**Django Cheat Sheet**

I recorded the process how this project is done so that the next time a create a Django project, I have a guide.

Credits to Corey Schafer for the tutorial.

YouTube Link: <https://www.youtube.com/watch?v=UmljXZIypDc&list=PL-osiE80TeTtoQCKZ03TU5fNfx2UY6U4p>

|  |  |
| --- | --- |
| **Start a project:**  Example Project Name: Python\_Django\_Blog | *django-admin startproject Python\_Django\_Blog* |
| **Create an application inside the project:**  Example App Name: blog | *cd Python\_Django\_Blog*  *python manage.py startapp blog*  The urls.py in Python\_Django\_Blog handles the main route before it passes the matched route to other applications.  urls.py will not be created by default in blog app, you need to add this manually.  The views.py in blog app handles what the routes in urls.py should return to the user (render). It is always tied together.  General phase:   * Create route from urls.py main project to app project. * Create views.py content from the project. * Connect urls.py to views.py in that project. |
| **Run the Web Server:** | *python manage.py runserver* |
| **Templates**  Templates (html files) are used so that an html file can be directly displayed. Templates should be created in a way where it is not repeatable (ex: having a base template.) | You need to add first the created app in the INSTALLED\_APPS in the main project (settings.py).  Add *blog.apps.BlogConfig*  Create a folder named templates/project\_name in the project’s folder and add the html files there. In our case we should have:  *templates/blog* in the blog folder.  To display it, we edit the views.py and use the render function pointing to that file. |
| **Passing static date from templates:**  Sample how to a static data from templates to be displayed in the web page. This data normally comes from a database, this is just a guide how to pass.  Jinja templating is used. | The render function accepts a third argument, a dictionary containing a list of dictionaries.    You must pass this to the render as:  *return render(request, "blog/home.html", {"posts": posts})*  Then in html you can loop through the posts using jinja as:  {% for post in posts %}  <h1{{post.title}}</h1>  <p>{{post.author}}</p>  <p>{{post.date\_posted}}</p>  <p>{{post.content}}</p>  {% endfor %} |
| **Using a base html template:**  Having a base template removes repetitions in html files. So when making changes, you don’t need to edit each every file. |  |
| **Adding static files bootstrap template:**  Setting up how static files should be created and accessed so you can add files like CSS and JS.  Then just simply adding the bootstrap template. | Just like in how you create the templates folder, same in static, you should have a folder like:  *static/blogs*  Then you can place inside there the files/folder you need. Example if you have css/main.css, then you can access it in the html files as: |
| **Referencing to links by name:**  Putting links in the html should not be hardcoded because when you change it at one place, you need to change it throughout the whole program. The best way is to reference it via links, that is why the urls.py have name in it. |  |
| **Migrations and Admin Page:**  Django comes with an admin page where you can basically do CRUD operations as an admin.  For you to create a super user or database, you have to create migrations and migrate first. | To create an admin page, you need first to create a superuser.  Commands:  *python manage.py makemigrations*  *python manage.py migrate*  The makemigrations command will detect the changes you have made in the database.  The migrate command applys those changes.  When you first create the project, no database has been applied yet but it is already created, you just need to run one. The database contains data for users which is build-in in Django where you can do things such as authentication, forgot password, login, logout.  To create a super user:  python manage.py createsuperuser  Then it’ll ask for username, email and password.  You can access the admin page by:  Localhost:8000/admin |
| **Creating a database:**  To create a database, we use the models.py which is an ORM.  Each class in models.py corresponds to a certain table in the database.  In this case we create a post class which contains title, content, and date\_posted.  The user class is already created in Django by default we just need to connect it to the post model which means one user can have multiple post. One to many relationships. |  |
| **Querying database contents in command line:**  We can play with the database by opening it in a terminal and type some python code to CRUD the database. | *python manage.py shell*  You can include the User model and Post model by:  import blogs.model import Post  from django.contrib.auth.models import User  Sample commands for CRUD:   * User.objects.all() * User.objects.first() * user = User.objects.filter(username=”Gusion”).first()   You now have the user object and you can view its attributed:   * user.id * user.pk   Create a new post based on the user id.   * post = Post(author = user, title = “My Title”, content = “My Content”)   Date will be automatically added since we configured it that way.  Save it.  post.save() |
| **Retrieving the database and displaying it to web page:**  Same concept as querying in command line, we now do it in the views.py to create list of dictionaries. |  |
| **Registering the created model in admin page:**  When you create a new model, it is not automatically added in the admin page, you need to register it. | To register it at the admin.py of the current app.      Now you can user CRUD in the admin page then changes will be displayed in the actual web page. |
