Setting up a Fresh Django Project

All of the commands in this file are for macOS (M1 chipset). Please adjust accordingly if you are using Windows.

Start

- [ ] Create a new base folder where ever you typically store your development projects.
- [ ] In the new folder create a new python virtual environment
    - [ ] I use the command pvenv, which is an alias stored in my .zshrc file

```
  alias pvenv="/opt/homebrew/opt/python@3.11/bin/python3.11 -m venv
venv && source venv/bin/activate && pip install --upgrade pip"
```

- [ ] The first package to install is pip-tools, The package pip-tools provides you with pip-compile. To use this feature build two files requirements-dev.in, and requirements.in
- [ ] In the requirements-dev.in file you should place the python packages that are only need for development such as black, flake8, or ruff. The entries should be in the format:

```
  black==<latest_version>
  ruff==<latest_version>
```

- [ ] Add the packages that are necessary for production to the requirements.in in the same manner. My typical default packages are:

```
  Django==<latest_version>
  django-extensions==<latest_version>
  environs[django]==<latest_version>
```

You should add additional packages to this file as needed, and recompile the requirements.in file.

- [ ] When ready run the command pip-compile as follows:

```
  pip-compile requirements.in
  pip-compile requirements-dev.in
```

- [ ] This will validate your package selections and create a requirements.txt and requirements-dev.txt.
- [ ] You can now populate your virtual environment in the usual way:

```
pip install -r requirements.txt
pip install -r requirements-dev.txt
```

- [ ] Next create your django project with the following command:

```
django-admin startproject project_name .
```

- [ ] Now there is mixed feelings about the use of the period or not to use it. My take is that if you created a top-level folder with the project name there is no real reason to duplicate that structure. There are times when it might be necessary to separate different parts of the project, such as a Django backend and a react frontend.
- [ ] I've now seen multiple ways to name the project. A lot like to use the actual project name, other's like to use the word 'core', while other's use words like 'django_project' or 'project'. In the end pick a naming convention that works for you and that will stand the test of time.
- [ ] DO NOT immediately run migrations! You first need to put in place your configuration files, a short list would be:

```
.env
.flake8
.gitignore
.markdownlint.json
.ruff.toml
LICENSE.txt
README.md
```

This list of files can grow or shrink depending on the projects need. please see Supporting Files for details on each of these files.

(back to top)

Configuration

INSTALLED_APPS

- [ ] Now is the time to add the necessary INSTALLED_APPS into settings.py. One package that I always add to each project is django-extensions, this must be added to INSTALLED_APPS as "django_extensions",.

TEMPLATES

- [ ] The next spot to look at is the templates settings. A lot of people prefer to not modify the TEMPLATES settings, and go with creating a templates folder in each app with a subfolder of the same name to hold the html files. This is a good practice if you are going to turn you project into a

module that can be added to other projects in the future. However if this is a one-off project then having all the html files together makes more sense.

```
"DIRS": [], or "DIRS": [BASE_DIR / 'templates'],
```

DATABASES

- [ ] For most developers the sqlite3 database that ships by default by Django is sufficient to use while developing the application. Of course when it comes to production however the application will require a more robust database such as PostgreSQL, or mysql.

sqlite3 example

```
DATABASES = {
"default": {
    "ENGINE": "django.db.backends.sqlite3",
    "NAME": BASE_DIR / "db.sqlite3",
    }
}
```

PostgreSQL example

```
DATABASES = {
'default': {
    'ENGINE': 'django.db.backends.postgresql',
    'NAME': "database_name",
    'USER': "database_user",
    'PASSWORD': "database_user_password",
    'HOST': "database_host_address",
    'PORT': "database_port",
    }
}
```

The database name, user, password and host address should be treated as secure environment variables and should be store in the .env file.

