Article Management:

This is similar to our last app (blog) in many ways. Significant changes would be:

- Allow administrator to add/edit an article.
- Allow ReST input instead of just plain text.
- Keep track of all edit sessions related to an article.

To demonstrate custom model managers, we would like to show only 'published' articles on the index page.

Let's write down the models:

```
from django.db import models
from django.contrib.auth.models import User
from django.template.defaultfilters import slugify


class PublishedArticlesManager(models.Manager):
    def get_query_set(self):
        return super(PublishedArticlesManager, self).get_query_set().filter(is_
↪published=True)
```

```python
class Article(models.Model):
    """Represents a wiki article"""

    title = models.CharField(max_length=100)
    slug = models.SlugField(max_length=50, unique=True)
    text = models.TextField(help_text="Formatted using ReST")
    author = models.ForeignKey(User, on_delete=models.CASCADE)
    is_published = models.BooleanField(default=False, verbose_name="Publish?")
    created_on = models.DateTimeField(auto_now_add=True)
    objects = models.Manager()
    published = PublishedArticlesManager()

    def __unicode__(self):
        return self.title

    def save(self, *args, **kwargs):
        if not self.slug:
            self.slug = slugify(self.title)
        super(Article, self).save(*args, **kwargs)

    @models.permalink
    def get_absolute_url(self):
        return ('wiki_article_detail', (), {'slug': self.slug})


class Edit(models.Model):
    """Stores an edit session"""

    article = models.ForeignKey(Article, on_delete=models.CASCADE)
    editor = models.ForeignKey(User, on_delete=models.CASCADE)
    edited_on = models.DateTimeField(auto_now_add=True)
    summary = models.CharField(max_length=100)

    class Meta:
        ordering = ['-edited_on']

    def __unicode__(self):
        return "%s - %s - %s" % (self.summary, self.editor, self.edited_on)

    @models.permalink
    def get_absolute_url(self):
        return ('wiki_edit_detail', self.id)
```

Most of the code should be familiar, some things that are new:

- The Article model will hold all articles, but only those with `is_published` set to `True` will be displayed on the front page.
- We have a defined a custom model manager called `PublishedArticlesManager` which is a queryset that only returns the published articles.
- Non-published articles would be used only for editing. So, we retain the default model manager by setting `objects` to `models.Manager`
- Now, to fetch all articles, one would use `Articles.objects.all`, while `Artilces.published.all` would return only published articles.
- A custom manager should subclass `models.Manager` and define the custom `get_query_set` property.

- The `Edit` class would hold an edit session by a registered user on an article.
- We see the use of `verbose_name` and `help_text` keyword arguments. By default, django will replace _ with spaces and Capitalize the field name for the label. This can be overridden using `verbose_name` argument. `help_text` will be displayed below a field in the rendered `ModelForm`
- The `ordering` attribute of meta class for `Edit` defines the default ordering in which `edits` will be returned. This can also be done using `order_by` in the queryset.

Now, we will need urls similar to our previous app, plus we would need a url to see the article history.

```
from django.urls import path, include
from .views import add_article, edit_article, article_history, ArticleList,
↪ArticleDetail

urlpatterns = [
    path('', ArticleList.as_view(), name='wiki_article_index'),
    path('article/<str:slug>',ArticleDetail.as_view(),name='wiki_article_detail'),
    path('history/<str:slug>', article_history, name='wiki_article_history'),
    path('add/article', add_article, name='wiki_article_add'),
    path('edit/article/<str:slug>', edit_article, name='wiki_article_edit'),
]
```

Note that:

- We will use the `DetailView` generic views for the article index page and detail page.
- Similarly, it would be better to write down custom views for edit article and article history pages.

Here are the forms we will need:

```
from django import forms

from .models import Article, Edit

class ArticleForm(forms.ModelForm):
    class Meta:
        model = Article
        exclude = ['author', 'slug']


class EditForm(forms.ModelForm):
    class Meta:
        model = Edit
        fields = ['summary']
```

Here:

- We are excluding `author` and `slug` which will be autofilled.
- We are inluding the `summary` field in `Edit` model only. The other fields (`article`, `editor`, `edited_on`) will be autofilled.

