# DJANGO NOTES

## Anaconda

Anaconda Distribution 5 is a free, easy-to-install package manager, environment manager and Python distribution with a collection of 1,000+ open source packages with free community support. Anaconda is platform-agnostic, so you can use it whether you are on Windows, macOS or Linux.

## Conda

Conda is an open source package management system and environment management system that runs on Windows, macOS and Linux. Conda quickly installs, runs and updates packages and their dependencies. Conda easily creates, saves, loads and switches between environments on your local computer. It was created for Python programs, but it can package and distribute software for any language.
Conda as a package manager helps you find and install packages. If you need a package that requires a different version of Python, you do not need to switch to a different environment manager, because conda is also an environment manager. With just a few commands, you can set up a totally separate environment to run that different version of Python, while continuing to run your usual version of Python in your normal environment.
In its default configuration, conda can install and manage the thousand packages at repo.continuum.io that are built, reviewed and maintained by Anaconda.

We'll see how to create development environments and how to install Django using the Anaconda distribution. "conda" works as an environment and package manager. Regarding environments, its functioning is similar to "virtualenv". In this way, you can use it to create environments, where projects will have their own isolated configurations and packages installed, not messing with other projects setup. This is considered a good practice, because when you are working with more than one project, you can quickly start to have version conflicts between packages versions and Python's version itself.

| | |
|---|---|
| **To create a new environment:** | conda create --name test_environment python=2.7 |
| **To activate an environment:** | activate test_environment |
| **To install django:** | pip install django==1.6 |
| **To deactivate the environment:** | deactivate |
| **To remove the environment:** | conda remove --name test_environment --all |

# Django

Django is a high-level Python web framework that encourages rapid development and clean, pragmatic design. Django makes it easier to build better web apps quickly and with less code.

Django comes with the following design philosophies −

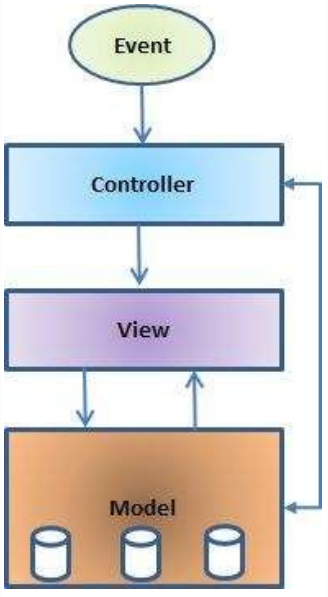
- **Loosely Coupled** − Django aims to make each element of its stack independent of the others.
- **Less Coding** − Less code so in turn a quick development.
- **Don't Repeat Yourself (DRY)** − Everything should be developed only in exactly one place instead of repeating it again and again.
- **Fast Development** − Django's philosophy is to do all it can to facilitate hyper-fast development.
- **Clean Design** − Django strictly maintains a clean design throughout its own code and makes it easy to follow best web-development practices.

**Advantages**-

- Object-Relational Mapping (ORM) Support [MySQL, Oracle, Postgres etc]
- Multilingual support
- Framework support [Ajax, RSS, Caching etc}
- Administration GUI
- Development Environment [lightweight web server for end to end app dev & testing]

Django supports MVC (Model-View-Controller) pattern, which is a software design pattern for developing web apps. MVC Pattern is made up of 3 parts-

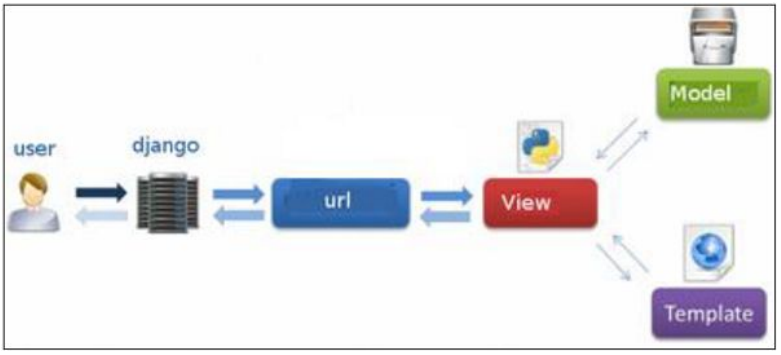
- **Model** - The lowest level, responsible for maintaining data
- **View** - Responsible for displaying all or a portion of data to the user
- **Controller** - Software code which controls interactions betweem the model and view.



