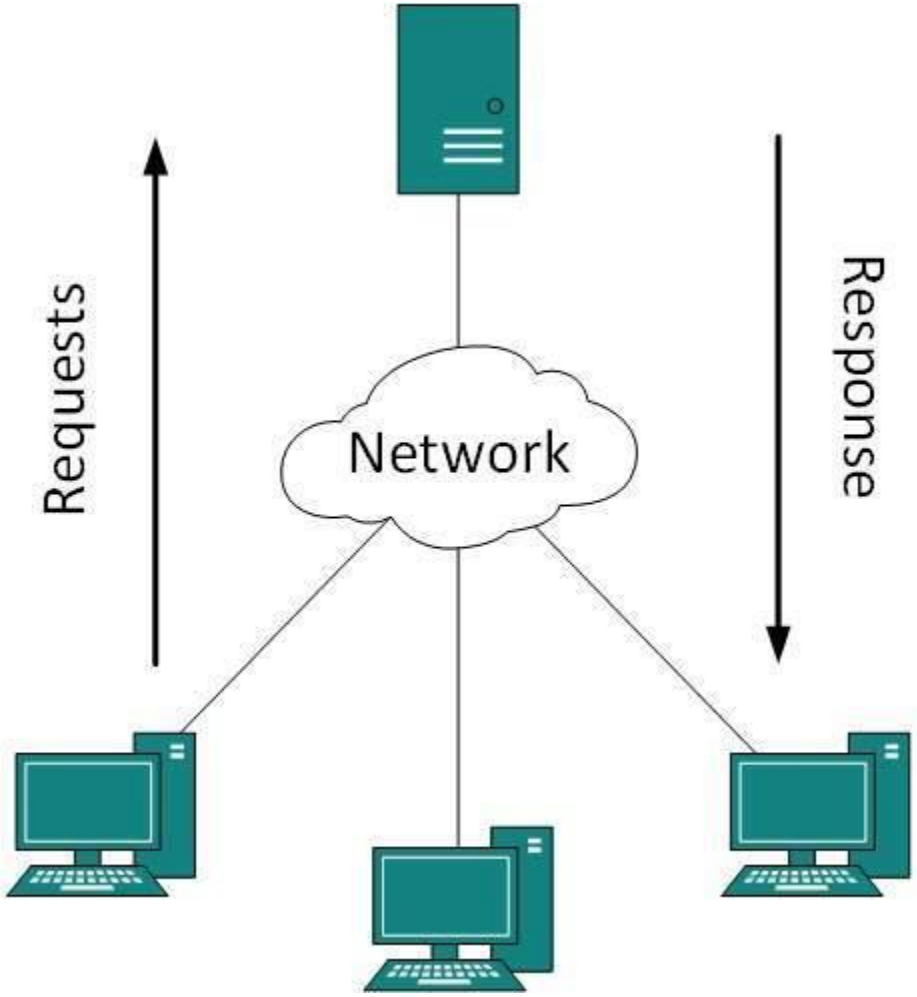
Python
Server-Side Framework
Models & Serialization

Lab 7

Review

- Server Side Development
  - Using python to develop backend services to support complex operations within web applications
  - Python code will be evaluated by the interpreter on the server, web-oriented content will be converted to HTML and sent to the client
  - All JS, CSS and HTML code will be evaluated on the browser
  - We use server-side system to:
    - Develop complex applications with complex permission structures and control levels
    - Build complex computational services (such as big data processing)



Client-Server Model (src: TutorialPoint)

Review

- Python 3, pip/virtualenv, Django
  - **Python 3** – Python is a popular language for developing many services including web development. While as of 2017-18 Python 2.7 is popular, Learning and developing applications in Python 3 is a more sustainable path.
  - **Django** – “Django is a high-level Python Web framework that encourages rapid development and clean, pragmatic design.” (source)
  - **Pip/virtualenv** – Both pip and virtualenv are tools to enable the isolated development of multiple applications with different library and versions. Again keep in mind that we need to use python 3 version of these tools

Review – Setup

- Ensure OS can support python-database connectivity driver for the DBMS used. In our case mysql - *sudo apt install -y python3-mysql.connector*
- Ensure we have the tools needed for development – *sudo apt install -y python3-pip python3-virtualenv python3-venv python3-django libevent-dev*
  - Includes:
    - Pip
    - Virtualenv (venv)
    - Django

Review – Setup

- Environment commands: The following commands go through the process of setting tools to create a virtualized environment for our app development
- Tasks include:
  - Creating a root directory for all project related resources
  - Creation of the virtual environment in the folder called venv
  - Install django to the current environment

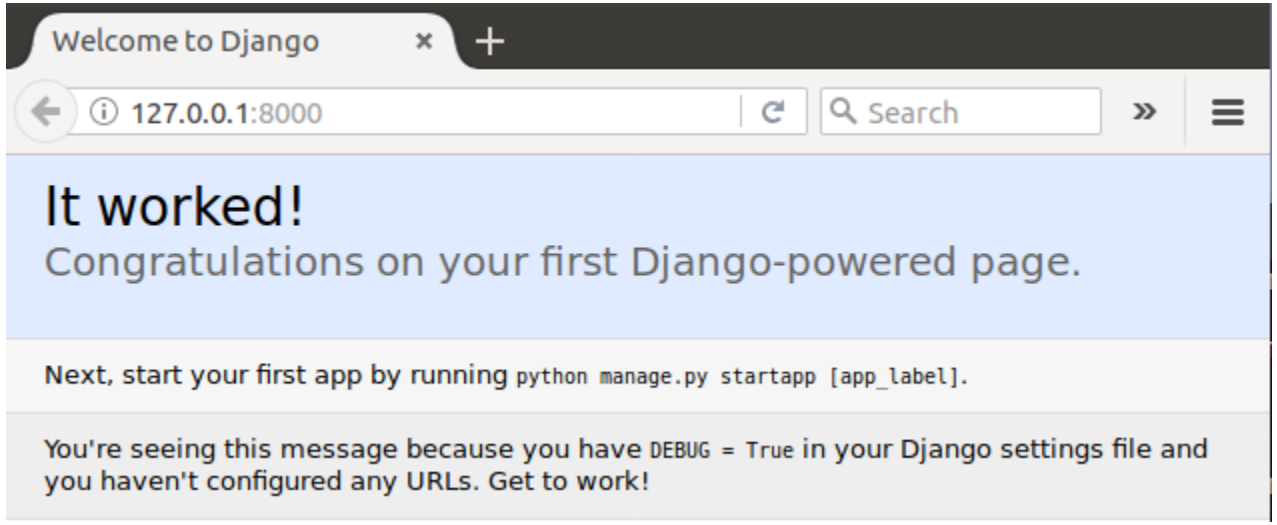
```
adminuser@adminuser-virt:~/dev$ mkdir python-sms
adminuser@adminuser-virt:~/dev$ cd python-sms/
adminuser@adminuser-virt:~/dev/python-sms$ python3 -m venv venv
adminuser@adminuser-virt:~/dev/python-sms$ ls
venv
adminuser@adminuser-virt:~/dev/python-sms$ source venv/bin/activate
(venv) adminuser@adminuser-virt:~/dev/python-sms$ which python
/home/adminuser/dev/python-sms/venv/bin/python
(venv) adminuser@adminuser-virt:~/dev/python-sms$ python --version
Python 3.5.3
(venv) adminuser@adminuser-virt:~/dev/python-sms$ pip install Django
Collecting Django
```

Review – Setup

- We create and run our initial application
- Tasks include:
  - Creating a Django application
  - Run the local development server for the Django application using the command “runserver”
  - Navigating to the link should provide the view of the default page of the Django application