| **User Registration:**  We create a form where user can register because it should not be done in admin page because its for the admin only.  Same on how blog was created just repeat the steps:   * Create new app. * Install the app. * Create static html file. * Create views that point to that html file. * The only difference is the route. * Create route on the main project “/register” and include that directly to the views of user by importing it the main project urls.py.   Then we use the built in UserCreationForm that is already available in Django. |  |
| **Handling form submission:**  Explanation in the comments. |  |
| **Extending the UserCreationForm**  Since Django has a default form for users, it is made in a way that it can be extendable. Current fields only allow username, password, and password confirmation.  Here we add an email address. | To extend the UserCreationForm we need to create a new .py file and inherit that UserCreationForm.  Let’s say forms.py    Then replace the UserCreationForm in views.py by UserRegistrationForm. |
| **Using crispyforms:**  This module is used to style the forms. | *pip install django-crispy-forms*  Add to INSTALLED\_APPS in settings.py:  "crispy\_forms",  Specifiy also the type of template:  CRISPY\_TEMPLATE\_PACK = "bootstrap4"  Load it in the .html you are using like this: |
| **Logging in and logging out:**  Django already has a built-in login and logout views, we just need to import it and fix some routes because the routes is already built-in, if we want to use our own, we edit it in settings.py. | First add the views in the main project urls.py    Then create the corresponding templates: login.html and logout.html  The form in login.html is built-in by Django. Add it like this.    When you login, there is a default route that django finds. We can override it by adding the name of the route in settings.py to redirect it to home page like this:    The logic of logout is already bult-in when the logout route is called. |
| **Views to be accessed by authenticated users only:** | If you want a views to be accessed by authenticated users only, we use the built-in Django login\_required decorator.    Now the url tha points to that profile view, needs to be logged in. There is a default redirect login URL but we need to edit it based on how we created our app. We override it again in settings.py with the name of the url route we want to redirect it to. |
| **Blog Profile:**  Here we display the user’s info and profile picture, as well as editing it and uploading a new picture. | We repeat the same process:   * Create a static html file in user’s app. * Link that html file in user’s view. * Create a route from main project to the views in users profile.   You can access the user’s info in the html like this:    Adding profile picture:     * We place a default picture if it does not exists, default.jpg. * The default location will be placed in the folder “profile\_pics”. * It is created default in the root folder of the project but it is not very good place since it will clutter the root folder if we have many files to upload.   We can change the default place to upload by:    This will set the uploaded files to be created in the media folder of the root project’s directory. One sets the place, the other one sets the URL.  In our case, the profile\_pics folder will now be created inside media folder.  /media/profile\_pics    Then you can access that URL in html like this:    But we need to add those media paths in our main project URL like this: |
| **Signals and Receiver:**  Signals and receiver are used when you want to create another object if another object was created as well. | In our case, we want to create a Profile object whenever a User object is created. We can do this by signals and receiver.  The Django documentation suggest to separate it to a new .py file (signals.py) in the current app’s directory. Some other old code directly add this to models.py but it is not recommendad to avoid import problems.   * Create a signals.py file in the user’s app. |
| **Display certain post via its id in link and display all the post by the user:** | Same workaround:   * Create a route in urls.py. The only difference is we now accept a id parameter. * Link it to views.py. Receive that id parameter. * Query the Posts object to get that post with specific id or get a list of all the post of a certain user. * Create a static html to link to it and pass that data via jinja. |
| **Create a new post:** | Same work around:   * Create route in urls.py. * Connect it to views.py. * Create a html static file. * Create a form and pass it to that html file. * To create a custom form, we inherit from forms.ModelForm then set the model equal to the object of the form we want to contain. |
| **Delete post:** | Same workaround, we get the id of the post, just create a page with confirm deletion. We just need another check that we can delete that view the confirmation to delete if the current logged in user is the also the author of the post. |
| **Edit post:** | Same workaround in delete post: |
| **Pagination:** | Django makes it easy to pagination pages, simply wrap the object you want to paginate in the paginator object.    Then you can access the post objects normally. We just add pagination at the bottom of the page.    The GET parameter for that views are passed to the url, logic handled inside views and then it displays it to the html. |
| **Forgot password feature:** | Creating a forgot password to send in email in Django is very straight forward. The views are built in, we just need to create the route and the static html page.    The way to send email is you have to setup your own smtp, in this case we just Gmail. We just need a host, that a gmail account. Allow less secure apps to access your gmail then add this to settings.py.    I placed the my gmail username and password in an environment variable in my system so that I don’t have to type it manually there since I’m pushing this repository to a public github. |
| **Complete the anchor links.** | Just add the url links in anchor tags. Done. I’ll extend the app on my own. |