# Django MVC - MVT Pattern

The Model-View-Template (MVT) is slightly different from MVC. In fact the main difference between the two patterns is that Django itself takes care of the Controller part (Software Code that controls the interactions between the Model and View), leaving us with the template. The template is a HTML file mixed with Django Template Language (DTL).

The following diagram illustrates how each of the components of the MVT pattern interacts with each other to serve a user request –




The developer provides the Model, the view and the template then just maps it to a URL and Django does the magic to serve it to the user.

## Create a Project

django-admin.py is a script that will create the directories and files for you.

- python django-admin.py startproject *projectName*

*Directory Structure-*

**mySite**/

Manage.py #helps with site management- starting web server, sync db etc

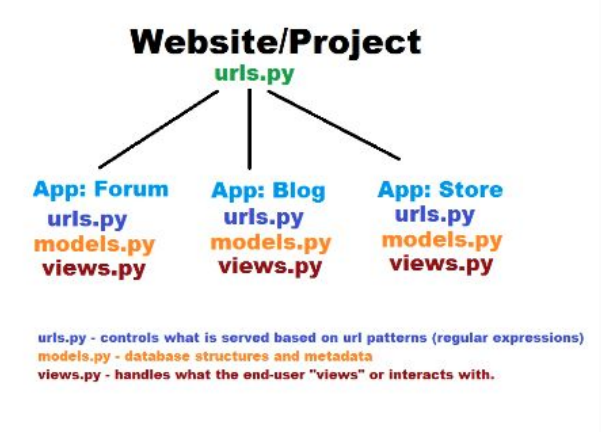
**mySite**/

__init.py__

settings.py #project settings - website configurations etc

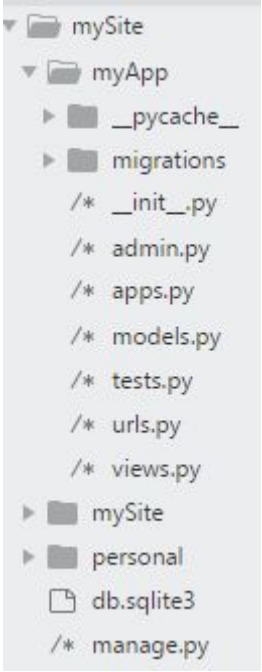
urls.py #list of patterns used by urlresolver, all links of project

wsgi.py #to deploy project over WSGI



## Create a new app for the project

python manage.py startapp myApp




Step 1: Modify the myApp/views.py to add the HTML response-

```
from django.shortcuts import render
from django.http import HttpResponse

def index(request):
    return HttpResponse("<h2>HEY!</h2>")
```

Step 2: Create myApp/urls.py file and add the view to the list of urlpatterns

```
from django.conf.urls import url
from . import views

urlpatterns = [
    url(r'^$', views.index, name='index'),
]
```

Step 3: Edit the mySite/urls file to redirect myApp requests to the corresponding myApp/urls.py file

```
from django.conf.urls import url,include
from django.contrib import admin

urlpatterns = [
    url(r'^admin/', admin.site.urls),
    url(r'^myApp/', include('myApp.urls')),
]
```

Step 4: Install the new app in the project by adding it to the list INSTALLED_APPS in the settings.py file of the project.

```
# ...this is just a slice of code within settings.py
# do not delete the other code
# just add 'myApp' to the list.
INSTALLED_APPS = [
    'myApp',
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]
```

## Jinja Templating

Jinja logic: {% block content %}

{% endblock %}

- Edit the views.py file:

```
from django.shortcuts import render

def index(request):
    return render(request, 'personal/home.html')
```

render takes the request parameter first, then the template to render, then an optional dictionary of variables to pass through to the template. With templates, and later on static files, you will be putting your templates in a templates directory, but each app is going to

have its own templates. That said, the entire website itself may have templates too, and these templates may actually simultaneously be used.

For example, generally, websites look very similar on their various pages, because they follow a template, usually referred to as a header. The header contains things like the navigation bar, maybe a banner image, and so on. This header stays the same, and should be contained in one main file. Imagine if you had a website with 100 unique pages, each containing the code for the navigation bar, and you wanted to change a button on it. You would have to edit 100 files, and that's very tedious! Thus, we have templates, like headers. That said, with Django, you're going to have headers as templates, and then other templates that "extend" these headers and act as parts.