In our custom views:

```
from django.contrib.auth.decorators import login_required
from django.contrib import messages
from django.shortcuts import redirect, render, get_object_or_404
from django.views.generic.list import ListView
from django.views.generic.detail import DetailView
from django.http import HttpResponse
```

```
from .models import Article, Edit
from .forms import ArticleForm, EditForm

@login_required
def add_article(request):
    form = ArticleForm(request.POST or None)
    if form.is_valid():
        article = form.save(commit=False)
        article.author = request.user
        article.save()
        msg = "Article saved successfully"
        messages.success(request, msg, fail_silently=True)
        return redirect(article)
    return render(request, 'wiki/article_form.html', { 'form': form })

@login_required
def edit_article(request, slug):
    article = get_object_or_404(Article, slug=slug)
    form = ArticleForm(request.POST or None, instance=article)
    edit_form = EditForm(request.POST or None)
    if form.is_valid():
        article = form.save()
        if edit_form.is_valid():
            edit = edit_form.save(commit=False)
            edit.article = article
            edit.editor = request.user
            edit.save()
            msg = "Article updated successfully"
            messages.success(request, msg, fail_silently=True)
            return redirect(article)
    return render(request, 'wiki/article_form.html',{'form': form, 'edit_form': edit_
↪form, 'article': article})

def article_history(request, slug):
    article = get_object_or_404(Article, slug=slug)
    queryset = Edit.objects.filter(article__slug=slug)
    return  render(request, 'wiki/edit_list.html',{'article': article, 'queryset':␣
↪queryset})

class ArticleList(ListView):
    template_name = "wiki/article_list.html"
    def get_queryset(self):
        return Article.objects.all()

class ArticleDetail(DetailView):
    model = Article
    template_name = "wiki/article_detail.html"
```

- We are using the `login_required` decorator to only allow logged-in users to add/edit articles in our case logged in administrator.
- `get_object_or_404` is a shortcut method which `gets` an object based on some criteria. While the `get` method throws an `DoesNotExist` when no match is found, this method automatically issues a `404 Not Found` response. This is useful when getting an object based on url parameters (`slug`, `id` etc.)
- `redirect`, as we have seen, would issue a `HttpResponseRedirect` on the `article's`

get_absolute_url property.

- edit_article includes two forms, one for the Article model and the other for the Edit model. We save both the forms one by one.
- Passing instance to the form will populate existing data in the fields.
- As planned, the author field of article and editor, article fields of Article and Edit respectively, are filled up before commiting save.
- article_history view first checks if an article with the given slug exists. If yes, it forwards the request to the object_list generic view. We also pass the article .
- Note the filter on the Edit model's queryset and the lookup on the related Article's slug.

To display all the articles on the index page:

wiki/templates/wiki/article_list.html:

```
{% if object_list %}

<h2>Recent Articles</h2>

<ul>
    {% for article in object_list %}
    <li>
        <a href="{% url 'wiki_article_detail' article.slug %}">{{ article.title }}</a>
    </li>
    {% endfor %}
</ul>

{% else %}
<h2>No articles have been published yet.</h2>
{% endif %}

<a href="{% url 'wiki_article_add' %}">Create new article</a>
```

We will include links to edit and view history in the article detail page:

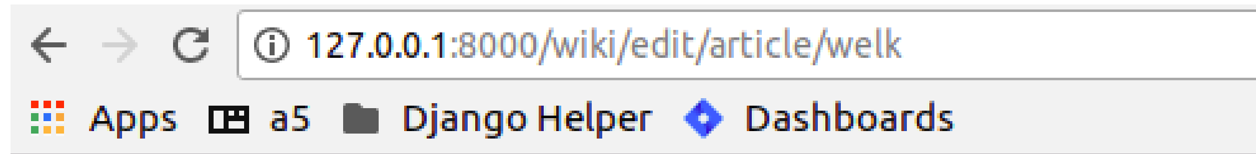
wiki/templates/wiki/article_detail.html:

```
{% if messages %}
    <div class="messages">
    <ul>
    {% for message in messages %}
        <li class="{{ message.tag }}">
            {{ message }}
        </li>
        {% endfor %}
    </ul>
    </div>
{% endif %}

{% if not object.is_published %}
    <label>Note: This article has not been published yet</label>
{% endif %}

<h2>{{ object.title }}</h2>

<p>
{{ object.text|restructuredtext }}
</p>

<h3>Actions<h3>
<ul>
    <li>
        <a href="{% url 'wiki_article_edit' object.slug %}">Edit this article</a>
    </li>
    <li>
        <a href="{% url 'wiki_article_history' object.slug %}">View article history</
↪a>
    </li>
</ul>

<a href="{% url wiki_article_index %}">See All</a>
```

