

# Syllabus

## Django Framework

- Installation
- Configuration
- Model-View-Controller Architecture
- Urls and Views
- Sessions and Cookies
- Database Connectivity
- Form Processing
- Model Forms
- User Registration and Authentication
- Introduction to REST Framework:- REST API, JSON module,Serialization

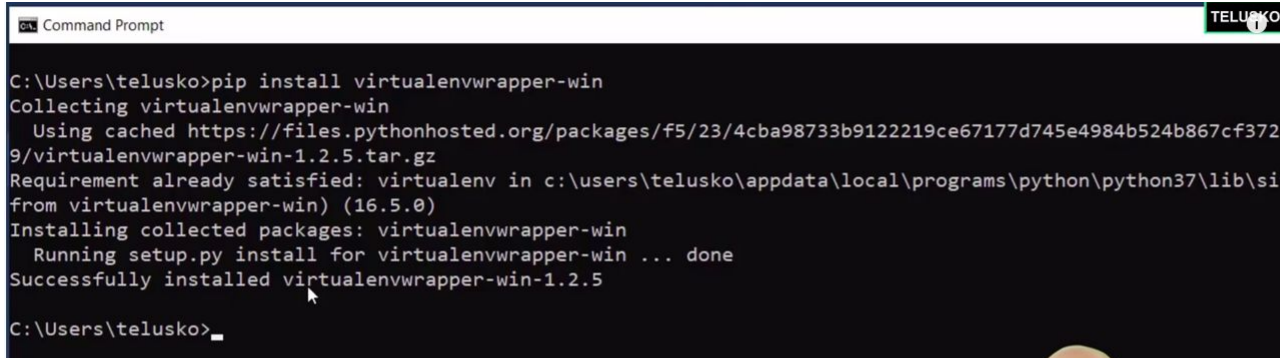
# Django Framework

Class notes

# Installation

1. Install python
2. Install pip
3. Install django

i)set up virtual environment



```
Command Prompt

C:\Users\telusko>pip install virtualenvwrapper-win
Collecting virtualenvwrapper-win
  Using cached https://files.pythonhosted.org/packages/f5/23/4cba98733b9122219ce67177d745e4984b524b867cf3729/virtualenvwrapper-win-1.2.5.tar.gz
Requirement already satisfied: virtualenv in c:\users\telusko\appdata\local\programs\python\python37\lib\site-packages (from virtualenvwrapper-win) (16.5.0)
Installing collected packages: virtualenvwrapper-win
  Running setup.py install for virtualenvwrapper-win ... done
Successfully installed virtualenvwrapper-win-1.2.5

C:\Users\telusko>
```

## ii)install virtual environment

```
C:\Users\telusko>mkvirtualenv test
Using base prefix 'c:\\users\\telusko\\appdata\\local\\programs\\python\\python37'
New python executable in C:\Users\telusko\Envs\test\Scripts\python.exe
Installing setuptools, pip, wheel...
done.
```

Here is the virtual env name is “test”

## iii)install Django

```
(test) C:\Users\telusko>pip install django
Collecting django
  Using cached https://files.pythonhosted.org/packages/54/85/0bef63668fb170888c1a2970ec897d4528...2dee27653381a332...
2/Django-2.2-py3-none-any.whl
Collecting sqlparse (from django)
  Using cached https://files.pythonhosted.org/packages/ef/53/900f7d2a54557c6a37886585a91336520e5...e2423ff1102daf4...
7/sqlparse-0.3.0-py2.py3-none-any.whl
Collecting pytz (from django)
  Using cached https://files.pythonhosted.org/packages/3d/73/fe30c2daaaa0713420d0382b16f...1bf7b6262...
6/pytz-2019.1-py2.py3-none-any.whl
Installing collected packages: sqlparse, pytz, django
```