First, we'll create a header file. Something super simple for now:

```
<!DOCTYPE html>
<html lang="en">

<head>
        <title>Harrison Kinsley</title>
        <meta charset="utf-8" />
</head>

<body class="body" style="background-color:#f6f6f6">
        <div>
          {% block content %}
          {% endblock %}
        </div>
</body>

</html>
```

The important Django-related bit is:

```
{% block content %}
{% endblock %}
```

...except that's not Django! It's not really Python either. A wild Jinja has appeared, Jinja2 specifically. Jinja is a Python templating engine, aimed at helping you to do dynamic things with your HTML like passing variables, running simple logic, and more! With Jinja, you will notice we are using {% %}, this denotes logic. For variables, you will see {%{% }}. The use of block content and endlock is one of the two major ways of incorporating templates within templates. This method is generally used for header/template combinations, but there is another you could use that I will show later. Next, let's create another HTML file that will serve as whatever will fill this block content. If you remember from 5 years ago, our intention was to serve a home.html, and now we finally have our chance!

personal/templates/personal/home.html

```
{% extends "personal/header.html" %}
{% block content %}
<p>Hey! Welcome to my website! Well, I wasn't expecting guests. Um, my name is Abhishek. I am a programmer.</p>
{% endblock %}
```

Much of this code is actually just Jinja logic, with just a simple sentence that is actually what will be generated. This is just a very simple example, normally there will be more here. What will wind up happening, however, is, when we tell Django to open this home.htmlfile, it will be read that this file "extends" header.html, and everything in the block content will be placed inside the block content logic that we built into the header.html file.

## Styling with Bootstrap, HTML & CSS

Bootstrap is a popular HTML/CSS and some javascript package that greatly helps people who are design deficient. Bootstrap isn't going to fix you entirely, but it can at least lend a helping hand. The Bootstrap CDN (content delivery network). At first use Bootstrap locally so you can also learn about referencing static files. Later, you may choose to use the CDN instead, as a CDN can deliver data at a faster speed than your server, due to being in multiple locations and serving from the location closest to your end-user. Once downloaded, extract to the dist folder, open that, possibly another, and then find the css, fonts, and js directories. Head to mysite/personal, making a new directory called static, and then another called personal within that one, and place the Bootstrap directories there. The path should be mysite\personal\static\personal, starting from the root mysite. Now that you have static files, you will want to make sure there is a setting for them in your mysite/settings.py file:

```
STATIC_URL = '/static/'
```

Now head to your personal/templates/personal/header.html template. We're going to call our new CSS file into action.

Add

```
{% load staticfiles %}
<link rel="stylesheet" href="{% static 'personal/css/bootstrap.min.css ' %}" type = "text/css"/>
```

To the <head></head> tag.

We use the load staticfiles command to load in all available static files. Next, we specifiy the path to the one we want in this case to reference for our stylesheet. The static directories start with /static/ and then we need to specify the rest of the way. For us, the rest of the way to the bootstrap.min.css file is personal/css/bootstrap.min.css. It is

important to note here that we can use this identical reference from any other app, Django is going to look in all /static/ directories in any installed apps for this path.

## New and updated <header.html>:

The main file we want to focus on, at least for now, for the general look and feel of our website is mysite/personal/templates/personal/header.html. Open that back up, and place in there something like:

```
<!DOCTYPE html>
<html lang="en">

<head>
	<title>Abhishek Singh</title>
	<meta charset="utf-8" />
	{% load staticfiles %}
	<link rel="stylesheet" href="{% static 'personal/css/bootstrap.min.css' %}" type = "text/css"/>
	<meta name="viewport" content = "width=device-width, initial-scale=1.0">

	<style type="text/css">
		html,
		body {
		  height:100%
		}
	</style>
</head>

<body class="body" style="background-color:#f6f6f6">
	<div class="container-fluid" style="min-height:95%; ">
		<div class="row">
			<div class="col-sm-2">
				<br>
				<center>
					<img src="{% static 'personal/img/profile.jpg' %}" class="responsive-img" style='max-height:100px;' alt="face">
				</center>
			</div>
			<div class="col-sm-10">
				<br>
				<center>
				<h3>Programming, Teaching, Entrepreneurship</h3>
				</center>
			</div>
		</div><hr>

		<div class="row">
		  <div class="col-sm-2">
```

```
            <br>

            <br>
             <!-- Great, til you resize. -->
                  <!--<div class="well bs-sidebar affix" id="sidebar"
style="background-color:#fff">-->
                  <div class="well bs-sidebar" id="sidebar"
style="background-color:#fff">
                    <ul class="nav nav-pills nav-stacked">
                          <li><a href='/'>Home</a></li>
                          <li><a href='/blog/'>Blog</a></li>
                          <li><a href='/contact/'>Contact</a></li>
                    </ul>
                  </div> <!--well bs-sidebar affix-->
            </div> <!--col-sm-2-->
            <div class="col-sm-10">

                  <div class='container-fluid'>
                  <br><br>
                    {% block content %}
                    {% endblock %}
                  </div>
            </div>
          </div>
      </div>
      <footer>
            <div class="container-fluid" style='margin-left:15px'>
                  <p><a href="#" target="blank">Contact</a> | <a
href="#" target="blank">LinkedIn</a> | <a href="#"
target="blank">Twitter</a> | <a href="#"
target="blank">Google+</a></p>
            </div>
      </footer>

</body>

</html>
```