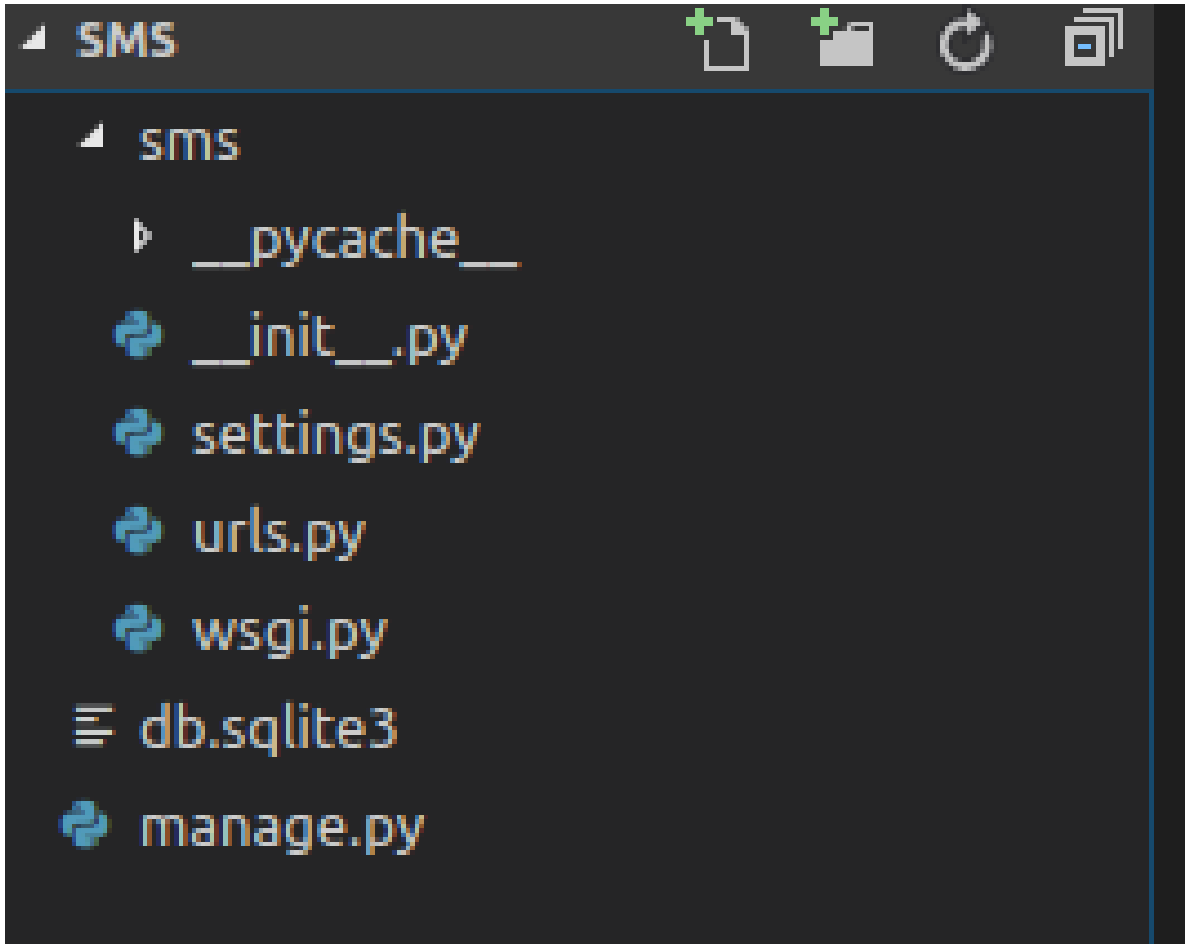
```
(venv) adminuser@adminuser-virt:~/dev/python-sms$ which django-admin
/home/adminuser/dev/python-sms/venv/bin/django-admin
(venv) adminuser@adminuser-virt:~/dev/python-sms$ django-admin startproject sms
(venv) adminuser@adminuser-virt:~/dev/python-sms$ ls
sms  venv
(venv) adminuser@adminuser-virt:~/dev/python-sms$ cd sms
(venv) adminuser@adminuser-virt:~/dev/python-sms/sms$ python manage.py runserver
Performing system checks
```

```
Django version 1.11.6, using settings 'sms.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```



Review - App

- The application files are provided as follows:
  - Settings: Contains the base configuration for the application. Default values are good for development, but may require changes for production.
  - Urls: Specify valid HTTP request that the project accepts
  - Wsig: Handles the connection between the python application and the HTTP server such as gunicorn, Apache and ngnix
  - manage: is a control file for the dnago system to perform common system operations



Review - App

- ALLOWED_HOSTS is the list of sites that Django will accept to make connection and requests.
- For local development (i.e. when DEBUG is True) we can have this be empty, however, when in production it must be set to a specific domain such as .example.com.

```
DEBUG = True

ALLOWED_HOSTS = []
```

Review - App

- The INSTALLED_APPS array is the list of applications that the Django project is aware of. It is initialized with a number of default Django applications that run the base project.
- A common development practice is to specify separate application list based on the source. These groupings include:
  - DJANGO APPS
  - THIRD PARTY APPS
  - LOCAL APPS

```
33  INSTALLED_APPS = [
34      'django.contrib.admin',
35      'django.contrib.auth',
36      'django.contrib.contenttypes',
37      'django.contrib.sessions',
38      'django.contrib.messages',
39      'django.contrib.staticfiles',
40  ]
```

```
33  # use tuple () rather than list[] because of immutability
34  DJANGO_APPS = (
35      'django.contrib.admin',
36      'django.contrib.auth',
37      'django.contrib.contenttypes',
38      'django.contrib.sessions',
39      'django.contrib.messages',
40      'django.contrib.staticfiles',
41  )
42  THIRD_PARTY_APPS = ()
43  LOCAL_APPS = ()
44  INSTALLED_APPS = DJANGO_APPS + THIRD_PARTY_APPS + LOCAL_APPS
```

Review - App

- The concept of middleware is taking requests/responses (HTTP communication) as they enter/leave the Django system and applying functions to them before/after being processed in a light weight manner (source)
- NB: Order of middleware matters

```
MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
]
```

Review - App

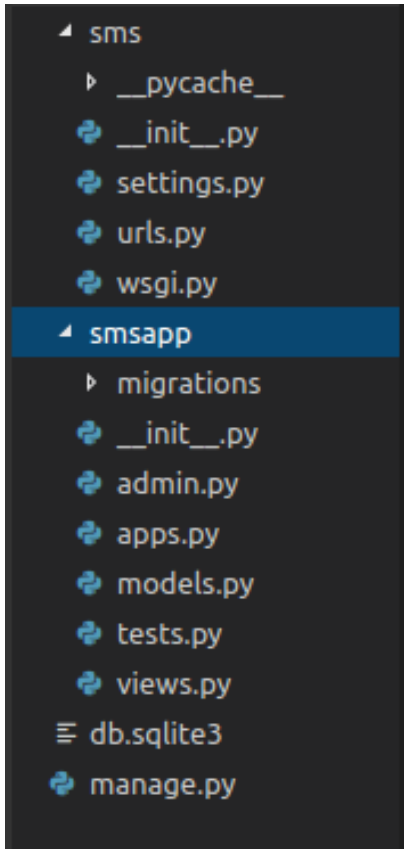
- A list containing the settings for all template engines to be used with Django. Each item of the list is a dictionary containing the options for an individual engine. (source)

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    },
]
```

Create App

- Create the application to serve student related content within our project

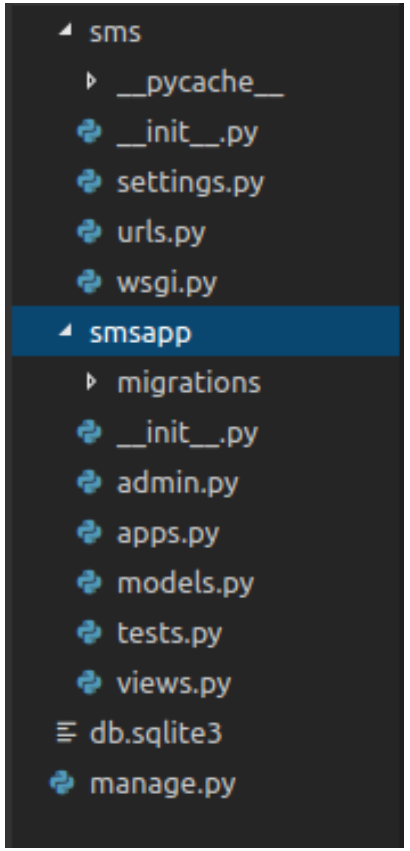
```
/sms$ python manage.py startapp smsapp
/sms$ code .
/sms$
```