To check the version of django

```
>>> django-admin --version
```

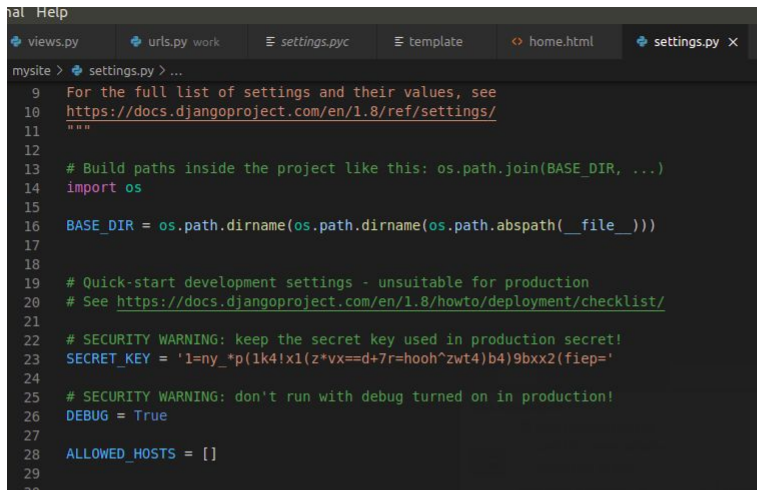
# Django settings

A Django settings file contains all the configuration of your Django installation. This document explains how settings work and which settings are available.

## The basics

A settings file is just a Python module with module-level variables.

Here are a couple of example settings:

A screenshot of a code editor window showing the Django settings.py file. The editor has a dark theme and several tabs at the top: 'views.py', 'urls.py work', 'settings.pyc', 'template', 'home.html', and 'settings.py' (which is the active tab). The code in the settings.py file is as follows:

```
mysite > settings.py > ...
9  For the full list of settings and their values, see
10 https://docs.djangoproject.com/en/1.8/ref/settings/
11  """
12
13  # Build paths inside the project like this: os.path.join(BASE_DIR, ...)
14  import os
15
16  BASE_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
17
18
19  # Quick-start development settings - unsuitable for production
20  # See https://docs.djangoproject.com/en/1.8/howto/deployment/checklist/
21
22  # SECURITY WARNING: keep the secret key used in production secret!
23  SECRET_KEY = 'l=ny_*p(1k4!x1(z*vx==d+7r=hooH^zwt4)b4)9bxx2(fiep='
24
25  # SECURITY WARNING: don't run with debug turned on in production!
26  DEBUG = True
27
28  ALLOWED_HOSTS = []
29
30
```

### Note

If you set `DEBUG` to `False`, you also need to properly set the `ALLOWED_HOSTS` setting.

Because a settings file is a Python module, the following apply:

- It doesn't allow for Python syntax errors.
- It can assign settings dynamically using normal Python syntax. For example:

```
MY_SETTING = [str(i) for i in range(30)]
```

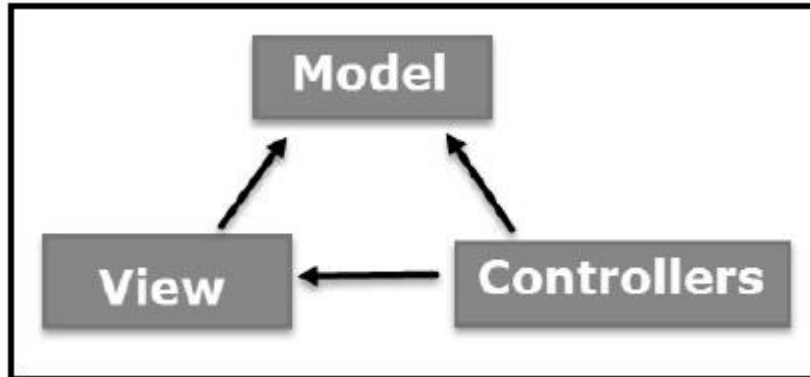
- It can import values from other settings files.

# Model-View-Controller Architecture

The Model-View-Controller (MVC) is an architectural pattern that separates an application into three main logical components: the model, the view, and the controller. Each of these components are built to handle specific development aspects of an application. MVC is one of the most frequently used industry-standard web development framework to create scalable and extensible projects.

## MVC Components

Following are the components of MVC –



## Model (is the data access layer)

The Model component corresponds to all the data-related logic that the user works with. This can represent either the data that is being transferred between the View and Controller components or any other business logic-related data. For example, a Customer object will retrieve the customer information from the database, manipulate it and update it data back to the database or use it to render data.

