**Introduction python with Django**

Python uses a package manager called *pip*

**Update pip package manager**

pip install --upgrade pip

**Install virtualenv (Optional Prerequisite)**

Virtualenv is not essential to develop Django applications, but I highly recommend you use it because it allows you to create virtual Python environments on a single system.

What happens if a new Django version is released after your first project and you want to start a second

project? Do you upgrade the first project to run on the new Django version or start the second project as if

the new Django version doesn’t exist? The first option requires additional work, while the second option

requires you to develop on an outdated Django version. By using virtual Python environments you avoid this

problem, because each project can run its own Django version in isolation.

**Set environment variable**

PATH :- C:\Users\techvision\AppData\Local\Programs\Python\Python36-32\pythonw.exe;

C:\Users\techvision\AppData\Local\Programs\Python\Python36-32\Lib\idlelib\idle.pyw;

C:\Users\techvision\AppData\Local\Programs\Python\Python36-32\Scripts

After installation, open the command prompt and check that the Python version matches the version you installed by executing:

python –version

## **Install virtualenv and virtualenvwrapper**[¶](https://docs.djangoproject.com/en/2.0/howto/windows/#install-virtualenv-and-virtualenvwrapper)

pip install virtualenvwrapper-win

**Then create a virtual environment for your project:**

mkvirtualenv myproject

The virtual environment will be activated automatically and you’ll see “(myproject)” next to the command prompt to designate that. If you start a new command prompt, you’ll need to activate the environment again using:

workon myproject

## **Install Django**[¶](https://docs.djangoproject.com/en/2.0/howto/windows/#install-django)

Django can be installed easily using **pip** within your virtual environment.

In the command prompt, ensure your virtual environment is active, and execute the following command:

pip install django

Django web applications typically group the code that handles each of these steps into separate files:



* **URLs:**While it is possible to process requests from every single URL via a single function, it is much more maintainable to write a separate view function to handle each resource. A URL mapper is used to redirect HTTP requests to the appropriate view based on the request URL. The URL mapper can also match particular patterns of strings or digits that appear in an URL, and pass these to a view function as data.
* **View:** A view is a request handler function, which receives HTTP requests and returns HTTP responses. Views access the data needed to satisfy requests via *models*, and delegate the formatting of the response to *templates*.
* **Models:** Models are Python objects that define the structure of an application's data, and provide mechanisms to manage (add, modify, delete) and query records in the database.
* **Templates:** A template is a text file defining the structure or layout of a file (such as an HTML page), with placeholders used to represent actual content. A *view* can dynamically create an HTML page using an HTML template, populating it with data from a *model*. A template can be used to define the structure of any type of file; it doesn't have to be HTML!

**Note**: Django refers to this organisation as the "Model View Template (MVT)" architecture. It has many similarities to the more familar [Model View Controller](https://developer.mozilla.org/en-US/docs/Web/Apps/Fundamentals/Modern_web_app_architecture/MVC_architecture)architecture.

Sending the request to the right view (urls.py)

A URL mapper is typically stored in a file named **urls.py**.

the mapper  (urlpatterns) defines a list of mappings between routes (specific URL patterns) and corresponding view functions. If an HTTP Request is received that has a URL matching a specified pattern then the associated view function will be called and passed the request.

urlpatterns = [

path('admin/', admin.site.urls),

path('book/<int:id>/', views.book-detail, name='book-detail'),

path('catalog/', include('catalog.urls')),

re\_path(r'^([0-9]+)/$', views.best),

]

**Create project:**

Create a project called mysite.

django-admin startproject mysite

\* To start a Django project you must use the django-admin executable or django-admin.py script that comes with Django.

\* A Django project name can be composed of numbers, letters, or underscores.

Django project structure

+<BASE\_DIR\_project\_name>

|

+----manage.py

|

+---+-<PROJECT\_DIR\_project\_name>

|

+-\_\_init\_\_.py

+-settings.py

+-urls.py

+-wsgi.py

• manage.py .- Runs project specific tasks. Just as django-admin is used to execute system wide Django tasks, manage.py is used to execute project specific tasks.

• \_\_init\_\_.py .- Python file that allows Python packages to be imported from directories where it’s present. Note \_\_init\_\_.py is not Django specific, it’s a generic file used in almost all

Python applications.

• settings.py .- Contains the configuration settings for the Django project.

• urls.py .- Contains URL patterns for the Django project.

• wsgi.py .- Contains WSGI configuration properties for the Django project. WSGI is the recommended approach to deploy Django applications on production (i.e., to the public). You

don’t need to set up WSGI to develop Django applications.

cd mysite

**Create App:**

django-admin startapp blog

**Start server:**

python manage.py runserver

Edit and add following files:

1. Mysiit : urls.py
2. from django.contrib import admin
3. from django.urls import path
4. from django.conf.urls import url,include
5. urlpatterns = [
6. path('admin/', admin.site.urls),
7. #path('blog/', include('blog.urls')),
8. url(r'^blog/', include('blog.urls')),
10. ]

2. Add urls.py in blog app

from django.urls import path

from . import views

urlpatterns = [

path('', views.index, name='index'),

]

3. edit and add in view file within blog app

from django.http import HttpResponse

def index(request):

return HttpResponse("Hello, world. You're at the polls index.")