Here's the form that would be used to create/edit an article:

wiki/templates/wiki/article_form.html

```
{% if article %}
    <h1>Edit article {{ article }}</h1>
{% else %}
    <h1>Create new article</h1>
{% endif %}

<form action="" method="POST">
    {% csrf_token %}
    <table>
        {{ form.as_table }}
        {{ edit_form.as_table }}
    </table>
    <input type="submit" name="submit" value="Submit">
</form>
```



Edit article Article object (3)

Title: Third Article

Text: I have become a writer

Formatted using ReST

Publish? ☑

Summary:

Submit

Note that the same form is used for add article and edit article pages. We pass the `article` context variable from edit page, so we can use it to identify if this is an add or edit page. We also render the `edit_form` passed from edit page. Rendering an undefined variable does not throw any error in the template, so this works fine in the add page.

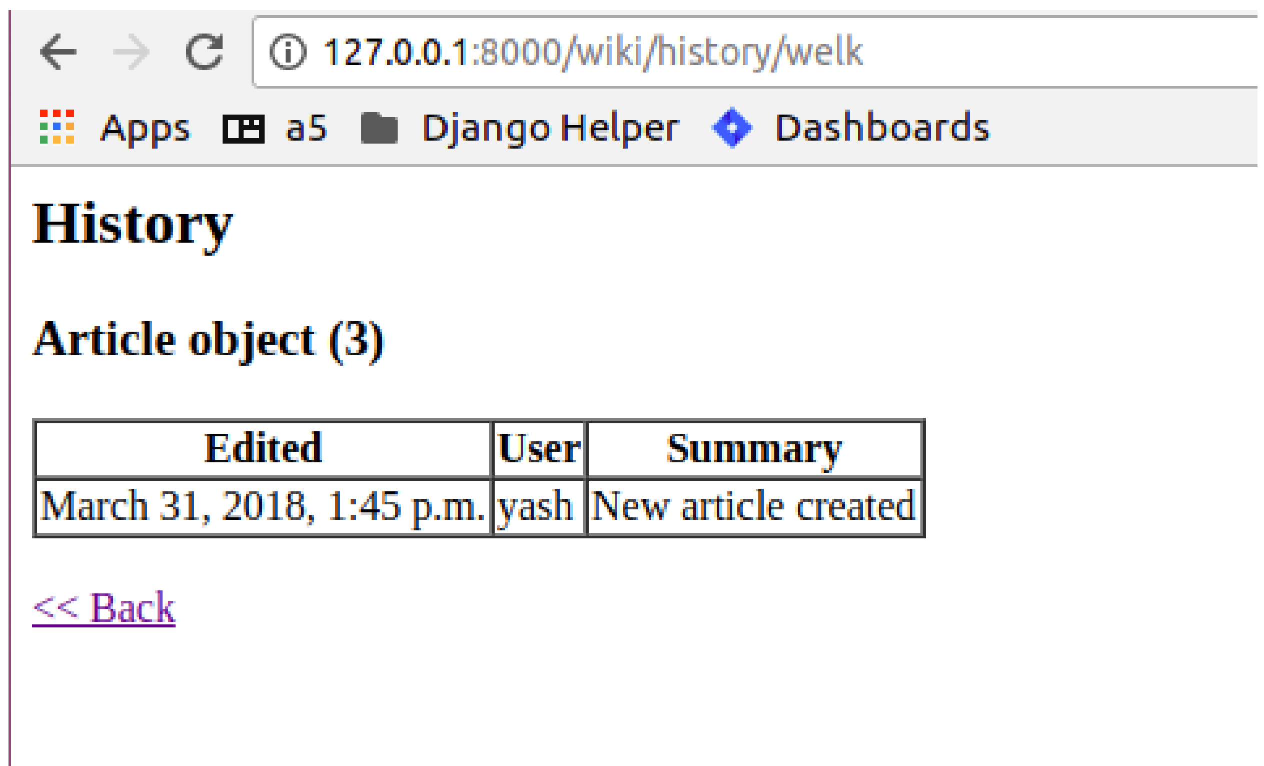
The article history template:

