

MyRecipes Django application

This document contains features an introduction to Django and Python. It focuses on creating a basic recipes app using Django - a web framework written in python.

Django & Python: What are they?

Out of all the programming languages out there, python remains to be one of the most learnt. It is used for all sorts of tasks and uses, starting from the most basic to the most complex. This makes python extremely powerful. You may have used applications that feature Python such as Spotify, Dropbox and YouTube. Programming languages use frameworks to simplify the creation of complex applications, we could essentially recreate all that a framework has to offer in any application but using one saves countless hours. One such framework is Django, Django is a web framework for the Python programming language that allows you to create web sites extremely fast with modern website features (including security, a built-in database and the creation of websites without the need for interfering with HTML for every change). In our case, we will be using Python's framework Django to create an application that will store our recipes.

The project

The goal of this project was to create a web-based application that would store recipes on a database using Django. To create this, I used the following tools:

- MacOSX 10.10.4.
- Web browser (Safari)
- Command line application (Terminal)
- Code editor with support for Python (Visual Studio Code)
- Code editor with support for HTML (Brackets)

I also used an application called sqlitebrowser to better understand SQLite and the database that would hold the information in my program. The actual website template was obtained from an online source (bootstrap) and modified to better fit my program.

The following design specifications were set for the project:

1. A list view where all the recipes can be viewed
2. A details view where recipe information could be viewed (including ingredients, utensils and directions)
3. An edit mode where the end user would be able to add and modify recipes.
4. The list and detail views were to be well designed to look pleasing to the end user (using HTML and CSS).

Installing the necessary software

I downloaded Python 3.4.3 for Mac OSX and installed it from python.org. The installation was very straightforward as it uses a standard OS X installer. To verify my installation, I executed the code `python` (or `python3`) on the terminal application on my mac. This starts the python shell and also lists the version number that is being used. Note: I have several versions of python installed, therefore I use the `python3` command to call version 3 of python, if I use the `python` command, it will run version 2. This may not be the case for others. Django uses python v3.

Next step was to download and install Django, which was different to installing a regular program. I simply followed the install guide on their website (<https://www.djangoproject.com/download/>). There are multiple procedures listed on their site. The procedure I used was using pip.

I downloaded a file called get-pip.py from the pip website (<https://pip.pypa.io/en/latest/installing.html#install-pip>).

I set my directory from OS X terminal to the download location.

I ran the following command from OS X Terminal to install pip:

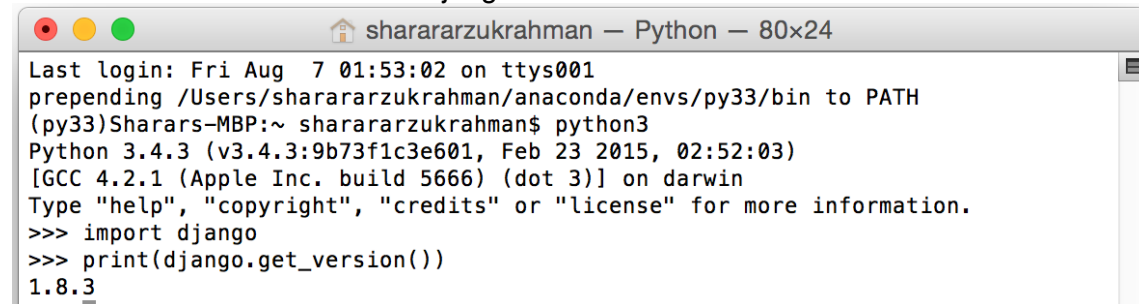
```
python3 get-pip.py
```

Next, I used pip to install Django by again using the Terminal application, the command was:

```
pip install Django==1.8.3
```

The command downloads and installs Django. After which, I restarted the Mac (just to be on the safe side).

After the reboot, I ran the terminal app and started the python shell, I used the following commands as shown to ensure django was installed.

A screenshot of a macOS Terminal window. The title bar shows the window name 'sharararzukrahman — Python — 80x24'. The terminal output shows the last login time, the path to the Python 3.4.3 environment being prepended to the PATH, and the execution of 'python3'. It then shows the Python version and GCC information. Finally, it shows the execution of 'import django' and 'print(django.get_version())', which outputs '1.8.3'.

```
Last login: Fri Aug 7 01:53:02 on ttys001
prepending /Users/sharararzukrahman/anaconda/envs/py33/bin to PATH
(py33)Sharars-MBP:~ sharararzukrahman$ python3
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 23 2015, 02:52:03)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import django
>>> print(django.get_version())
1.8.3
```

As said above, make sure you have the right code editors that support the HTML and Python syntaxes.