The head just contains some meta info for the most part. The only change from before is the inclusion of some more CSS. This CSS just sets it to be the case that our html and body elements will take up 100% of the website. We also added

```
<meta name="viewport" content = "width=device-width,
initial-scale=1.0">,
```

which is mainly a suggestion by Bootstrap. Next, we begin defining a div in the body tag as a container with the minimum height of 95% of the page. We do this because

we may have times when the page is longer than 100% of the page (hence needing a scrollbar), but we also don't want our footer up at the top of the page. Next, we're using these "row" elements. This is Bootstrap. So are our container-fluid divs. For the row elements, check out: Bootstrap Grids. Next, we're using a Bootstrap Nav within a combination of a well and sidebar. I included an example of an affixed sidebar, where you can scroll and the sidebar stays in place. You can comment out the way we build the div, uncomment that one, and see for yourself. The problem comes up when you go to resize the page to something more like a cell-phone size.

This template calls for a profile.jpg image with the line

```
<img src="{% static 'personal/img/profile.jpg' %}" class="responsive-img" style='max-height:100px;' alt="face">.
```


## Passing variables from python to html using Jinja variables

To add a ‘contact’ page to the website, we need to do the following-

We will need a view, and a url pattern to return that view. Let's head into the personal/views.py file first:

```
from django.shortcuts import render
from django.http import HttpResponse

def index(request):
    return render(request, 'personal/home.html')

def contact(request):
        return render(request, 'personal/basic.html',{'content':['If you would like to contact me, please email me.','singh.abhishek.07197@gmail.com@gmail.com']})
```

Here, we're rendering the template, and also passing a dictionary. We can then reference 'content' within our template by referencing {{ content }}, for example. This will return the value of ['If you would like to contact me, please email me.','hskinsley@gmail.com'] within our template to work with. Speaking of that template, let's make it.

personal/templates/personal/basic.html

```
{% extends "personal/header.html" %}
{% block content %}
        {% for c in content%}
          <p>{{c}}</p>
        {% endfor %}
{% endblock %}
```

This basic.html template is aimed at producing simple text-based messages within the template. Thus, we can actually use it for a large variety of pages, it just happens to be a contact page in this case. We take the content, which is a list, and then we use a Jinja2 for loop to iterate through the list.

We've got our view done, now we need that URL pattern. So, let's add our URL pattern to the personal/urls.py file:

```
from django.conf.urls import url
from . import views

urlpatterns = [
    url(r'^$', views.index, name='index'),
    url(r'^contact/', views.contact, name='contact'),
]
```

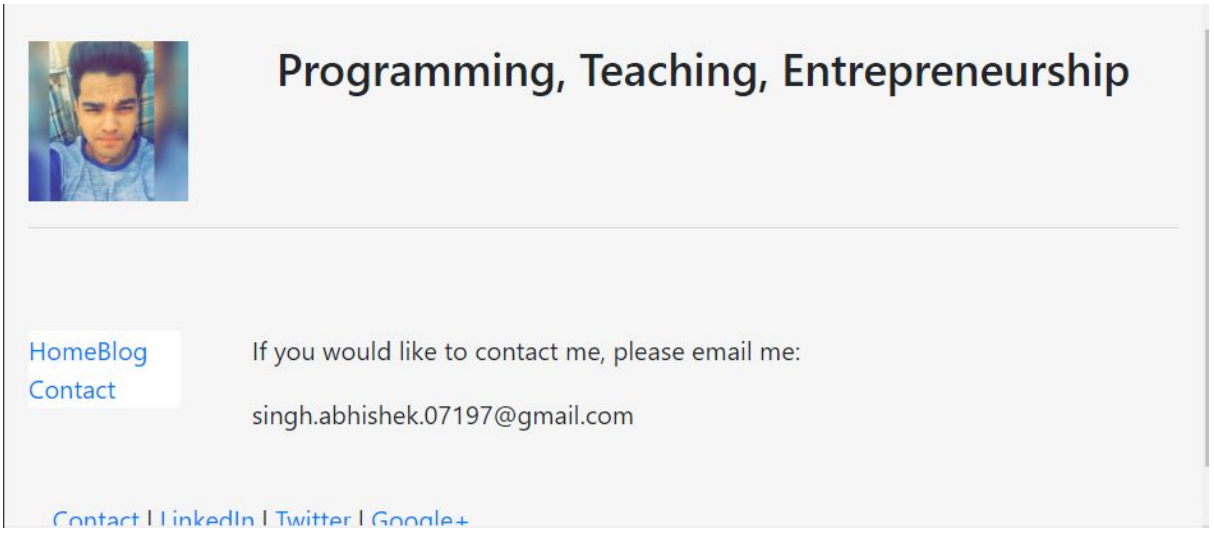
Now we are actually using this ^$ pattern in two locations, but we should be able to logically figure out this problem is caused by the main mysite/urls.py file, since it basically enforces that