**Connection with Data Base**

**Database Django** ENGINE **value**

MySQL django.db.backends.mysql

Oracle django.db.backends.oracle

PostgreSQL django.db.backends.postgresql\_psycopg2

SQLite django.db.backends.sqlite3

**Django connection parameter Default value Notes**

|  |  |  |
| --- | --- | --- |
| ATOMIC\_REQUESTS | False | Enforces (or not) a transaction for each view request. By default set to False, because opening a transaction for every view has additional overhead. The impact on performance depends on the query patterns of an application and on how well a database handles locking |
| AUTOCOMMIT | True | By default set to True, because otherwise it would require explicit transactions to perform commits. |
| CONN\_MAX\_AGE | 0 | The lifetime of a database connection in seconds. By default 0 which closes the database connection at the end of each request. Use None for unlimited persistent connections. |
| ENGINE | ''(Empty string) | The database back end to use |
| HOST | ''(Empty string) | Defines a database host, where an empty string means localhost. |
| NAME | ''(Empty string) | The name of the database to use. For SQLite, it’s the full path to the database file. When specifying the path, always use forward slashes, even on Windows (e.g., C:/www/STORE/sqllite |
| OPTIONS | {} (Empty dictionary) | Extra parameters to use when connecting to the database. Available parameters vary depending on database back end, consult the back end module’s own documentation. |
| PASSWORD | '' (Empty string) | The password to use when connecting to the database. Not used with SQLite |
| PORT | '' (Empty string) | The port to use when connecting to the database. An empty string means the default port. Not used with SQLite |
| USER | '' (Empty string) | The username to use when connecting to the database. Not used with SQLite |

**Install Python Database Packages**

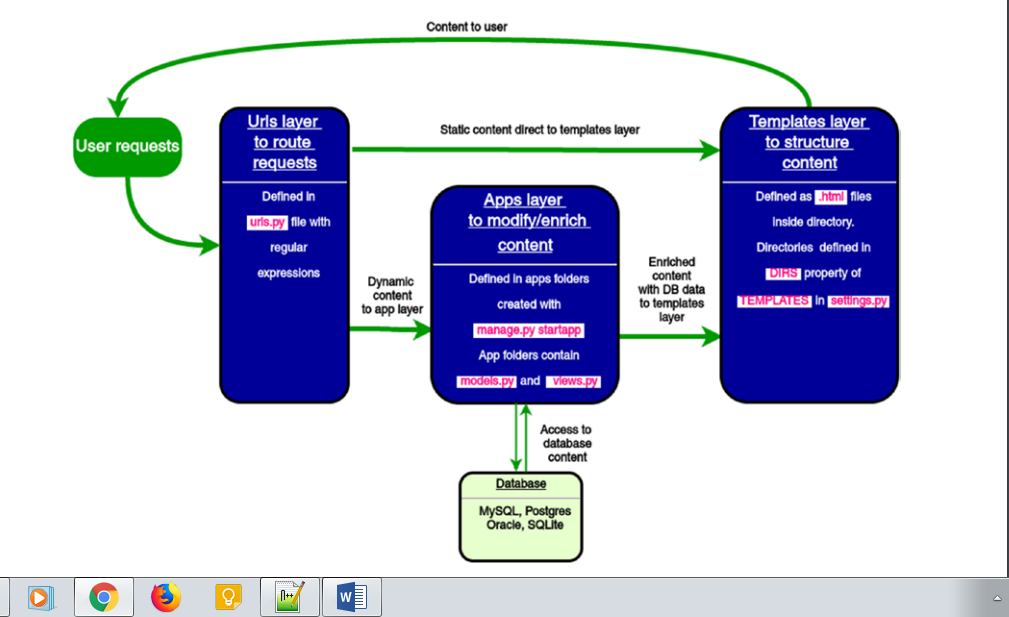
Database Python package pip installation syntax

|  |  |  |
| --- | --- | --- |
| PostgreSQL | psycopg2 | pip install psycopg2 |
| MySQL | mysql-python | pip install mysql-python |
| Oracle | cx\_Oracle | pip install cx\_Oracle |
| SQLite | Included with Python 2.5+ | N/A |

**Then Run migration**

python manage.py migrate

**Set Up Content: Understand Urls, Templates, and Apps**



| *Table 14-4. Methods on User Objects* | |
| --- | --- |
| **Method** | **Description** |
| is\_authenticated() | Always returns True for “real” User objects. This is a way to tell if the user has been authenticated. This does not imply any permissions, and it doesn’t check if the user is active. It only indicates that the user has sucessfully authenticated. |
| is\_anonymous() | Returns True only for AnonymousUser objects (and False for “real” User objects). Generally, you should prefer using is\_authenticated() to this method. |
| get\_full\_name() | Returns the first\_name plus the last\_name, with a space in between. |
| set\_password(passwd) | Sets the user’s password to the given raw string, taking care of the password hashing. This doesn’t actually save the User object. |
| check\_password(passwd) | Returns True if the given raw string is the correct password for the user. This takes care of the password hashing in making the comparison. |
| get\_group\_permissions() | Returns a list of permission strings that the user has through the groups he or she belongs to. |
| get\_all\_permissions() | Returns a list of permission strings that the user has, both through group and user permissions. |
| has\_perm(perm) | Returns True if the user has the specified permission, where perm is in the format "package.codename". If the user is inactive, this method will always return False. |
| has\_perms(perm\_list) | Returns True if the user has *all* of the specified permissions. If the user is inactive, this method will always return False. |
| has\_module\_perms(app\_label) | Returns True if the user has any permissions in the given app\_label. If the user is inactive, this method will always return False. |
| get\_and\_delete\_messages() | Returns a list of Message objects in the user’s queue and deletes the messages from the queue. |
| email\_user(subj, msg) | Sends an email to the user. This email is sent from the DEFAULT\_FROM\_EMAIL setting. You can also pass a third argument, from\_email, to override the From address on the email. |