Create App

- The command generates the files:
  - **admin**: Specify available commands that will be integrated into the administrative functions of the project that are available within the /admin path
  - **apps**: Is the bootstrap entry point for the smsapp application
  - **models**: is the files where the data models of the application are traditionally defined and referenced from
  - **tests**: allow the specification of test cases from the application
  - **views**: the generation of views that will load the appropriate data and respective content to be displayed

```
/sms$ python manage.py startapp smsapp
/sms$ code .
/sms$
```



Create App

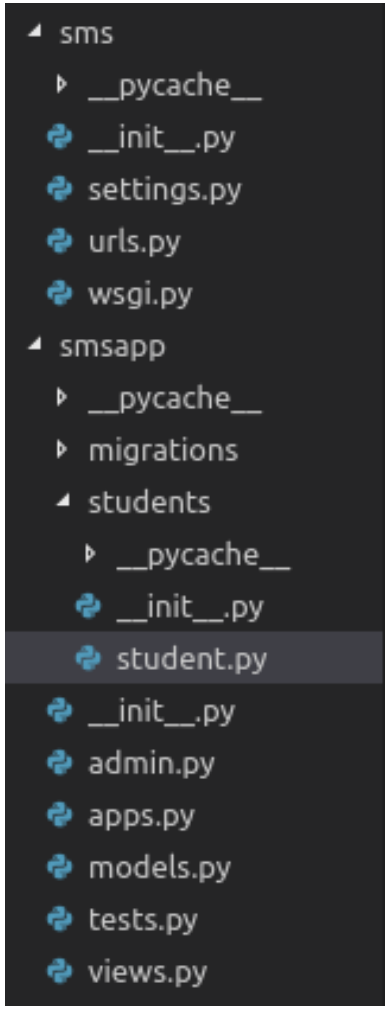
- After creating the app we are required to add our app to the project by modifying the array of INSTALLED_APPS for the project

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

Building Models

Add Model

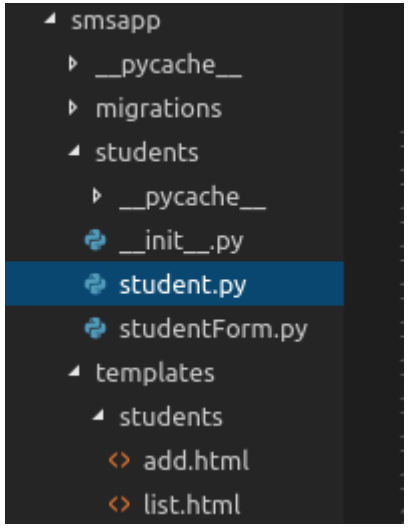
- In our example we created our own folder called “students” to store information related to the students.
- This will help us to have better separation and modularity of our code
- In this folder we created:
  - __init__.py used to specify that this folder should be treated as a module
  - student.py used to develop the student model class



Add Model

- Within the student.py, we provide the imports needed:

```
import uuid
from django.db import models
from django.db.models import (
  UUIDField,
  CharField,
  IntegerField,
  BooleanField,
  DateTimeField
)
```



student.py

```
1 import uuid
2 from django.db import models
3 from django.db.models import (
4   UUIDField,
5   CharField,
6   IntegerField,
7   BooleanField,
8   DateTimeField
9 )
```

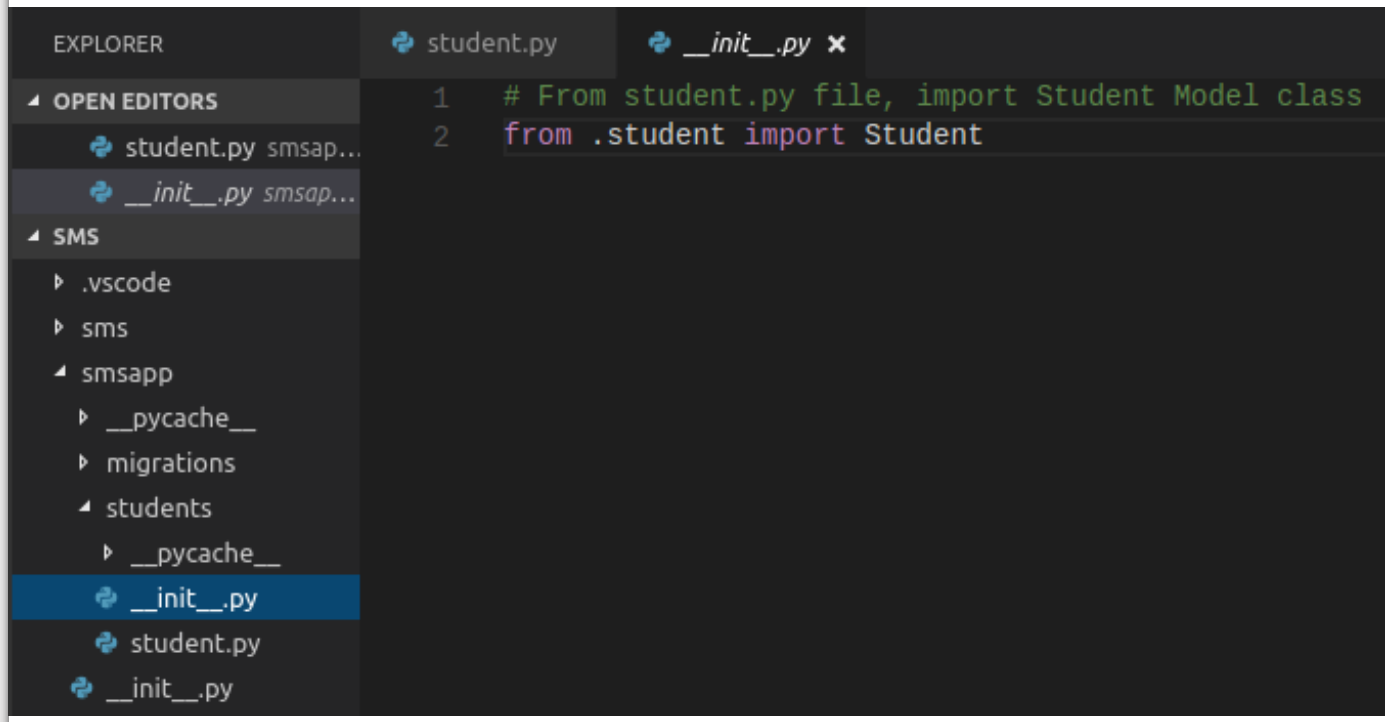
Add Model

- We then create the student class that extends the Model
- We utilize a UUID field rather than an sequential ID number

```
class Student(models.Model):
    id = UUIDField(primary_key=True, default=uuid.uuid4, editable=False)
    # required field
    name = CharField(max_length=100)
    # Required with default
    countrycode = CharField(max_length=3, blank=False, default='TTO')
    # Is student current
    isActive = BooleanField(default=False)
    # year entered (required)
    started = IntegerField()
    # timestamp
    created = DateTimeField(auto_now_add=True)
    # Choices
    # define constants
    UNDERGRAD = "UG"
    POSTGRAD = "PG"
    LEVEL_CHOICE = (
        (UNDERGRAD, "Undergraduate"),
        (POSTGRAD, "Postgraduate")
    )
    level = CharField(
            max_length=2,
            choices=LEVEL_CHOICE,
            default=UNDERGRAD,
        )

    class Meta:
        ordering = ('created',)
```