## View( is the business logic layer and)

The View component is used for all the UI logic of the application. For example, the Customer view will include all the UI components such as text boxes, dropdowns, etc. that the final user interacts with.

## Controller ( is the presentation layer)

Controllers act as an interface between Model and View components to process all the business logic and incoming requests, manipulate data using the Model component and interact with the Views to render the final output. For example, the Customer controller will handle all the interactions and inputs from the Customer View and update the database using the Customer Model. The same controller will be used to view the Customer data.



# Django Urls Path

Django has a `urls.py` file under the project by default. It also has a pre-defined path for the *admin* app. However, Django recommends mapping all resources via another *urls.py* newly created under the app. The below explains it:

*mysite -- urls.py*

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

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

*myapp -- urls.py*

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

urlpatterns = [
    path('', views.index), # app homepage
]
```

## Django View Function

The URL mapping will redirect requests from project URLs to app URLs and then to the respective view function. A sample view function code may look like this:

```
def index(request):  
    return render(request, 'index.html', {})
```

or,

```
from django.http import HttpResponse  
def index(request):  
    return HttpResponse("Hello World")
```

Here, the request is the URL request mapping and calling the view function. render combines a given template with a given context dictionary. {} denotes the dictionary of values that can be added to the template context.

# Sessions and Cookies

Django provides a session framework that lets you store and retrieve data on a per-site-visitor basis. Django abstracts the process of sending and receiving cookies, by placing a session ID cookie on the client side, and storing all the related data on the server side. So the data itself is not stored client side.

## Setting up Sessions

To enable the session functionality, you'll need to make sure that the `MIDDLEWARE_CLASSES` in `settings.py` has `'django.contrib.sessions.middleware.SessionMiddleware'` activated.

There are different ways in which you can configure the session engine which controls how it stores sessions, i.e. in a database or in a cache. The simplest way is to use the default option and store the data in a model/database (i.e. `django.contrib.sessions.models.Session`), is by adding `'django.contrib.sessions'` to your `INSTALLED_APPS`.

You will have to run *`python manage.py syncdb`* to create the session database table.

## Testing Cookies

To test out whether cookies work on your client, you can use some convenience methods provided by Django's request object (`set_test_cookie()`, `test_cookie_worked()`, `delete_test_cookie()` ). In one view you will need to set a cookie and in another view you'll need to test it. The reason you need to set the cookie in one view and test it in another is that you need to wait to see if the client has actually accepted the cookie.

In `views.py`, set the test cookie in the *index* view, and test the cookie in your *about* view.

```
def index(request):  
    ...  
    request.session.set_test_cookie()  
    ...  
  
def about(request):  
    ...  
    if request.session.test_cookie_worked():  
        print "The test cookie worked!!!"  
        request.session.delete_test_cookie()  
    ...
```

---

## Add and Using Cookies

Everyone loves knowing how many times they have visited a site. Hmmm, maybe not, but let's add that functionality now using cookies as an example.

To do this you'll need two cookies. One to track the number of times, and one to keep track of when the user last visited the site, so that you can increment the visit count only once per day.

```
def index(request):  
    ...  
    ...  
  
    visits = int( request.COOKIE.get('visits', '0') )  
  
    response = HttpResponseRedirect(template.render(context))  
  
    if request.COOKIE.has_key('last_visit'):  
        last_visit = request.COOKIE['last_visit']  
        # the cookie is a string - convert back to a datetime type  
        last_visit_time = datetime.strptime(last_visit[:7], "%Y-%m-%d %H:%M:%S")  
        curr_time = datetime.now()  
        if (curr_time-last_visit_time).days > 0:  
            # if at least one day has gone by then inc the visit count.  
            response.set_cookie('visits', visits + 1 )  
            response.set_cookie('last_visit', datetime.now())  
    else:  
        response.set_cookie('last_visit', datetime.now())  
  