`wiki/templates/wiki/edit_list.html`

```
<h2>History</h2>

<h3>{{ article }}</h3>

<table border="1" cellspacing="0">
    <thead>
        <th>Edited</th>
        <th>User</th>
        <th>Summary</th>
    </thead>
    <tbody>
        {% for edit in object_list %}
        <tr>
            <td>{{ edit.edited_on }}</td>
```

```
            <td>{{ edit.editor }}</td>
            <td>{{ edit.summary }}</td>
        </tr>
        {% endfor %}
        <tr>
            <td>{{ article.created_on }}</td>
            <td>{{ article.author }}</td>
            <td>New article created</td>
        </tr>
    </tbody>
</table>

<br />
<a href="{% url 'wiki_article_detail' article.slug %}"><< Back</a>
```

Displays a table with the history.



CHAPTER 6

Chapter 6. Building a Quora like site

6.1 Topics in this chapter:

We have covered basics in last few chapters, like Generic Views, template languages, ORM, interaction with django admin etc. Now in this chapter we will be creating `Custom User`, who will be able to access the Qusetion and answers in the Quora like app.

6.2 Quora like Application:

We have checked Quora for checking many qusetions in our past. Qusetions may be both technical or non technical. In this tutorial we will be creating a Qura like application but not exactly the Quora.

6.3 Application Includes:

- Registering custom users (Substitute of django's admin user)
- Custom Users Login/Logout Functionality
- Questions asked by users.
- Answered Questions by Users
- Dashboard user specific.

6.4 Django features to learn in this chapter:

- Class Based Views
- Basics of Django Testing

- Customising Users

Lets Begin

We will be creating a project from scratch, lets brush-up !!!

```
$ django-admin startproject quora
$ cd quora
$ python manage.py startapp core // Custom User Trick
```

Note: Never use the built-in Django User model directly, even if the built-in Django User implementation fulfill all the requirements of your application. *Once you are done with customising your Custom user then only do makemigrations & migrate*

6.5 Make custom user:

- Step 1): Goto `core/models.py` and add this

```
from django.db import models
from django.contrib.auth.models import AbstractUser


class User(AbstractUser):
    pass
```

- Step 2): In your settings.py file add a line just after ALLOWED_HOSTS = [].

```
AUTH_USER_MODEL = 'core.User' // It can be kept anywhere in the file but good to keep␣
↪just after Allowed hosts.
```

Note: Don't forget to add your newly created app to installed apps in `settings.py` file.

```
INSTALLED_APPS = [
    'django.contrib.admin',
    . . .
    . . .
    'core',
]
```

Congratulations you have customised your Django user Model. now lets migrate changes.

```
$ python manage.py makemigrations
$ python manage.py migrate
$ python manage.py createsuperuser // follow the instructions
```

We will now create the Custom user's entry in Django Admin, as by the above process we won't be able to see its entry in Django admin's dashboard. So , in `core/admin.py` we should add :

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

admin.site.register(User, UserAdmin)
```

6.6 Class Based Views

Class-based views provide an alternative way to implement views as Python objects instead of functions. They do not replace function-based views, but have certain differences and advantages when compared to function-based views:

- Organization of code related to specific HTTP methods (GET, POST, etc.) can be addressed by separate methods instead of conditional branching.
- Object oriented techniques such as mixins (multiple inheritance) can be used to factor code into reusable components.

Example

```
from django.http import HttpResponse
// Function Based View.
def my_view(request):
    if request.method == 'GET':
        # <view logic>
        return HttpResponse('result')

from django.http import HttpResponse
from django.views import View
// Class Based View
class MyView(View):
    def get(self, request):
        # <view logic>
        return HttpResponse('result')
```

6.7 Register Custom User

Now that we are aware of Class Based View let's implement **user registration using the same**.

Add the below code to `core/forms.py`

```
from django import forms
from .models import User

class RegisterForm(forms.ModelForm):
    password = forms.CharField(widget=forms.PasswordInput())

    class Meta:
        model = User
        fields = ['email', 'first_name', 'last_name', 'password', 'username']
```

We will now use the above forms in our views, add the below code to `core/views.py`.