Starting the project

To create the myRecipes project, the first step was to initiate the project using the django command *startproject*: first navigate to a directory of your choice in the OS X terminal and run the following command (make sure you're not in the python shell, there should be no >>> at the beginning of a line):

```
django-admin startproject myRecipes
```

This created a myRecipes directory in my chosen directory, there were several files and folders created that are needed for the project. This is one of the main advantages of django, it greatly reduces the time you spend coding by allowing us to use and modify templates

While you're browsing the directories, open up the settings.py (in myRecipes/myRecipes) file and change the timezone to yours, here is a list of timezone inputs: <http://stackoverflow.com/>

[questions/13866926/python-pytz-list-of-timezones](https://stackoverflow.com/questions/13866926/python-pytz-list-of-timezones). I set the timezone as follows:

```
87 # https://docs.djangoproject.com/en/1.8/topics/i18n/
88
89 LANGUAGE_CODE = 'en-us'
90
91 TIME_ZONE = 'Etc/GMT+6'
92
93 USE_I18N = True
94
```

Next step was to create a database and to check functionality of the builtin server, django has built in SQLite.

The following commands were used in the terminal after navigating terminal to the myRecipes folder:

```
python3 manage.py migrate
python3 manage.py runserver
```

In the above, the first looks at the apps installed in the settings file and creates the necessary databases. The second starts a development server which can be accessed (by default) on the following IP address: <http://127.0.0.1:8000/>

You will see a Django welcome page when trying to access this address.

The project is now set up.

Creating models

Data is stored in a database using models, we will now create a model to store our data. As mentioned in the design specification, I needed to store the following information about a recipe:

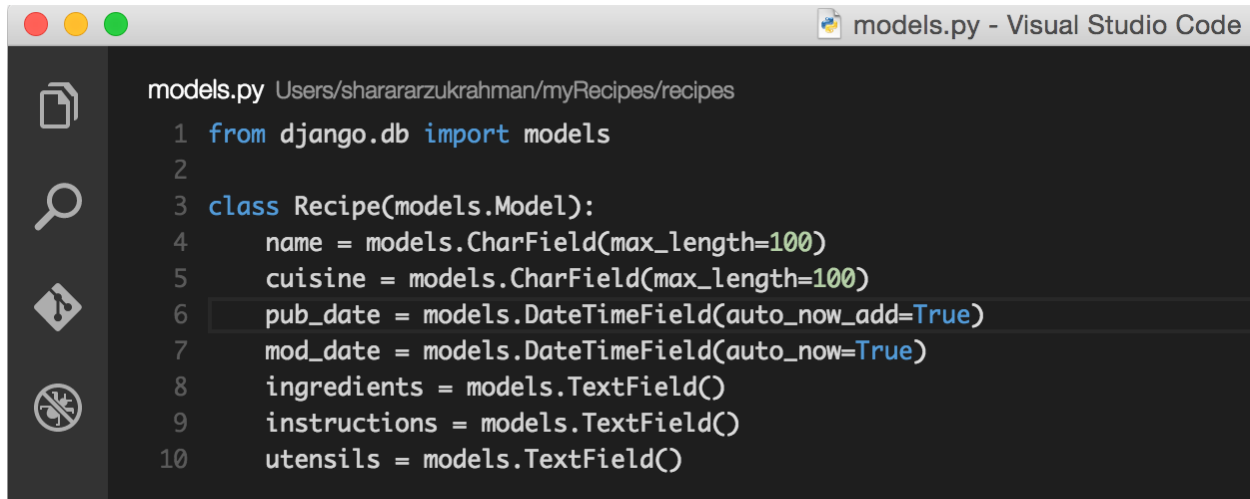
- Ingredients
- Tools needed
- Directions
- I also chose to add the following:
 - Cuisine
 - Publish date
 - Date last modified

Data models are created and stored in the models.py file, you will notice that there is no such file in our directory, models are part of the apps our project uses. The project is currently only using apps that are built into django. We need to create a custom app. This was done by opening terminal and navigating to the myRecipes directory, after which I ran a command to create an app called recipes:

```
python3 manage.py startapp recipes
```

This creates the model file we need.