INTERNATIONALIZATION

- [ ] By default Django will use UTC, and if your application will have a global reach this is probably a good setting, however or a regional application you will want to change the settings from UTC to your local timezone, for example America/Toronto

STATIC & MEDIA FILES

- [ ] The only setting that Django provides by default is the STATIC_URL. These two additional lines need to be added to ensure that all static files are picked up as expected:

```
STATICFILES_DIRS = [BASE_DIR / 'static']
STATIC_ROOT = BASE_DIR / "static_cdn"
```

- [ ] if you will be using Pillow or some other image manipulation package you will need to have the following lines configured in settings.py:

```
MEDIA_URL = "media/"
MEDIA_ROOT = BASE_DIR / "media"
```

To make this all function you need to add the following lines to the project's urls.py:

```
from django.conf import settings
from django.conf.urls.static import static
...
urlpatterns = [
    ...
]

urlpatterns += static(settings.MEDIA_URL,
document_root=settings.MEDIA_ROOT)
urlpatterns += static(settings.STATIC_URL,
document_root=settings.STATIC_ROOT)
```

(back to top)

Creating Supporting folders

- [ ] Create the following three folders in the root of the project, note the sub folders under static, and templates:
  - [ ] media
  - [ ] static
    - [ ] assets # favicon.ico and other static assets
    - [ ] css # Additional css files
    - [ ] images # Static images for the website
    - [ ] js # JavaScript support files
  - [ ] templates
    - [ ] includes
    - [ ] partials
    - [ ] folder for each application

(back to top)

First App - Accounts (Custom User Model)

This section will detail the following:

- [ ] Create a CustomUser Model
- [ ] Update the settings.py file accordingly
- [ ] Customize the UserCreationForm and UserChangeForm
- [ ] Add the customer user model to accounts/admin.py

New Application:

- [ ] Create the new application, it should be named accounts:

```
python manage.py startapp accounts
```

- [ ] Create an AbstractUser model for the CustomUser model:

```
from django.contrib.auth.models import AbstractUser
# from django.db import models


class CustomUser(AbstractUser):
    pass
```

- [ ] Add the new application to the INSTALLED_APPS list:

```
INSTALLED_APPS = [
    ...
    # Applications
    "accounts.apps.AccountsConfig",
    ...
]
```

- [ ] Below the INSTALLED_APPS list add the setting:

```
AUTH_USER_MODEL = "accounts.CustomUser"
```

- [ ] Now you can proceed to run makemigrations and migrate commands:

```
python manage.py makemigrations accounts
python manage.py migrate
```

Customizing Forms

Now that we have a basic user model we need to customize the sign up and login forms to suite our needs.

- [ ] Create the following two forms in a new file named accounts/forms.py

```
    from django.contrib.auth import get_user_model
    from django.contrib.auth.forms import UserCreationForm, UserChangeForm


    class CustomUserCreationForm(UserCreationForm):
        class Meta:
            model = get_user_model()
            fields = (
                "email",
                "username",
            )


    class CustomUserChangeForm(UserChangeForm):
        class Meta:
            model = get_user_model()
            fields = (
                "email",
                "username",
            )
```

Register the CustomUser model with admin.py

- [ ] The new model needs to be registered in admin.py so it becomes visible in Django Administration. Add the following lines to admin.py:

```
    ...
    from django.contrib.auth import get_user_model
    from django.contrib.auth.admin import UserAdmin

    from accounts.forms import CustomUserChangeForm,
CustomUserCreationForm

    CustomUser = get_user_model()


    class CustomUserAdmin(UserAdmin):
        add_form = CustomUserCreationForm
        form = CustomUserChangeForm
        model = CustomUser
        list_display = [
            "email",
            "username",
            "is_superuser",
```

```
        ]


    admin.site.register(CustomUser, CustomUserAdmin)
```