Add Model

- Add the class we created as part of the student module that we are creating by adding the class to the __init__.py file
- This makes importing from other parts of the application easier
- We load the student class from the student file.
- The .student means the student file is in the same directory as the init file



Add Model

- We add the Model we created within the smsapp models.py folder

student.py models.py

```python
from django.db import models


# From student packages, import Student Model class
from smsapp.students import Student
```

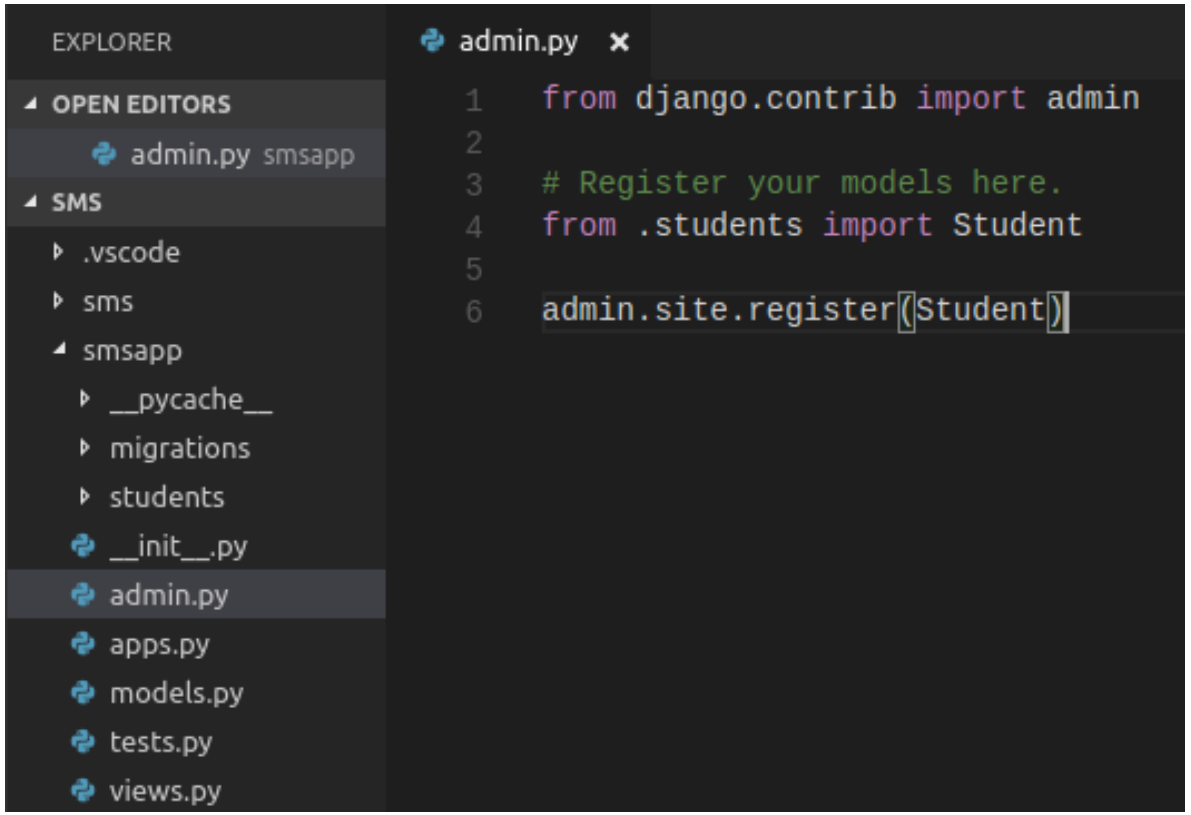
Add Model

- After we create the model and then we initiate the migration

```
(venv) adminuser@adminuser-virt:~/dev/python-sms/sms$ python manage.py makemigrations smsapp
Migrations for 'smsapp':
  smsapp/migrations/0001_initial.py
    - Create model Student
(venv) adminuser@adminuser-virt:~/dev/python-sms/sms$ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, sessions, smsapp
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying auth.0008_alter_user_username_max_length... OK
  Applying sessions.0001_initial... OK
  Applying smsapp.0001_initial... OK
```

Add Model

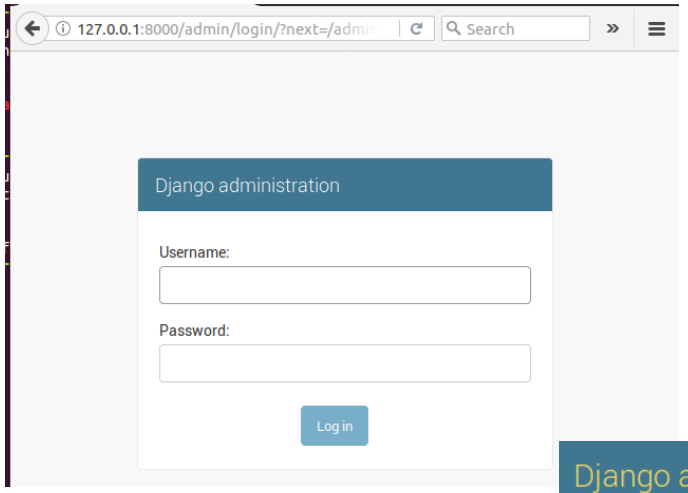
- We then add the Student Model to the administrative functionality
- This will give us the ability to manage records from the administrative interface

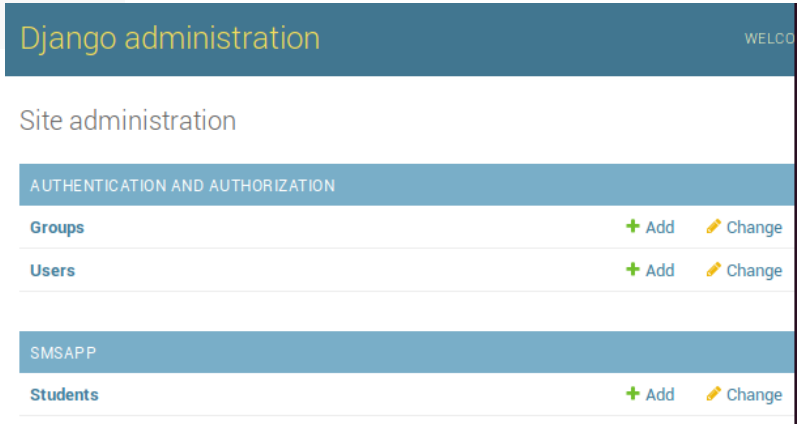


Add Model

- After created a new super user
- We login and in the administration capabilities we see the student model we specified previously