Here is the model I created for my project:



```
models.py Users/sharararzukrahman/myRecipes/recipes
1 from django.db import models
2
3 class Recipe(models.Model):
4     name = models.CharField(max_length=100)
5     cuisine = models.CharField(max_length=100)
6     pub_date = models.DateTimeField(auto_now_add=True)
7     mod_date = models.DateTimeField(auto_now=True)
8     ingredients = models.TextField()
9     instructions = models.TextField()
10    utensils = models.TextField()
```

The first line in this file imports the ability to make models. From the specification I set myself, I just needed one class. Each class has its own data table in our database and classes can be linked to one another if necessary. The class I created was called Recipe.

I needed to add the relevant fields to store the information I needed to store, along with the field name to store the recipes name.

Name and cuisine are smaller fields and shouldn't be more than a hundred characters. I needed to limit this because I plan to use these fields as titles and subtitles in the website the end user will access, if users enter large amounts of text in these, my website will not display the way I need it to. These limits were also set to make the editor I plan to use for users to edit the recipes aesthetically pleasing. Name and cuisine are *CharFields*, which has the required argument of *max_length* or maximum length. Ingredients, instructions and utensils are larger amounts of information so these were *TextFields*, you will notice there are no limitations set; we have to be sure to set up our end-user view so that it appears clean to them. Two more things to cover — *pub_date* and *mod_date*, these correspond to publishing date and date last modified respectively. *pub_date* and *mod_date* need to store dates and times, they have the field type of *DateTimeField*. I added a few optional arguments that allow me to store the publishing date and date last modified without input from the user. *auto_now_add* only automatically adds the date once and cannot be modified, whereas the *auto_now* adds the date each time the user changes the data, the boolean value of *True* turns these arguments on.

Our models are complete!

I now have to create a table in my database according to the data model I created, this was done with a few commands in the terminal application. The first step to do is to open the *settings.py* file we accessed earlier when setting the timezone, under installed apps add a line (while maintaining format) with the name of the app we started earlier, in my case this was *recipes*. The terminal commands that were used were:

```
python3 manage.py makemigrations
```

Followed by:

```
python3 manage.py migrate
```

These two commands must be run every time the models are changed (because the way data is stored in our database is being changed). The 'make migrations' command creates migrations for the changes in our model, and the 'migrate' command applies the changes to our database.

Setting up the admin site

We are making some real progress, time to set up the editor the end user will use. I chose to do this with the built-in admin site in Django as this also has authentication features, so when other people come to view the recipe database they need to be logged in to edit recipes.

To create an administrator I ran the following command from terminal:

python3 manage.py createsuperuser

Terminal will then prompt you to enter a username, email and password. With the development server started (*python3 manage.py runserver*), accessing <http://127.0.0.1:8000/admin/> will show a login screen where the username and password created earlier need to be used.

Django administration

Username:

Password:

The goal now is to setup the admin area of the site so that all the information the user needs is available.

This was done as shows in the admin.py file in the 'recipes' application directory:

```
admin.py Users/sharararzukrahman/myRecipes/recipes
1 from django.contrib import admin
2
3 from .models import Recipe
4
5
6 class RecipeAdminView(admin.ModelAdmin):
7     list_display = ('name', 'cuisine', 'pub_date', 'mod_date')
8
9
10 admin.site.register(Recipe, RecipeAdminView)
11
```

I import the class I created in my models file using: *from .models import Recipe*.

I created a class called `RecipeAdminView` in this file that will allow me to view fundamental details about a recipe on the admin site. The `list_display` list will display the content I defined in it as columns in the admin layout. I added `name`, `cuisine`, `pub_date` and `mod_date`. These are all fields defined in my `models.py` file. The final line of code in this file registers the classes I created and imported them into my admin site.

The admin site (the editor in this case) is ready to go now. You will now be able to access the recipes app from the main page and be able to add and modify recipes by accessing the recipes link on the django administration site.

Here is the result of the `RecipeAdminView` class we created earlier:

Here is a specific recipe being edited:

Change recipe

Name:

Cuisine:

Ingredients:

Instructions:

Utensils:

[Delete](#) [Save and add another](#) [Save and continue editing](#) [Save](#)

This is fantastic, we have an editor in our hands. But the user does not have a way to view the recipes without accessing the editor yet. We will now create a 'view' which the end user will see.