```
from django.shortcuts import render
from .forms import RegisterForm
from django.contrib.auth import login
from django.contrib.auth.hashers import make_password


class RegisterView(FormView):
```

```
    def get(self, request):
        content = {}
        content['form'] = RegisterForm
        return render(request, 'register.html', content)

    def post(self, request):
        content = {}
        form = RegisterForm(request.POST, request.FILES or None)
        if form.is_valid():
            user = form.save(commit=False)
            user.password = make_password(form.cleaned_data['password'])
            user.save()
            login(request, user)
            return redirect(reverse('dashboard-view'))
        content['form'] = form
        template = 'register.html'
        return render(request, template, content)
```

There are few thing which we have imported like login(), make_password() etc, it will be good to know about them.

- To log a user in, from a view, use `login()`. It takes an HttpRequest object and a User object. login() saves the user's ID in the session, using Django's session framework.
- `make_password` creates a hashed password in the format used by this application. It takes one mandatory argument: the password in plain-text.
- we will talk about `dashboard-view` further in this tutorial. For now just relate it like, once you register yourself you will be redirected to the `dashbord-view`.

Its still not over we still have to make some modifications in settings.py , urls.py and adding of templates. If you have followed previous chapters you may try on your own. Still you can refer to content below.

Add below code to `core/urls.py` and `quora/urls.py` respectively.

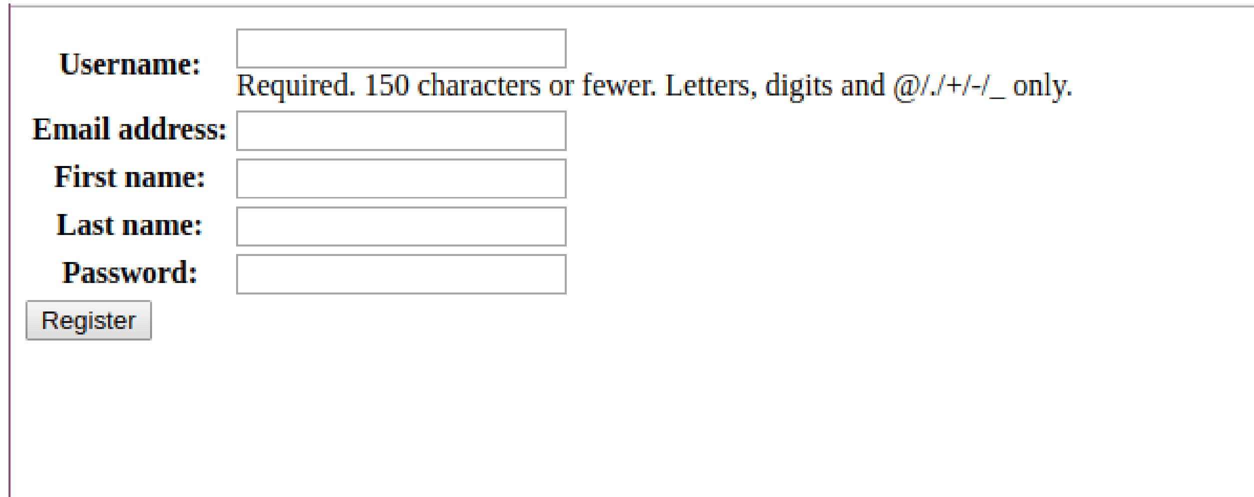
```
from django.urls import path
from .views import RegisterView

urlpatterns = [
    path('register/', RegisterView.as_view(), name='register-view'),
]

// quora/urls.py
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('core/', include('core.urls')),
]
```

Now we will add a new directory to our project as `project/templates` in our case `quora/templates`. And inside templates directory add a new file `templates/register.html` and add the below code.