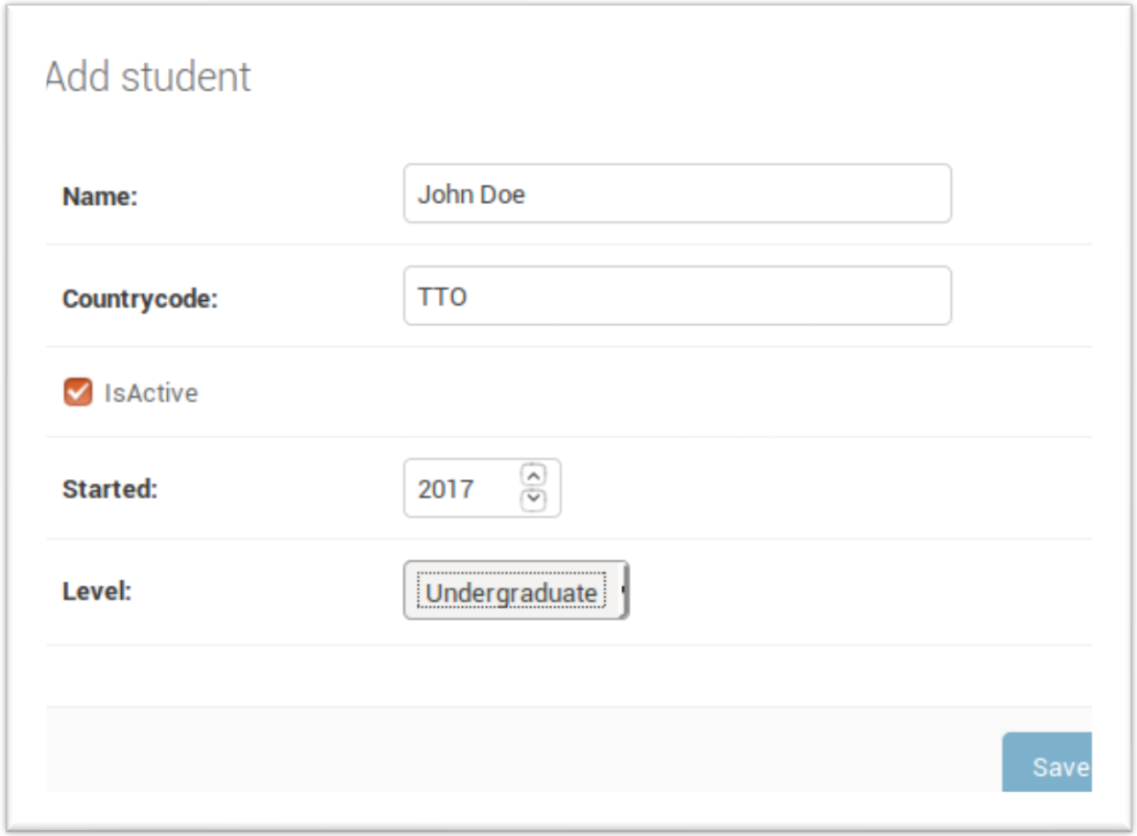
```
(venv) adminuser@adminuser-virt:~/dev/python-sms/sms$ python manage.py createsuperuser
Username (leave blank to use 'adminuser'): admin
Email address: test@test.com
Password:
Password (again):
Superuser created successfully.
(venv) adminuser@adminuser-virt:~/dev/python-sms/sms$
```





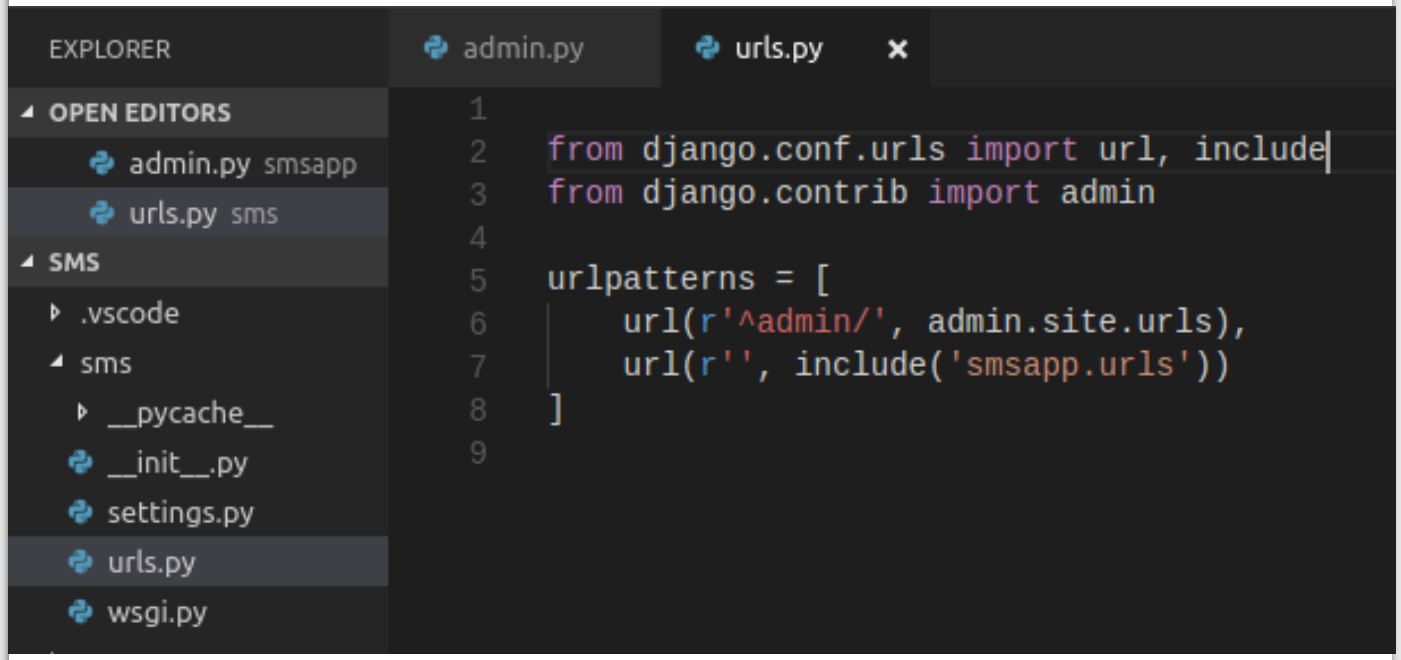
Add Model

- Add the data to the administrative interface form when we click the add option



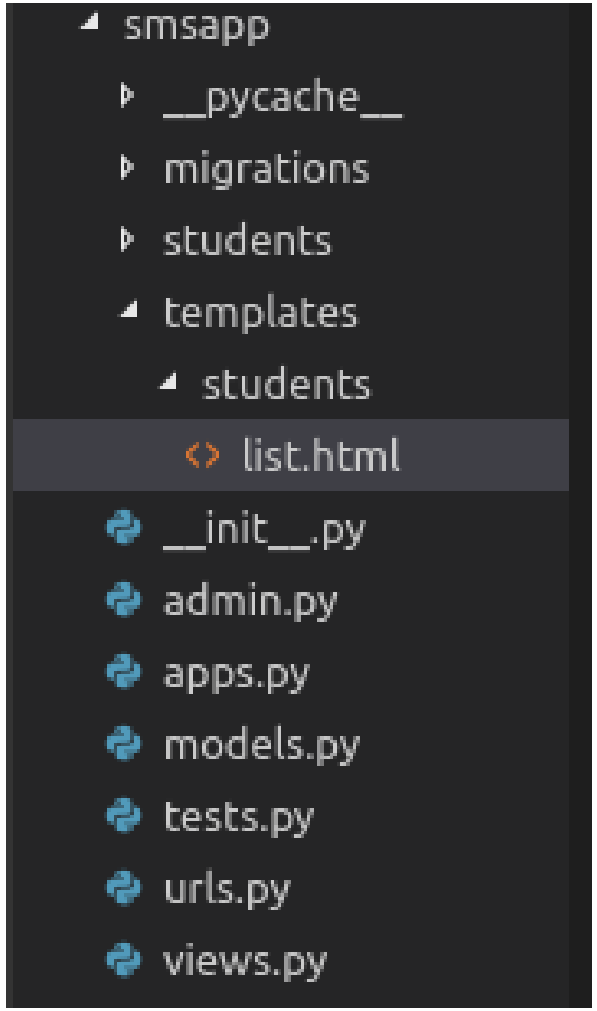
Add Model

- We want to add paths to view and manage the data associated with out smsapp.
- We edit the urls.py of the project to redirect all other urls to the application within the project