Creating a view

I already specified in the design specification that there must be a list view and a details view. This corresponds to two functions in the file `views.py` in `myRecipes/recipes/`. This is how I set up the file:

```
views.py Users/sharararzukrahman/myRecipes/recipes
1 from django.shortcuts import render
2 from django.http import Http404
3 from .models import Recipe
4
5 def index(request):
6     recipelist = Recipe.objects.all()
7     context = {'recipelist': recipelist}
8     return render(request, 'recipes/index.html', context)
9
10 def detail(request, recipe_id):
11     try:
12         recipe = Recipe.objects.get(pk=recipe_id)
13     except Recipe.DoesNotExist:
14         raise Http404("Recipe is not defined")
15     return render(request, 'recipes/detail.html', {'recipe': recipe})
16
17
```

The first 3 lines of code imports the necessary functions to create the web page I need. The third line imports data from our model. The render module renders a final website that corresponds to data in the function. The `http404` module is used to warn the user that a Recipe is not defined. Let's look closer at each function

The `index` function corresponds to the list view I want my user to view when my user first accesses the site. The render command takes three arguments which I needed to create, context takes information from the objects defined in the Recipe class in my data model to display it using the HTML template called `index.html` (which has not been created yet).

The `detail` function takes an argument called `recipe_id`, `recipe_id` will be used to refer to each of the recipes stored using a primary key or pk in my database. When a new recipe is defined in my database, it receives a primary key which can be used to call it. Here I'm calling the object inside my database based on the primary key and `recipe_id` values. The function will try to read the information associated with my recipe and attempt to return it using the render command using the template called `detail.html`. If it fails, it will raise the exception placed under 'except'. If the recipe does not exist it will raise the `Http404` page with the text "recipe is not defined".

The behavior of our views are now defined. The next goal is to control how our URL's are assigned. Create a file in the myRecipes/recipes directory called urls.py. There is another urls.py file in the myRecipes/myRecipes directory. Our project will use these two files to map out URLs in a way that will allow us to view our recipes. We will first start by modifying the urls.py file in myRecipes/myRecipes.

Here is how I set it up:

```
urls.py Users/sharararzukrahman/myRecipes/myRecipes
1 from django.conf.urls import include, url
2 from django.contrib import admin
3 from django.views.generic.base import RedirectView
4 from recipes.models import Recipe
5
6 urlpatterns = [
7     url(r'^$', RedirectView.as_view(url='/recipes/')),
8     url(r'^admin/', include(admin.site.urls)),
9     url(r'^recipes/', include('recipes.urls')),
10 ]
```

The first two lines imports modules that are necessary for us: the admin module contains all the necessary information for our admin site and the two modules called in the first line allow us to include other urls.py files (such as the one in our myRecipes/recipes directory) and the url module allows for us to create url patterns.

The third line imports a module that allows us to redirect sites to other sites.

The fourth line imports our Recipe class in our models file.

We then create a URL Pattern which maps out how urls are created.

First thing you will notice is that every time you access the base development server (<http://127.0.0.1:8000/>) you are presented with a site that does not exist, we want our end user to be able to access the recipe index view that we defined previously. This is where the redirect module comes to play. The following line directs the browser from 127.0.0.1:8000 to 127.0.0.1:8000/recipes/ when the site is opened. This means the site will directly redirect the url corresponds to `r'^$', RedirectView.as_view(url='/recipes/')`,

The line of code below uses the admin module called and maps out our admin page, this is a good built in feature of Django and requires no modifications. `r'^admin/'` corresponds to the 127.0.0.1:8000/admin site in our development server.

```
url(r'^admin/', include(admin.site.urls)),
```


The last line of our url patterns below calls upon the urls.py file placed in our recipes directory and maps it under 127.0.0.1:8000/recipes.

```
url(r'^recipes/', include('recipes.urls')),
```

This is our main URL mapping done. Time to map out the URLs for our recipes application and create URLs so they correspond to the recipe_ids (and primary keys). Opening the myRecipes/recipes/urls.py file we created previously will allow us to add the necessary URL mapping configurations.

Here is how I set mine up:

```
urls.py Users/sharararzukrahman/myRecipes/recipes
1 from django.conf.urls import url
2
3 from . import views
4
5 urlpatterns = [
6     url(r'^$', views.index),
7     url(r'^(?P<recipe_id>[0-9]+)/$', views.detail),
8 ]
```

The modules that are needed for this are the URL model and our views file in our recipes application:

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

I have previously showed the function of the url module. Since the urls defined here will be used to create views with templates (and also raise 404 errors and what not), the views file is referenced as a module.