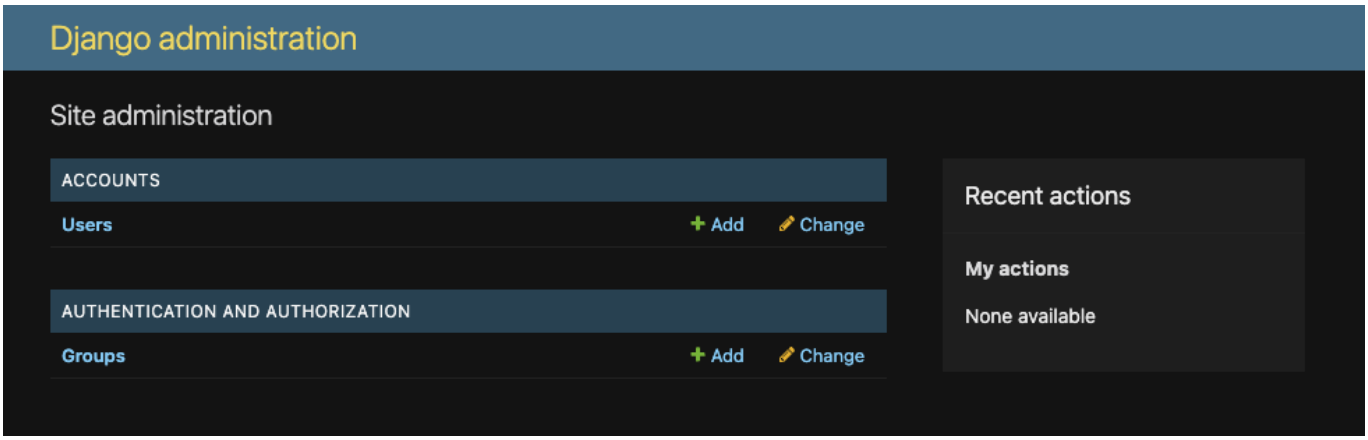
Create a Super User

Now that we have the user model in place we can go ahead and create a superuser so we can log into Django Admin and check our work.

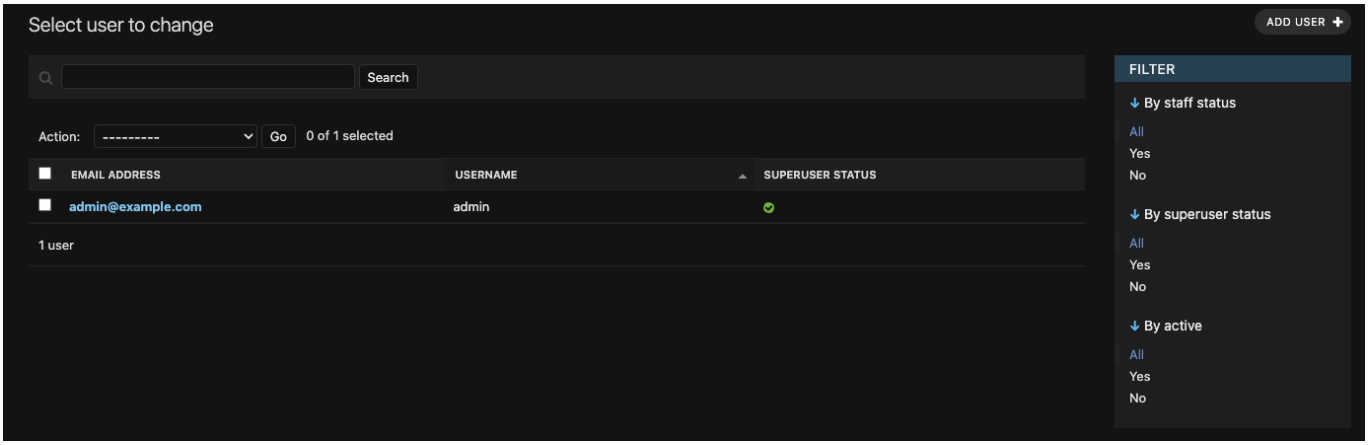
```
    python manage.py createsuperuser --username djadmin --email
djadmin@example.com
```

Set a password on the administration account when prompted. Once done you can login and see something similar to this:

home Page



Users Page



(back to top)

Testing the accounts model

Untested code is just that - untested. You won't know if it behaves the way you expect or if it can be used to do something else. Here is a quick simple test for the CustomUser model we created.

```
    from django.contrib.auth import get_user_model
    from django.test import TestCase


    class CustomerUserTests(TestCase):
        def test_create_user(self):
            User = get_user_model()
            user = User.objects.create_user(
                username="david", email="dave@example.com",
password="Testing1234"
            )
            self.assertEqual(user.username, "david")
            self.assertEqual(user.email, "dave@example.com")
            self.assertTrue(user.is_active)
            self.assertFalse(user.is_staff)
            self.assertFalse(user.is_superuser)

        def test_create_superuser(self):
            User = get_user_model()
            user = User.objects.create_superuser(
                username="superadmin", email="superadmin@example.com",
password="Testing1234"
            )
            self.assertEqual(user.username, "superadmin")
            self.assertEqual(user.email, "superadmin@example.com")
            self.assertTrue(user.is_active)
            self.assertTrue(user.is_staff)
            self.assertTrue(user.is_superuser)
```

(back to top)

GIT

Here would be a good point to initialize a git repository and make an initial commit of the work done to date.