Add Model

- Create the template file



Add Model

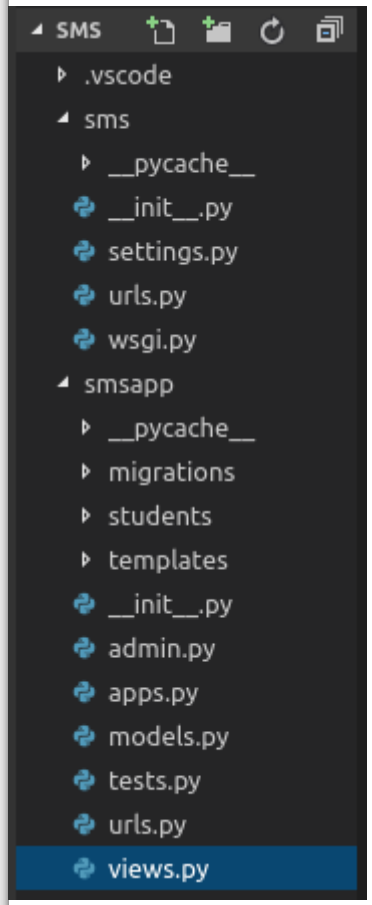
- Write the HTML Code for the template
- We include python related functionality between the {% … %} tags

list.html × | urls.py smsapp | views.py | urls.py sms

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Document</title>
</head>
<body>
  <h2>Students</h2>
  <ul>
    {% for student in students %}
      <li>{{ student.name }}</li>
    {% endfor %}
  </ul>
</body>
</html>
```

Add Model

- Specify the view that will:
  - load data from database
  - Load the html from the template
  - Associate the data from models and the template

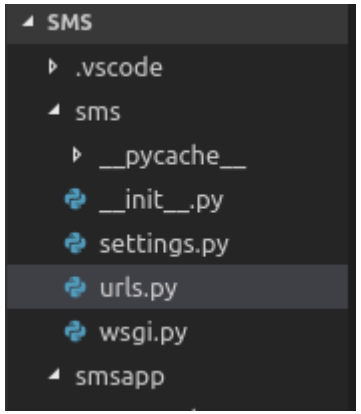


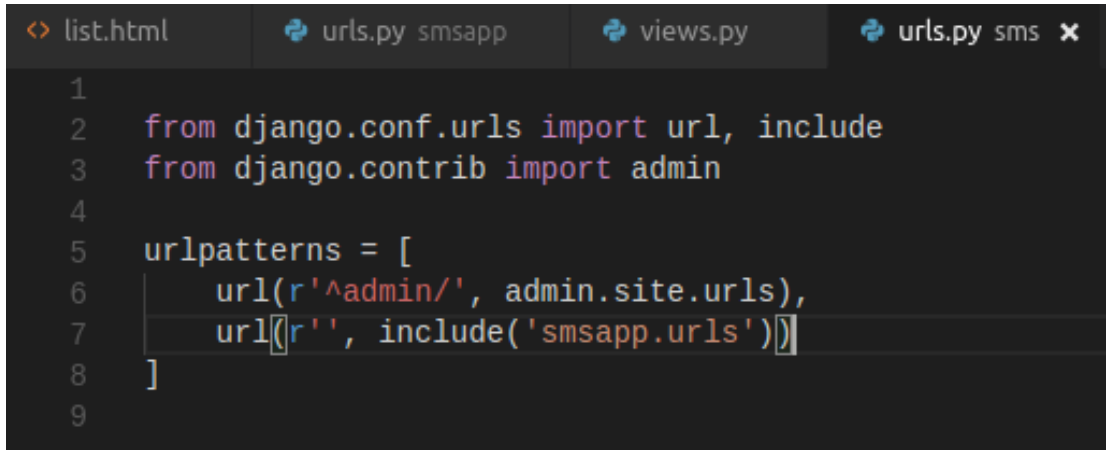
list.html | urls.py smsapp | views.py | urls.py sms

```python
from django.shortcuts import render
from .students import Student

def full_student_listing(request):
    allStudents = Student.objects.all()
    return render(request, 'students/list.html', {
        'students': allStudents
    })
```

Add Model

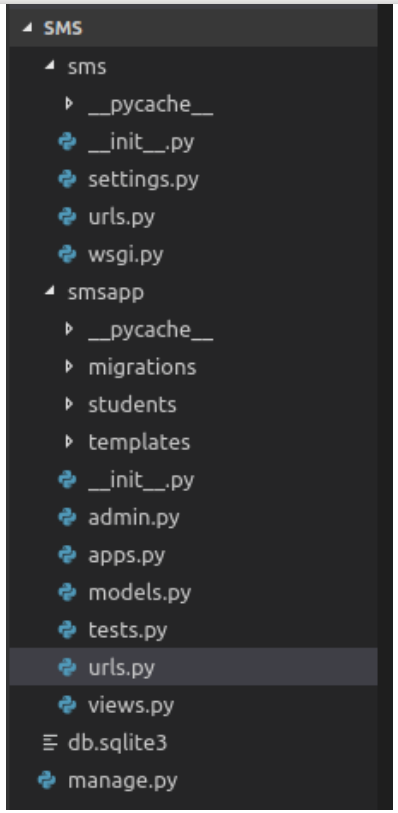
- Specify the view that will:
  - Within the project, we specify that the urls should be loaded from our smsapp application.
  - So the project ('sms') urls is passed to the application ('smsapp')





Add Model

- Specify the view that will:
  - Within the app ('smsapp') we will create a urls.py file and specify the route and its related view
  - In the urls we specify the route.
  - In this example we specify that the default url will load the view called 'full_student_listing'

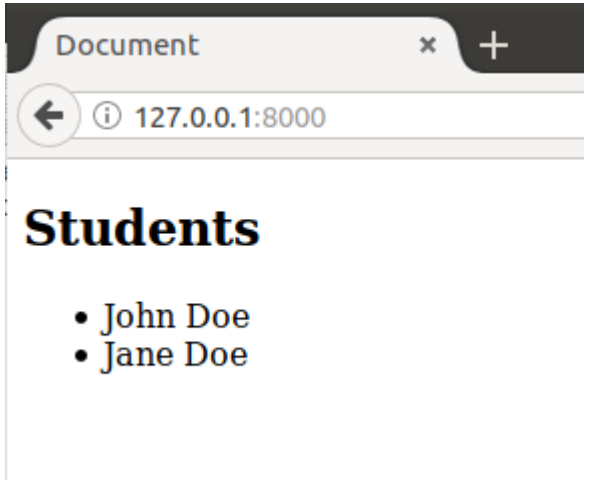


urls.py

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

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

Add Model

- Once everything is linked, the code should appear with a listing of all students



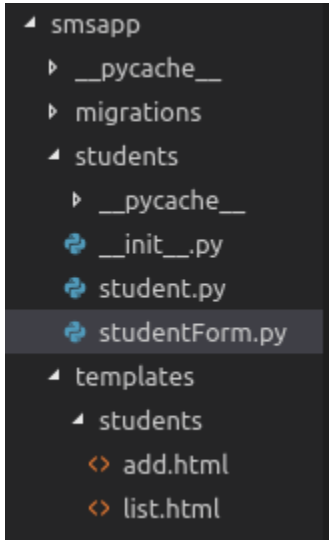
Activity 1

- Develop a model and related page for courses
- The course model will have the fields
  - id (uuid)
  - name (String)
  - credit (Integer) [default 3]
  - isActive (boolean) [default true]
  - created (date)
- Ensure to use the admin interface to create at least two courses
- Put the listing of courses on the same page with the students

Adding ModelForms

Add ModelForm

- Within our student folder we create a StudentForm class that extends the ModelForm
- We specify the fields by telling the system to exclude no fields (which is the same as saying use all fields)

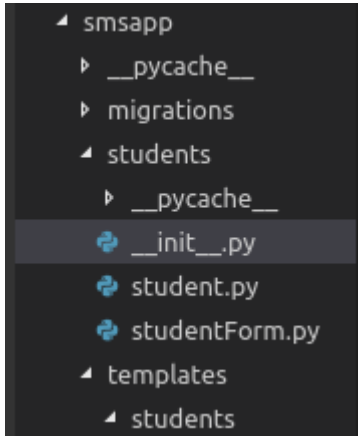


studentForm.py × urls.py views.py