The url patterns here are somewhat more complex.

In the following line the index function in our views file is called as this corresponds to the list of recipes which is the purpose of our index function. Here, `r'^$',` again refers to the base directory, which in this case is 127.0.0.1/recipes/ (as this is the base directory for this urls file, similar to how our file directory is set up)

```
url(r'^$', views.index),
```

The next line here is where the magic occurs:

```
r'^(?P<recipe_id>[0-9]+)/$'
```

This creates urls based on the recipe_id (or primary key) in our database, if a recipe_id number does not correspond to a recipe in our database the http404 error is raised, this information is all read from the detail function in our views file, which this line of code makes reference to.

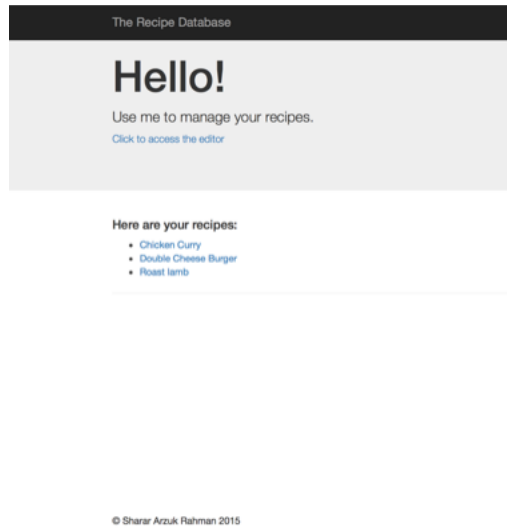
```
url(r'^(?P<recipe_id>[0-9]+)/$', views.detail),
```

The way our views are created as well as URL mapping is done are now complete. We are almost ready to use the site. The only thing left is to create the html files that our web browsers will read to display the information.

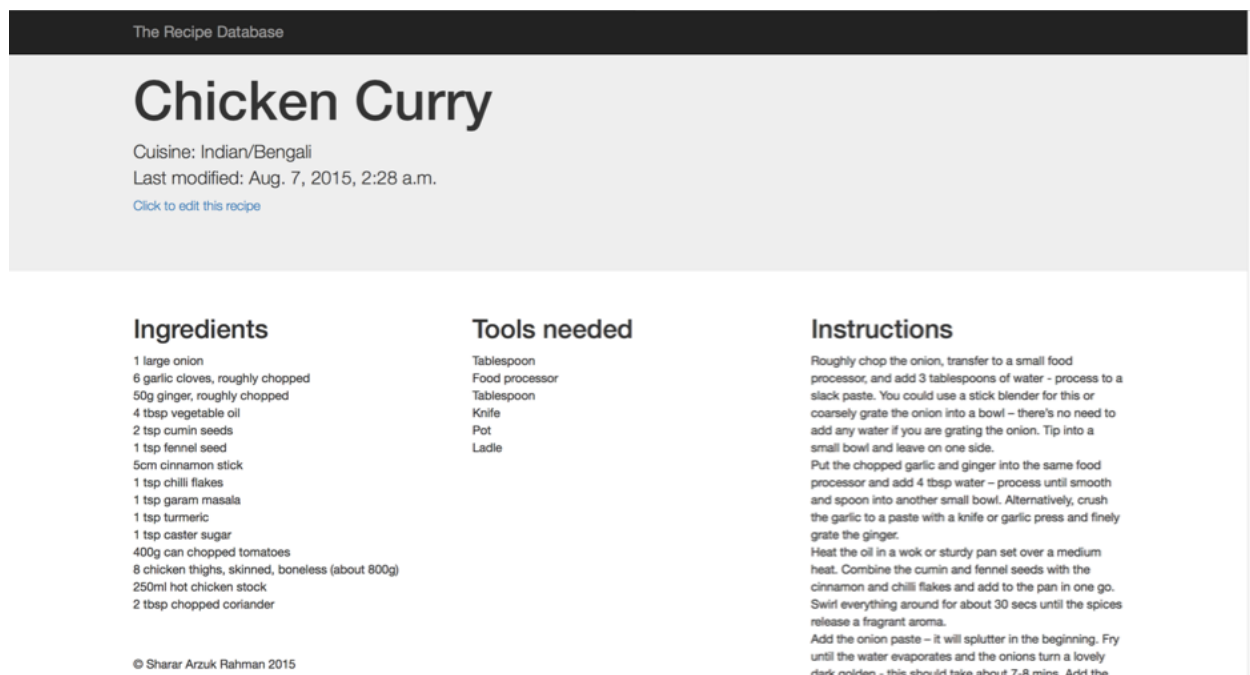
The first step is to hop on your favorite HTML editor and create a website as you wish. The main and interesting part (especially to people with experience in HTML) about these files is not the html code itself, but the way python is injected into the html to call data from our database.