```
<form action="" method="POST">
{% csrf_token %}
<table>
    {{ form.as_table }}
</table>
```

```
<input type="submit" name="register" value="Register" />
</form>
```

Runserver and go to `localhost:8000/core/register/`



What next? Take some time and think what will be the next thing to do. And come back to te chapter.

Yes, you are right , after registering the user we will redirect him to his dashboard and also create a way by which he/she could login and logout from the application. And the code for this can be found below.

core/views.py

```
from django.shortcuts import render
from django.shortcuts import HttpResponseRedirect, redirect
from django.urls import reverse
from django.views.generic.edit import FormView
from django.utils.decorators import method_decorator
from django.views.decorators.csrf import csrf_exempt
from django.contrib.auth import authenticate
from django.contrib.auth import login, logout
from django.contrib.auth.hashers import make_password
from .models import User
from questans.models import Questions, Answers, QuestionGroups
from .forms import LoginForm, RegisterForm


class DashboardView(FormView):

    def get(self, request):
        content = {}
        if request.user.is_authenticated:
            user = request.user
            user.backend = 'django.contrib.core.backends.ModelBackend'
            ques_obj = Questions.objects.filter(user=user)
            content['userdetail'] = user
            content['questions'] = ques_obj
            ans_obj = Answers.objects.filter(question=ques_obj[0])
            content['answers'] = ans_obj
            return render(request, 'dashboard.html', content)
        else:
            return redirect(reverse('login-view'))
```

```
class RegisterView(FormView):

    @method_decorator(csrf_exempt)
    def dispatch(self, request, *args, **kwargs):
        return super(RegisterView, self).dispatch(request, *args, **kwargs)

    def get(self, request):
        content = {}
        content['form'] = RegisterForm
        return render(request, 'register.html', content)

    def post(self, request):
        content = {}
        form = RegisterForm(request.POST, request.FILES or None)
        if form.is_valid():
            save_it = form.save(commit=False)
            save_it.password = make_password(form.cleaned_data['password'])
            save_it.save()
            login(request, save_it)
            return redirect(reverse('dashboard-view'))
        content['form'] = form
        template = 'register.html'
        return render(request, template, content)


class LoginView(FormView):

    content = {}
    content['form'] = LoginForm

    @method_decorator(csrf_exempt)
    def dispatch(self, request, *args, **kwargs):
        return super(LoginView, self).dispatch(request, *args, **kwargs)

    def get(self, request):
        content = {}
        if request.user.is_authenticated:
            return redirect(reverse('dashboard-view'))
        content['form'] = LoginForm
        return render(request, 'login.html', content)

    def post(self, request):
        content = {}
        email = request.POST['email']
        password = request.POST['password']
        try:
            users = User.objects.filter(email=email)
            user = authenticate(request, username=users.first().username,
↪password=password)
            login(request, user)
            return redirect(reverse('dashboard-view'))
        except Exception as e:
            content = {}
            content['form'] = LoginForm
            content['error'] = 'Unable to login with provided credentials' + e
            return render_to_response('login.html', content)
```

```python
class LogoutView(FormView):

    def get(self, request):
        logout(request)
        return HttpResponseRedirect('/')
```

core/urls.py

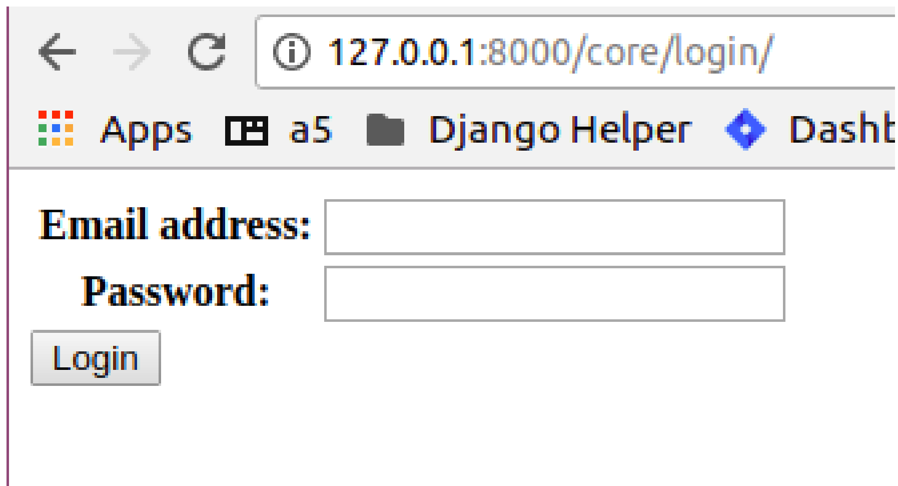
```python
from django.contrib import admin
from django.urls import path
from .views import LoginView, RegisterView, DashboardView, LogoutView