**Using AWS S3 for File Storage**

* Create an account in AWS.
* You need to enter card information, but it won’t charge for free suite.
* Search for S3 in services.
* Create bucket with default settings.
* Edit the CORS configuration in the created bucket with the contents here:

<https://devcenter.heroku.com/articles/s3-upload-python>

* Change the allowed origin to “\*”
* Create a user that has more restriction than the admin of the bucket for security purposes.
* Search IAM in services again. Create new user with programmatic access only.
* Add S3 Full Access in the permission of the created user by navigating in Attach existing process directory.
* Just press next with default settings.
* Finally, you will have a window with access key id and secret access key, this will be placed in Django, not directory in code but in environment variables.

AWS\_ACCESS\_KEY\_ID=””

AWS\_SECRET\_ACCESS\_KEY=””

AWS\_STORAGE\_BUCKET\_NAME=””

* pip install django-storages
* pip install boto3
* Edit settings.py, by adding “storages” at INSTALLED\_APPS.
* Add at the end of the settings.py the key=id in the env variables.
* Add also the following:

AWS\_S3\_FILE\_OVERWRITE = False

AWS\_DEFAULT\_ACL = None

DEFAULT\_FILE\_STORAGE = “storages.backends.s3boto3.S3Boto3Storage”

* Comment the resize image in the profile’s model because it won’t work in S3.
* Add the files to S3, drag and drop the contents inside the media folder, not the media folder itself.
* Next, next by default settings.

**Deploying Using Heroku**

* Install heroku.
* Install git.
* *pip install gunicorn*
* *pip freeze > requirements.txt*
* You need to add and commit the requirements.txt to the project’s repository. Git is required,
* *heroku create pythondjangoblog-rom*
* *heroku open*
* *git push heroku master*
* It will have an error about staticfiles, next step solves it.
* Add *STATIC\_ROOT = os.path.join(BASE\_DIR, "staticfiles")* in settings.py
* From here on, for every change, you need to commit it, then git push Heroku master again.
* NOTE that all the credentials used and SECRET\_KEY that is included in settings.py, you should use an environment variable because its values is supposed not to be included in the repository. Then you set its values in the Heroku config file where it cannot be viewed by other users except the admin.

Text

Description automatically generated

Text

Description automatically generated

* Then in your Heroku, you run this commands:

heroku config:set SECRET\_KEY=”b&h3))1(^3yw@@7kgu$i4@1l+-d@7s8-(#9!-c^(80fs#v\_#vh”

heroku config:set EMAIL\_HOST\_USER=” ”

heroku config:set EMAIL\_HOST\_PASSWORD=””

* Push again and the command said it has been deployed, but when you open your browser, it will display this error:
* Graphical user interface

  Description automatically generated
* Create "Procfile" no extension in the root of project’s folder with the contents:

*web: gunicorn Python\_Django\_Blog.wsgi*

* Add the link of the created heroku app in the ALLOWED\_HOST in settings.py

*ALLOWED\_HOST = ["pythondjangoblog-rom.herokuapp.com"]*

* Set *DEBUG=False* in settings.py.
* Push
* pip install django-heroku
* import django\_heroku in settings.py.
* Add this to end of settings.py

django\_heroku.settings(locals())

* Push
* At this point you should have this error:

Text

Description automatically generated

* The sqlite3 db was deleted and it should be since we will user postgresql
* heroku run python manage.py migrate
* heroku run bash, then create an admin account.