the only way to get to the personal app is to not have any text, thus /contact/ cannot simply get there. Let's remove that dollar sign from mysite/urls.py:

```
from django.conf.urls import url, include
from django.contrib import admin

urlpatterns = [
    url(r'^admin/', admin.site.urls),
    url(r'^', include('personal.urls')),
]
```

Now run the server, and try again. You should have success when visiting http://127.0.0.1:8000/contact/




## Models

Each class in the models.py is a database table. Each variable is a table column, and then a datatype and maybe some further attributes like max_length and so on are set. If you wish to reference these objects and get something back besides that it is a Post or Category object, then you need to define the __str__ method. If you are using Python 2.7, you need to do __unicode__ instead of __str__.

We add a table 'Post' to our "Blog" by modifying the models.py file-

```
from django.db import models

class Post(models.Model):
    title = models.CharField(max_length = 140)
    body = models.TextField()
    date = models.DateTimeField()
```

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

## Views and Templates

We need to define our blog/urls.py file, create any template we need and traditionally also modify the blog/views.py. But we deploy some standard django functionality by adding generic views from django (DetailView, ListView wtc) to the blog/urls.py file:

```
from django.conf.urls import url, include
from django.views.generic import ListView, DetailView
from blog.models import Post

urlpatterns = [
            url(r'^$', ListView.as_view(
                        queryset=Post.objects.all().order_by("-date")[:25],
                        template_name="blog/blog.html")),
        ]
```

url() takes in a regex, view and template as parameters. Here, we are making use of the ListView, which is used mainly for a page that presents a list of something to the user. In our case, the list is going to be the last 25 blog posts ordered by date, descending (hence the - sign). Rather than writing an SQL-specific statement, we just reference our model 'Post'. Because we have the backend database chosen in our settings, Django knows how to build the query in the background.

Next, we need a blog HTML page, which we've called blog/templates/blog/blog.html as per the template_name in the blog/urls.py file:

```
{% extends "personal/header.html" %}
{% block content %}
        {% for post in object_list %}
                <h5>{{ post.date|date:"Y-m-d" }}<a href="/blog/{{post.id}}">  {{
post.title }}</a></h5>
        {% endfor %}
{% endblock %}
```

since it's a list, we're going to iterate through it. We'll display the post.date, in Y-m-d fashion. We then specify a dynamic URL for the specific blog itself so a user could click the link to head here. The url will lead to /blog/<postid>, and the text for the URL will be the title of the blog. We are able to reference any of the elements we've defined in the blog/models.py file.

# Databases and Migrations

Like how when you create a new app, the first thing you need to do is install it in settings.py, whenever you define new models, you want to migrate them.

```
python manage.py makemigrations
```

Output:

```
Migrations for 'blog':
  0001_initial.py
    - Create model Category
    - Create model Post
```

This makemigrations command tells Django that you've made some model changes, and you want to save them as a migration. Migrations are used any time you want Django to recognize changes to the models and database schema. Adding data to an existing database table, for example, is not something that needs a migration, changing that table's structure (by changing the model), would require a migration. With the above command, you will now have a blog/migrations directory, containing the proposed migration. You can also tell Django you want to make migrations for only a specific app, like:

```
python manage.py makemigrations blog
```

Once you've made migrations, nothing has actually happened yet. You can run a migrate, but there is one more check you can make:

```
python manage.py sqlmigrate blog 0001
```