    return response
```

# Database Connectivity

## Databases¶

Django officially supports the following databases:

- PostgreSQL
- MariaDB
- MySQL
- Oracle
- SQLite

There are also a number of database backends provided by third parties.

Django attempts to support as many features as possible on all database backends. However, not all database backends are alike, and we've had to make design decisions on which features to support and which assumptions we can make safely.

This file describes some of the features that might be relevant to Django usage. It is not intended as a replacement for server-specific documentation or reference manuals.

## Connection management

Django opens a connection to the database when it first makes a database query. It keeps this connection open and reuses it in subsequent requests. Django closes the connection once it exceeds the maximum age defined by `CONN_MAX_AGE` or when it isn't usable any longer.

In detail, Django automatically opens a connection to the database whenever it needs one and doesn't have one already — either because this is the first connection, or because the previous connection was closed.

At the beginning of each request, Django closes the connection if it has reached its maximum age. If your database terminates idle connections after some time, you should set `CONN_MAX_AGE` to a lower value, so that Django doesn't attempt to use a connection that has been terminated by the database server. (This problem may only affect very low traffic sites.)

At the end of each request, Django closes the connection if it has reached its maximum age or if it is in an unrecoverable error state. If any database errors have occurred while processing the requests, Django checks whether the connection still works, and closes it if it doesn't. Thus, database errors affect at most one request; if the connection becomes unusable, the next request gets a fresh connection.

## Encoding¶

Django assumes that all databases use UTF-8 encoding. Using other encodings may result in unexpected behavior such as “value too long” errors from your database for data that is valid in Django. See the database specific notes below for information on how to set up your database correctly.



## PostgreSQL connection settings

See `HOST` for details.

To connect using a service name from the `connection service file` and a password from the `password file`, you must specify them in the `OPTIONS` part of your database configuration in `DATABASES`:

settings.py



```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'OPTIONS': {
            'service': 'my_service',
            'passfile': '.my_pgpass',
        },
    },
}
```

.pg\_service.conf



```
[my_service]
host=localhost
user=USER
dbname=NAME
port=5432
```

.my\_pgpass



```
localhost:5432:NAME:USER:PASSWORD
```

## Optimizing PostgreSQL's configuration

Django needs the following parameters for its database connections:

- `client_encoding`: 'UTF8',
- `default_transaction_isolation`: 'read committed' by default, or the value set in the connection options (see below),
- `timezone`:
  - when `USE_TZ` is `True`, 'UTC' by default, or the `TIME_ZONE` value set for the connection,
  - when `USE_TZ` is `False`, the value of the global `TIME_ZONE` setting.

If these parameters already have the correct values, Django won't set them for every new connection, which improves performance slightly. You can configure them directly in `postgresql.conf` or more conveniently per database user with `ALTER ROLE`.

Django will work just fine without this optimization, but each new connection will do some additional queries to set these parameters.

# Forms in Django

Unless you're planning to build websites and applications that do nothing but publish content, and don't accept input from your visitors, you're going to need to understand and use forms.

Django provides a range of tools and libraries to help you build forms to accept input from site visitors, and then process and respond to the input.

We've described HTML forms briefly, but an HTML `<form>` is just one part of the machinery required.

In the context of a web application, 'form' might refer to that HTML `<form>`, or to the Django `Form` that produces it, or to the structured data returned when it is submitted, or to the end-to-end working collection of these parts.

## The Django **Form** class

At the heart of this system of components is Django's **Form** class. In much the same way that a Django model describes the logical structure of an object, its behavior, and the way its parts are represented to us, a **Form** class describes a form and determines how it works and appears.

In a similar way that a model class's fields map to database fields, a form class's fields map to HTML form `<input>` elements. (A **ModelForm** maps a model class's fields to HTML form `<input>` elements via a **Form**; this is what the Django admin is based upon.)

A form's fields are themselves classes; they manage form data and perform validation when a form is submitted. A **DateField** and a **FileField** handle very different kinds of data and have to do different things with it.

A form field is represented to a user in the browser as an HTML "widget" - a piece of user interface machinery. Each field type has an appropriate default Widget class, but these can be overridden as required.

## Building a form

### The work that needs to be done

Suppose you want to create a simple form on your website, in order to obtain the user's name. You'd need something like this in your template:

```
<form action="/your-name/" method="post">
  <label for="your_name">Your name: </label>
  <input id="your_name" type="text" name="your_name" value="{{ current_name }}">
  <input type="submit" value="OK">
</form>
```

This tells the browser to return the form data to the URL `/your-name/`, using the `POST` method. It will display a text field, labeled “Your name:”, and a button marked “OK”. If the template context contains a `current_name` variable, that will be used to pre-fill the `your_name` field.

You’ll need a view that renders the template containing the HTML form, and that can supply the `current_name` field as appropriate.

When the form is submitted, the `POST` request which is sent to the server will contain the form data.

Now you’ll also need a view corresponding to that `/your-name/` URL which will find the appropriate key/value pairs in the request, and then process them.

This is a very simple form. In practice, a form might contain dozens or hundreds of fields, many of which might need to be pre-populated, and we might expect the user to work through the edit-submit cycle several times before concluding the operation.

We might require some validation to occur in the browser, even before the form is submitted; we might want to use much more complex fields, that allow the user to do things like pick dates from a calendar and so on.

At this point it’s much easier to get Django to do most of this work for us.

## Building a form in Django

### The Form class

We already know what we want our HTML form to look like. Our starting point for it in Django is this:

forms.py



```
from django import forms

class NameForm(forms.Form):
    your_name = forms.CharField(label='Your name', max_length=100)
```



his defines a `Form` class with a single field (`your_name`). We've applied a human-friendly label to the field, which will appear in the `<label>` when it's rendered (although in this case, the `label` we specified is actually the same one that would be generated automatically if we had omitted it).

The field's maximum allowable length is defined by `max_length`. This does two things. It puts a `maxlength="100"` on the HTML `<input>` (so the browser should prevent the user from entering more than that number of characters in the first place). It also means that when Django receives the form back from the browser, it will validate the length of the data.

A `Form` instance has an `is_valid()` method, which runs validation routines for all its fields. When this method is called, if all fields contain valid data, it will:

- return `True`
- place the form's data in its `cleaned_data` attribute.

The whole form, when rendered for the first time, will look like:

```
<label for="your_name">Your name: </label>
<input id="your_name" type="text" name="your_name" maxlength="100" required>
```

Note that it **does not** include the `<form>` tags, or a submit button. We'll have to provide those ourselves in the template.

# Creating forms from models

## ModelForm

```
class ModelForm
```

If you're building a database-driven app, chances are you'll have forms that map closely to Django models. For instance, you might have a `BlogComment` model, and you want to create a form that lets people submit comments. In this case, it would be redundant to define the field types in your form, because you've already defined the fields in your model.

For this reason, Django provides a helper class that lets you create a `Form` class from a Django model.

For example:

```
>>> from django.forms import ModelForm
>>> from myapp.models import Article

# Create the form class.
>>> class ArticleForm(ModelForm):
...     class Meta:
...         model = Article
...         fields = ['pub_date', 'headline', 'content', 'reporter']

# Creating a form to add an article.
>>> form = ArticleForm()

# Creating a form to change an existing article.
>>> article = Article.objects.get(pk=1)
>>> form = ArticleForm(instance=article)
```

## User Registration and Authentication

# User registration

As you may have seen, Django comes with a built-in user registration form. We just need to configure it to our needs (i.e. collect an email address upon registration).

## Create the register form

*env > mysite > main > (New File) forms.py*

```
from django import forms
from django.contrib.auth.forms import UserCreationForm
from django.contrib.auth.models import User

# Create your forms here.

class NewUserForm(UserCreationForm):
    email = forms.EmailField(required=True)

    class Meta:
        model = User
        fields = ("username", "email", "password1", "password2")

    def save(self, commit=True):
        user = super(NewUserForm, self).save(commit=False)
        user.email = self.cleaned_data['email']
        if commit:
            user.save()
        return user
```

Django comes with a pre-built register form called `UserCreationForm` that connects to the pre-built model `User`. However, the `UserCreationForm` only requires a username and password (`password1` is the initial password and `password2` is the password confirmation).

To customize the pre-built form, first create a new file called `forms.py` in the app directory.

This new file is created in the same directory as `models.py` and `views.py`.

Then call `UserCreationForm` within a new class called `NewUserForm` and add another field called `email`. Save the email to the user.

Add more fields as needed to the `UserCreationForm`.

For more reference

<https://ordinarycoders.com/blog/article/django-user-register-login-logout>

Introduction to REST Framework:- REST API, JSON module,Serialization

<https://www.django-rest-framework.org/tutorial/1-serialization/>