```python
from django.forms import ModelForm
from .student import Student

class StudentForm(ModelForm):
    class Meta:
        model = Student
        exclude = ()
```

Add ModelForm

- Add the class we created as part of the student module that we are creating by adding the class to the __init__.py file
- This makes importing from other parts of the application easier

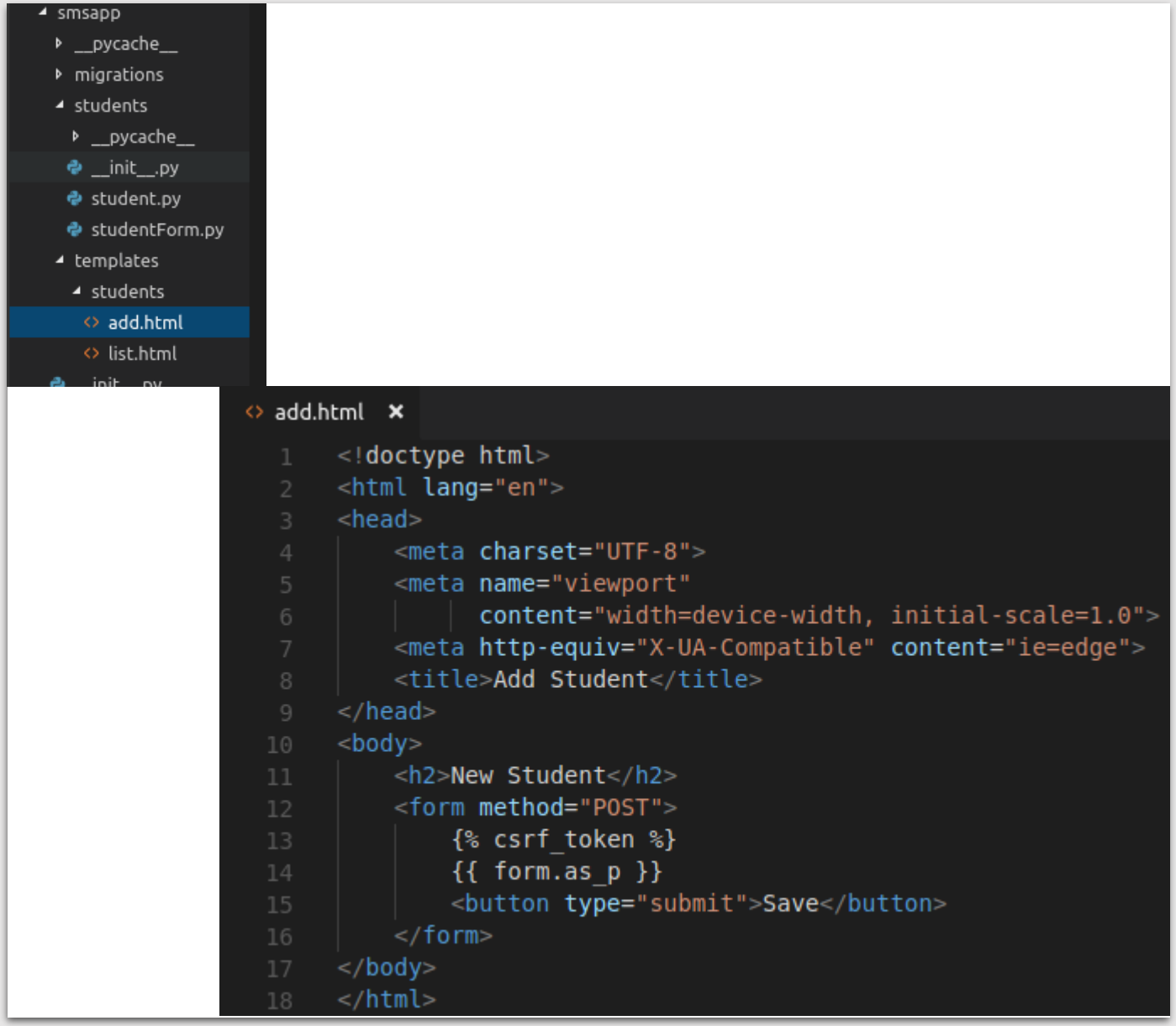


studentForm.py | __init__.py | urls.py | views.py

```
# From student.py file, import Student Model class
from .student import Student
from .studentForm import StudentForm
```

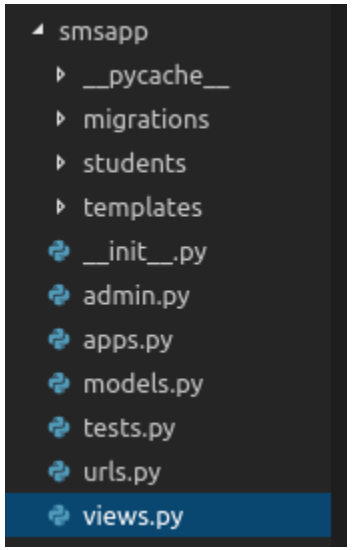
Add ModelForm

- We add the fields of the form to the html
- We build a template ‘add.html’ that will utilize the form ‘StudentForm’ that we just created
- The student form is a local variable in the template called “form”. The as p will load the fields of the forms as paragraphs. For others consider the [documentation]
  - Tables
  - Paragraphs
  - Lists



Add ModelForm

- Add the new function add_student to display the form defined within the add.html file

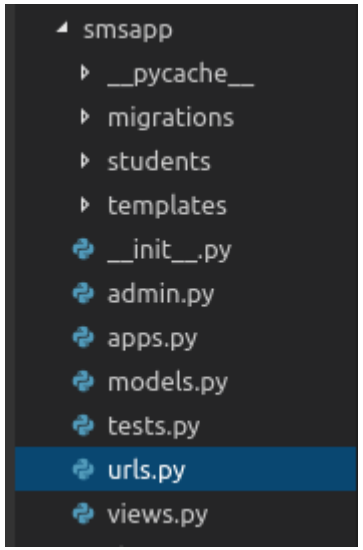


studentForm.py | add.html | views.py

```python
from django.shortcuts import render
from .students import Student, StudentForm

def full_student_listing(request):
    allStudents = Student.objects.all()
    return render(request, 'students/list.html', {
        'students': allStudents
    })

def  add_student(request):
    form  = StudentForm()
    return render(request, 'students/add.html', {'form': form})
```

Add ModelForm

- We associate the view with the URL pattern

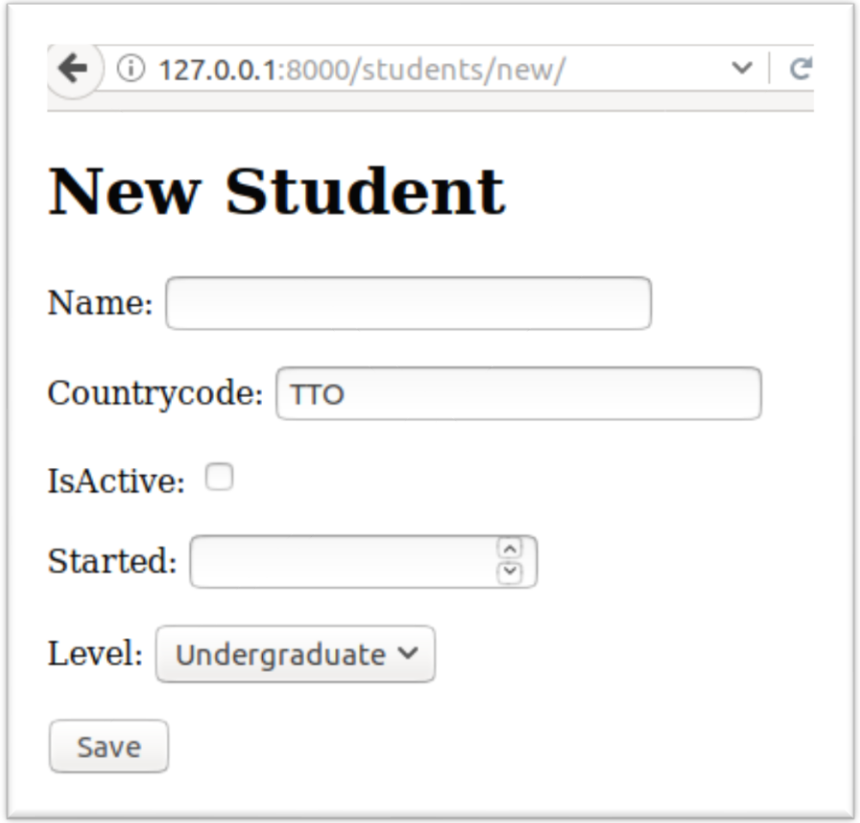


studentForm.py | add.html | urls.py

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

urlpatterns = [
    url(r'^$', views.full_student_listing, name='student_full_list'),
    url(r'^students/new/$', views.add_student, name='student_new'),
]
```