My website ended up looking as follows:

The main list view:



The details view:



I will not place much focus into the HTML elements but rather the python elements in our html file. If you look at our views.py file, you will see that files are represented under a recipes directory, i.e. recipes/index.html. This is done in good practice because a django project may have many files called index.html and it is important to isolate them into different directories, otherwise if index.html was called, django would call the first index.html file it saw. To create the directories and files we need, open myRecipes/recipes and create a directory called templates, inside it, create a directory called recipes. The file directory should be like this : myRecipes/recipes/templates/recipes/. Inside this directory place two files: index.html and detail.html, just as defined in our views.py file. We are ready to look at the python code in our html files!

Looking at the index.html file

There are only two bits of code that are important in our HTML file, I wanted to be able to access the editor (admin site) to the area where all the recipes are displayed. To do this I simply accessed the admin site and copied the URL corresponding to the correct edit site.

```
<p>Use me to manage your recipes.<br> <a style="font-size:15px" href ="/admin/recipes/recipe"> Click to access the editor</a></p>
```

The picture below is where the recipes are listed using python and html, I wanted to view the recipes as a list so I created an unordered list using , but before that I created an if statement to verify the recipes exist, or else the text “No recipes are available is displayed”. You will note that python is embedded into our html file using either {% %} or {{ }}, the first corresponds to functions and the second corresponds to variables. A ‘for loop’ is executed to print the ‘name’ (in our models.py file) attribute of the ‘recipe’ in our recipelist (which is defined in our views files) as a list. Each recipe is hyperlinked to it’s corresponding recipe id (or primary key). We already know that when the recipe/number/ (where number is an integer) is opened in our web browser, the detail.html file is referenced as written in our views.py file. One thing to note is that ‘for loops’ must be ended since python’s indentations are not supported, this is done with {% endfor %} and in the case of ‘if’ statements {% endif %}. This concludes how the index.html file works:

```
{% if recipelist %}
    <h4> Here are your recipes:</h4>
    <ul>
        {% for recipe in recipelist %}
            <li><a href ="/recipes/{{ recipe.id }}/">{{ recipe.name }}</a></li>
        {% endfor %}
    </ul>
{% else %}
    <p>No recipes are available.</p>
{% endif %}
```

Looking at the detail.html file

This is even simpler since all that is required is to print the fields that are already stored. Our views file knows to call the right recipe stored in our database and display its various attributes (using recipe_id).

Since calling variables are so simple. I used the name, cuisine and mod_date (or date last modified) variables and put them in various places the user would want to see them

I also added a link on the detail.html page that would open the admin site to the location where the recipe information can be edited to allow for direct access, this is done again using the primary key. This is shown below:

```
<h1>{{ recipe.name }}</h1>
<p>Cuisine: {{ recipe.cuisine }}<br> Last modified: {{ recipe.mod_date }} <br> <a style="font-size:15px" href ="/admin/recipes/recipe/{{ recipe.id }}">
Click to edit this recipe</a></p>
```

My website was set up to display the recipe information in divs that have controlled widths to allow for a well contained display. To allow for my site to display the fields in the database so that line breaks are supported (and text is not displayed in one huge line), I had to add the linebreaksbr command to each of the textfield variables. So for our ingredients this was {{ recipe.ingredients|linebreaksbr }}, which calls the ingredients field under our recipes and prints it with the line breaks as we need them. This is shown below:

```
<h2>Ingredients</h2>

{{ recipe.ingredients|linebreaksbr }}

</div>
<div class="col-md-4">
  <h2>Tools needed</h2>
  <p> {{ recipe.utensils|linebreaksbr }} </p>

</div>
<div class="col-md-4">
  <h2>Instructions</h2>
  <p> {{ recipe.instructions|linebreaksbr }}</p>

</div>
```

DONE!

You have successfully replicated my django project (minus the HTML aspects), feel free to use it by accessing the development server and play around with the various functions.