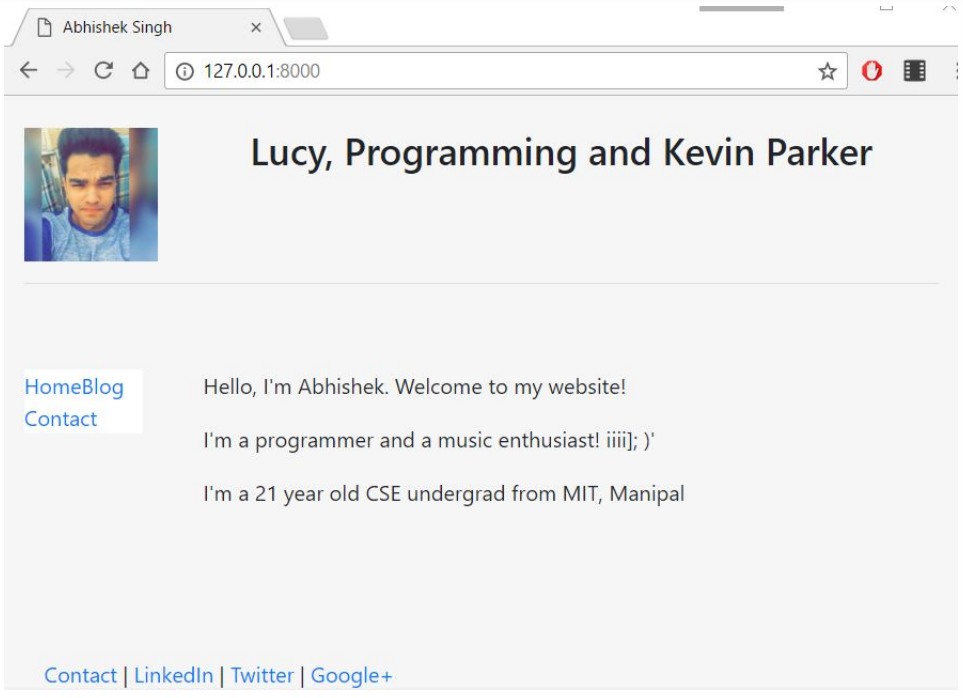
This will output the proposed SQL that will be run for you by Django when you migrate. It is a good idea to just read it over. Example -

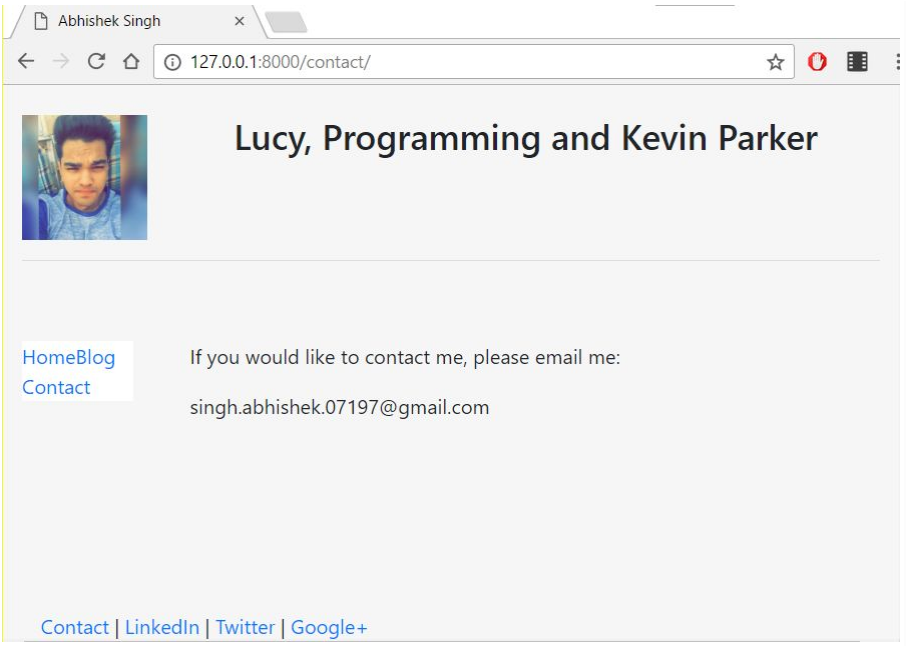
```
(new_test_environment) D:\Anaconda\envs\new_test_environment\myFirstDjangoProject\mySite>python manage.p
y sqlmigrate blog 0001
BEGIN;
--
-- Create model Post
--
CREATE TABLE "blog_post" ("id" integer NOT NULL PRIMARY KEY AUTOINCREMENT, "title" varchar(140) NOT NULL
, "body" text NOT NULL, "date" datetime NOT NULL);
COMMIT;
```