Add ModelForm

- When we load the URL to see the html that we created
- Note however, that this page cannot save the information



Activity 2

- Create the Form and the associated ModelForm for the course Model created in Activity 1
- Assign the route /courses/new to the display of the form to create a new course

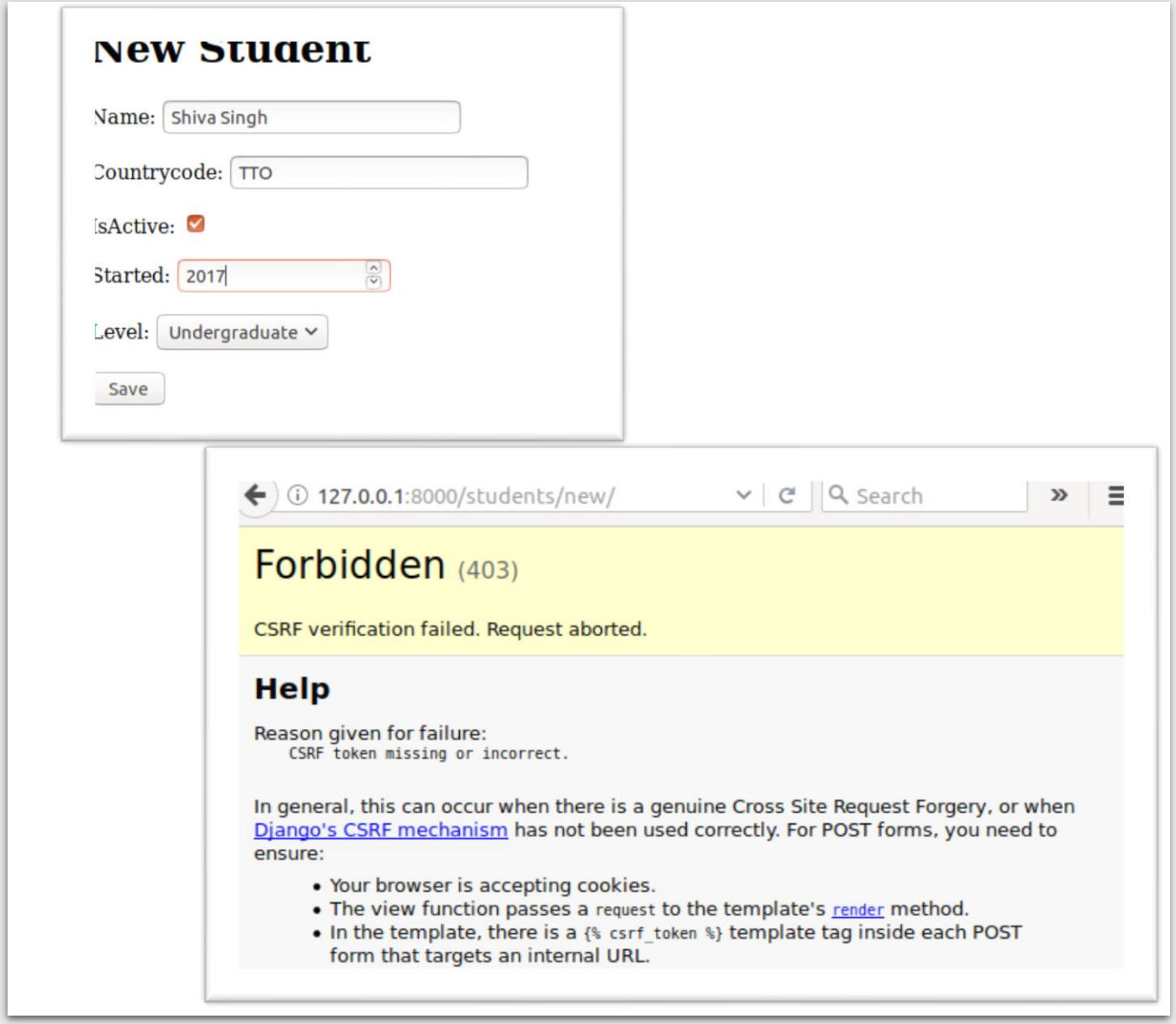
Add ModelForm

- We modify the add_student function to handle the request made via the form.
- Usually the action attribute in the form would specify the location where the information is submitted, however, since it was left out, the data will be sent to the same url used to display the form.
- So we put in a condition to handle the POST request, retrieve the enclosed data, ensure data is valid (based on rules defined in model) and then save information
- If the save operation is successful then we redirect to the home page

```
from django.shortcuts import render
from django.http import HttpResponseRedirect
from .students import Student, StudentForm

def full_student_listing(request):
  allStudents = Student.objects.all()
  return render(request, 'students/list.html', {
      'students': allStudents
    })

def  add_student(request):
  stuForm  = StudentForm()

  if request.POST:
    stuForm = StudentForm(request.POST)
    if stuForm.is_valid():
      stuForm.save()
      return HttpResponseRedirect('/')

  return render(request, 'students/add.html', {'form': stuForm})
```

Add ModelForm

- When we add the data and submit the form as is, we will get a Forbidden error.
- From this error, we can inspect the error message displayed on the page to determine the source of the error and a possible solution to the error



Add ModelForm

- We specify the csrf token within the form as required by the error
- This token is a security measure enforced by the framework to protect against Cross-Site Request Forgery attacks.
- Cross-Site Request Forgery (CSRF) is an attack that forces an end user to execute unwanted actions on a web application in which they're currently authenticated. (source)

studentForm.py | add.html | views.py

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Add Student</title>
</head>
<body>
    <h1>New Student</h1>
    <form method="POST" class="post-form">
        {% csrf_token %}
        {{ form.as_p }}
        <button type="submit" class="save btn btn-default">Save</button>
    </form>
</body>
</html>
```

Add ModelForm

- We modify the view to handle the data submitted
- Recall that the view accepts the request and generates a response based on the result of process

```python
def add_student(request):
    form = StudentForm()
    # The form when submmitted will make a POST request to the URL
    if request.method == 'POST':
        # Populate the form with the data submitted via the request
        form = StudentForm(request.POST)
        # the validation rules is defined between the form and the model
        if form.is_valid():
            # We extract the cleaned data to protect against vulnerabilities
            student = Student()
            student.name = form.cleaned_data['name']
            student.countrycode = form.cleaned_data['countrycode']
            student.isActive = form.cleaned_data['isActive']
            student.started = form.cleaned_data['started']
            student.level = form.cleaned_data['level']
            # After retrieving the cleaned data, save record
            student.save()
            # If saved successful then redirect to home page
            return HttpResponseRedirect("/")

    return render(request, 'students/add.html', {'form': form})
```

Activity 3

- (Required) Complete the save operation for the courses form created from Activity 2.

Reference

1. Modern Django Development – Dylan Stein – Part 0, Part 1, Part 2
2. https://www.digitalocean.com/community/tutorials/how-to-install-python-3-and-set-up-a-local-programming-environment-on-ubuntu-16-04
3. https://docs.djangoproject.com/en/1.10/topics/http/middleware/
4. https://docs.djangoproject.com/en/1.10/ref/settings/#templates

Recommended Resource

- https://github.com/DonJayamanne/pythonVSCode