urlpatterns = [
    path('login/', LoginView.as_view(), name='login-view'),
    path('register/', RegisterView.as_view(), name='register-view'),
    path('dashboard/', DashboardView.as_view(), name='dashboard-view'),
    path('logout/', LogoutView.as_view(), name='logout-view'),
]
```

We have also configured two more templates i.e., `templates/login.html` and `templates/dashboard.html` with minimal functionality.

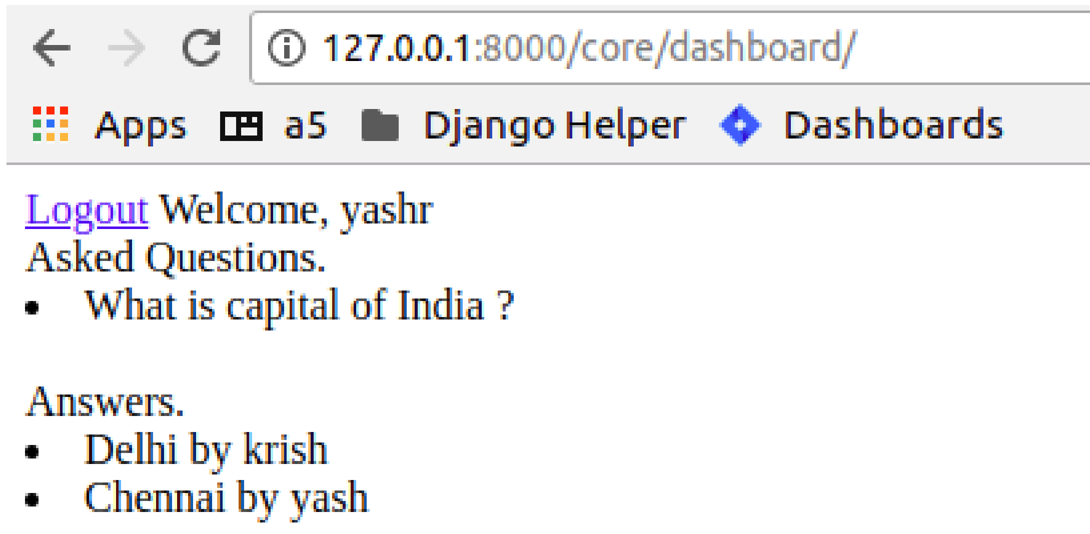
```
// login.html
<form action="" method="POST">
    {% csrf_token %}
    <table>
        {{ form.as_table }}
    </table>
    <input type="submit" name="login" value="Login" />
</form>
```

```
// dashboard.html
<a href='{% url "logout-view" %}'>Logout</a>
Welcome,
{{userdetail.username}}
<br/>
Asked Questions.
<br/>
{% for question in questions %}
<li>{{question.title}}</li>
{% endfor %}
<br/>
Answers.
{% for answer in answers %}
<li>{{answer.answer_text}} by {{answer.user.username}} </li>
{% endfor %}
```

So, the login page looks like



and dashboard is like



Wait, are we missing something ? Yes, till now we haven't discussed about the adding Questions and Answers in the Quora app, which is the foundation for the application.

Create an app in django let's name it as `questans` and add the below code to its models.

```
$ python manage.py startapp questans
```

Now in `questans/models.py` lets add the below code.