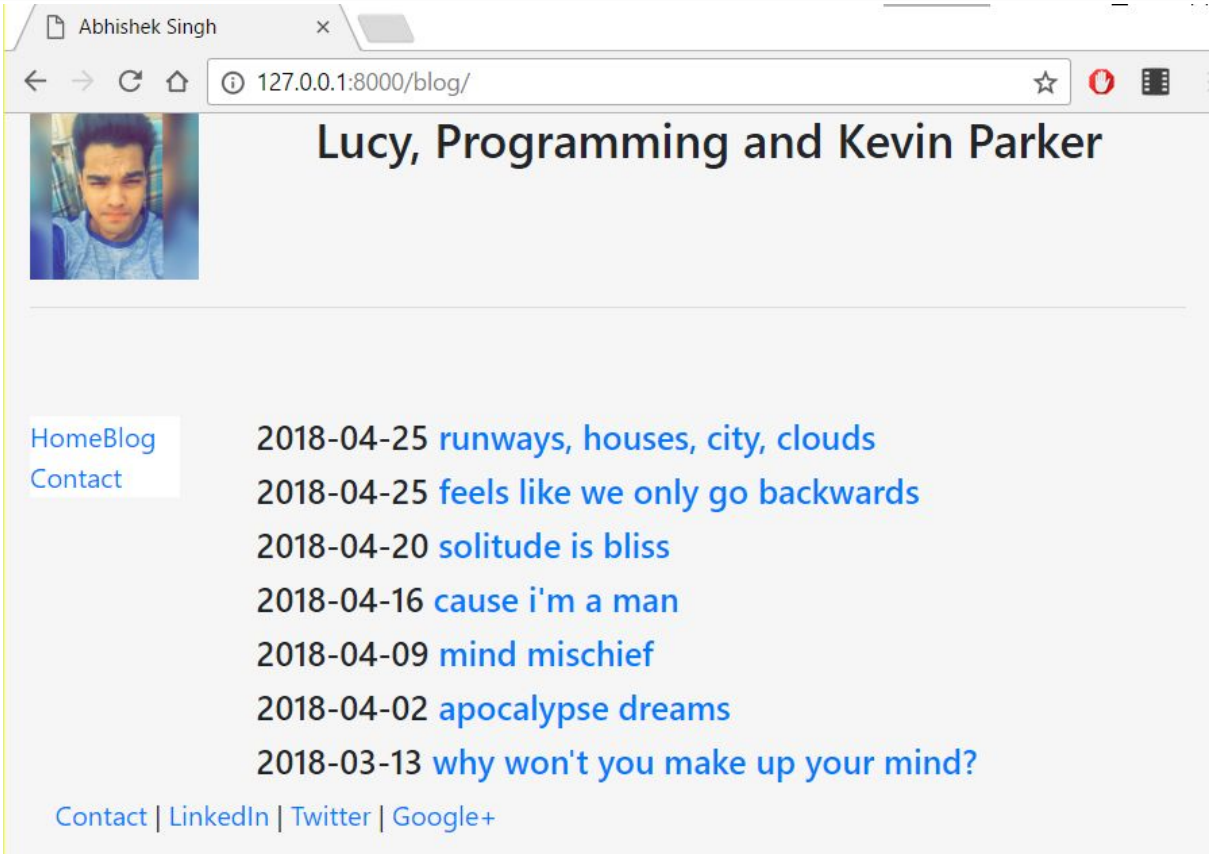
If all looks good, then you will do python manage.py migrate. This will actually perform the migrations. If this is your first time doing this, you should see quite a bit has been migrated:

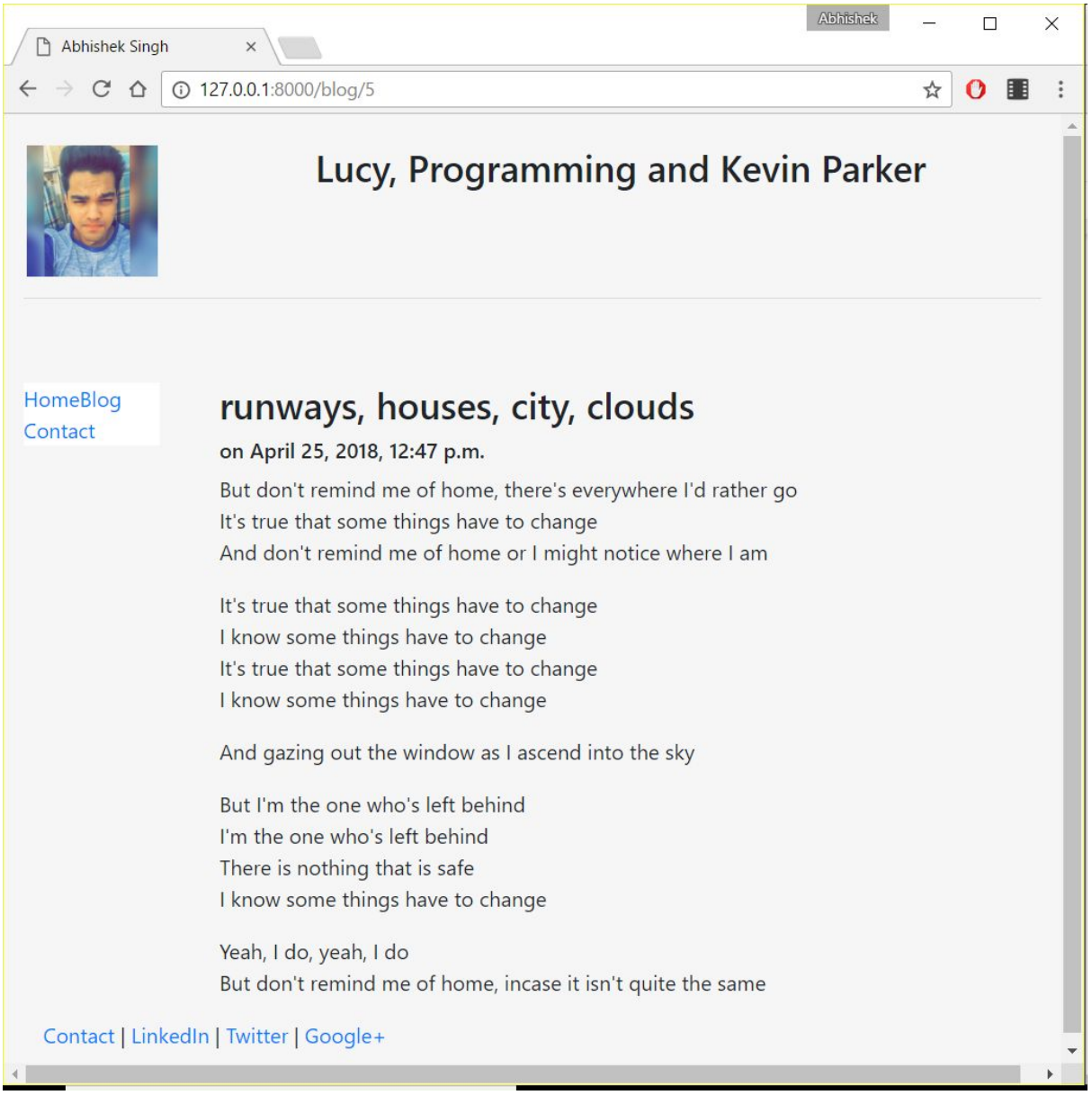
```
(new_test_environment) D:\Anaconda\envs\new_test_environment\myFirstDjangoProject\mySite>python manage.p
y migrate
Operations to perform:
  Apply all migrations: admin, auth, blog, contenttypes, sessions
Running migrations:
  Applying blog.0001_initial... OK
```

# Final View




Abhishek Singh
127.0.0.1:8000/contact/
Lucy, Programming and Kevin Parker
HomeBlog
Contact
If you would like to contact me, please email me:
singh.abhishek.07197@gmail.com
Contact | LinkedIn | Twitter | Google+


Abhishek Singh
127.0.0.1:8000/blog/
Lucy, Programming and Kevin Parker
HomeBlog
Contact
2018-04-25 runways, houses, city, clouds
2018-04-25 feels like we only go backwards
2018-04-20 solitude is bliss
2018-04-16 cause i'm a man
2018-04-09 mind mischief
2018-04-02 apocalypse dreams
2018-03-13 why won't you make up your mind?
Contact | LinkedIn | Twitter | Google+


Abhishek
Abhishek Singh
127.0.0.1:8000/blog/5
Lucy, Programming and Kevin Parker
HomeBlog
Contact
runways, houses, city, clouds
on April 25, 2018, 12:47 p.m.
But don't remind me of home, there's everywhere I'd rather go
It's true that some things have to change
And don't remind me of home or I might notice where I am
It's true that some things have to change
I know some things have to change
It's true that some things have to change
I know some things have to change
And gazing out the window as I ascend into the sky
But I'm the one who's left behind
I'm the one who's left behind
There is nothing that is safe
I know some things have to change
Yeah, I do, yeah, I do
But don't remind me of home, incase it isn't quite the same
Contact | LinkedIn | Twitter | Google+