```
from django.db import models
from django.conf import settings
from django.utils.text import slugify

class Questions(models.Model):
    user = models.ForeignKey(settings.AUTH_USER_MODEL, on_delete=models.CASCADE,␣
↪null=True, blank=True)
    title = models.TextField()
```

```
    group = models.ForeignKey('QuestionGroups', on_delete=models.CASCADE, null=True,␣
↪blank=True)
    created_on = models.DateTimeField(auto_now=True)
    updated_on = models.DateTimeField(auto_now_add=True)
    slug = models.SlugField()

    def save(self, *args, **kwargs):
        self.slug = slugify(self.title)
        super(Questions, self).save(*args, **kwargs)

    def __unicode__(self):
        return self.title


class Answers(models.Model):
    user = models.ForeignKey(settings.AUTH_USER_MODEL, on_delete=models.CASCADE)
    question = models.ForeignKey(Questions, on_delete=models.CASCADE)
    answer_text = models.TextField()
    is_anonymous = models.BooleanField(default=False)

    def __unicode__(self):
        return self.id


class QuestionGroups(models.Model):
    name = models.CharField(max_length=100)

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

Things to note in above code

- In Question models we have created a `slug` field to make it more readable.
- We have also given flexibility for the user to save its answer anonymously.
- QuestionGroup is kept to differentiate question of different topics.

For now we can add questions and answers in the Quora app using its admin and make it more interactive in next part `Quora with bootstrap`. Add the below code to `questans/admin.py`.:align:

```
from django.contrib import admin
from .models import Questions, Answers, QuestionGroups


class AnswerInline(admin.TabularInline):
    model = Answers


class QuestionsAdmin(admin.ModelAdmin):

    inlines = [AnswerInline]
    class Meta:
        model = Questions


class QuestionGroupsAdmin(admin.ModelAdmin):

    class Meta:
```

```
        QuestionGroups

admin.site.register(Questions, QuestionsAdmin)
admin.site.register(QuestionGroups, QuestionGroupsAdmin)
```

Now our MVP of quora application is ready. We will discuss about Django's Testing Framework.

6.8 Basics of Django Testing:

Django's unit tests use a Python standard library module: unittest. This module defines tests using a class-based approach

We have described very basic example for testing our appication. Add the below code to `core/tests.py`

```
from django.test import TestCase
from .models import User

class LogInTest(TestCase):
    def setUp(self):
        self.credentials = {
            'username': "demo",
            'email': 'demo@gmail.com',
            'password': 'secret',
            'first_name': 'demo',
            'last_name': 'user',
        }
        User.objects.create_user(**self.credentials)

    def test_register(self):
        response = self.client.post('/core/register/', self.credentials, follow=True)
        self.assertEqual(response.status_code, 200)

    def test_user(self):
        user = User.objects.get(username="demo")
        self.assertEqual(user.username, "demo")
```

Run tests using the below command.

```
$ python manage.py test
```

CHAPTER 7

Chapter 7. Building a Project management application

(Topics introduced: File uploads, Integrating with Amazon S3, Complex RSS feeds-builds on chapter 4, Generating graphics using PIL. Sending Email with Django. Generating PDFs for pages. Exporting Data.)

(This and the next chapter would have larger amount of code than the previous chapters. These chapters would be a rehash of previous chapters, and would show how all these concepts work together.)

Diving in. [Code listing]

File uploads. (We allows users to upload files in this app.) Using the Django file widget to upload files. The problem with large files. Using S3, as a file store. Restricting access to S3 files, using Django authentication.

Advanced RSS feeds. Generating RSS feeds per project. Password protecting RSS feeds.

Using PIL. (We generate charts for the project, so we use PIL) Using PIL to generate charts for the project.

Sending email. The logs for the project are sent to the user via mail.

Genrating PDFs. The reports for the project are available as PDF. This is done using HTML2PDF library.

Exporting PDF. The data for a project is accessible in CSV format. Here we show exporting of a data on a per project, or a more granular level.

CHAPTER 8

Chapter 8. Building a Social news Site

(Topics introduced: This chapter uses techniques learnt in previous chapters, and introduce Caching and Testing.)

Diving in. [Code listing]

Introduce caching. The various caching backends. Page level caching. More granular caching.

Introduce testing for Django. Testing Django models using doctests, and unittests. Testing Django views.

Walking though the code. In this chapter, we walk through the code, to see how all these fit together.

Performance tuning the code. Logging the queries used, through a middleware. When select related makes sense. Profiling Django applications.