(back to top)

Creating a Second App

- [ ] Time to add the primary application for the project. In this example it will be tasks:

```
    python manage.py startapp core
```

- [ ] Register the app in settings.py by adding the following:

```
    INSTALLED_APPS = [
        ...
        # Applications
```

```
        "accounts.apps.AccountsConfig",
        "tasks.apps.TasksConfig",
        ...
    ]
```

(back to top)

Creating a Base Template & home page template

- [ ] In the templates folder created earlier you should now create three sub-folders, one named includes, the other registration, and tasks. In the templates folder itself you want to build a base.html file using your editor of choice. This file will contain all the code you don't want to repeat on every page. A basic example is:

```
{% load static %}
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Tasker {% block title %}{% endblock title %}</title>
    <link rel="shortcut icon" href="{% static 'assets/favicon.ico' %}"
type="image/x-icon">
    <link rel="stylesheet"
href="https://bootswatch.com/5/yeti/bootstrap.min.css">
    <link rel="stylesheet" href="{% static 'css/base.css' %}">
    <script src="{% static 'js/base.js' %}" defer></script>
</head>
<body>
    {% include 'includes/navbar.html' %}
    <div class="container mt-4">
        {% block content %}

        {% endblock content %}
    </div>
    <script
src="https://cdn.jsdelivr.net/npm/@popperjs/core@2.11.8/dist/umd/popper.mi
n.js" integrity="sha384-
I7E8VVD/ismYTF4hNIPjVp/Zjvgyol6VFvRkX/vR+Vc4jQkC+hVqc2pM8ODewa9r"
crossorigin="anonymous"></script>
    <script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/js/bootstrap.min.js
" integrity="sha384-
0pUGZvbkm6XF6gxjEnlmuGrJXVbNuzT9qBBavbLwCsOGabYfZo0T0to5eqruptLy"
crossorigin="anonymous"></script>
</body>
</html>
```

As you can see the example includes bootstrap 5 provided by Bootswatch, and scripts provided by bootstrap. I use the naming convention base.css and base.js for additional CSS directives and JavaScript

modules. These files can be named whatever you prefer.

(back to top)

Adding django-crispy-forms

If you are using Bootstrap to style your project you can quickly style forms with 'django-crispy-forms along with the supporting template pack. Since Bootstrap 5 is well established you should add the following two packages to requirements.in:

```
django-crispy-forms==<latest_version>
crispy-bootstrap5==<latest_version>
```

Once these two packages area added to requirements.in don't forget to run:

```
    pip-compile requirements.in
    pip install -r requirements.txt
```

You need to add the following two line to INSTALLED_APPS in settings.py:

```
    INSTALLED_APPS = [
        ...
        "crispy_bootstrap5",
        "crispy_forms",
        ...
    ]
```

You also need to add the following two lines to settings.py below INSTALLED_APPS:

```
    CRISPY_ALLOWED_TEMPLATE_PACKS = "bootstrap5"
    CRISPY_TEMPLATE_PACK = "bootstrap5"
```

- [ ] Now in any template where a form is used you need to make the following changes:

```
    # At the top of the template files but below any extends directive
    {% load crispy_forms_tags %}

    # Where ever there is a forms tag, change it from
    {{ form.as_P }} to {{ form|crispy }}
```

Your forms should now be styled more appropriately to match your bootstrap styling.

(back to top)

Setup for Tailwindcss & HTMX using Node and npm

While Bootstrap and Bootswatch are good solutions for quickly styling your project, bootstrap has the reputation of always looking like bootstrap. It's very noticeable as a style. Tailwindcss on the other hand is more extensible and you can drastically change the style of a website in very short order. It also has a built in dark mode.

This section will document how to setup and configure node with npx/npm and add tailwindcss and HTMX.

Prerequisite

1. Ensure that a recent version of node is installed. Check Node.js for the latest version.
2. Use the following command to install the necessary packages:

```
npm install -D tailwindcss postcss autoprefixer
```

3. Once the packages are installed you can run the command:

```
npx tailwindcss init
```

4. At the root of the project create a file named postcss.config.js, the file should contain the following lines:

```
module.exports = {
plugins: {
    tailwindcss: {},
    autoprefixer: {},
}
}
```

5. Configure the template path based on your project:

```
/** @type {import('tailwindcss').Config} */
module.exports = {
content: ['./templates/**/*.html'],
theme: {
    extend: {},
},
plugins: [],
}
```

6. Under your static folder create a css folder (as suggested earlier), and in the css folder create a file named main.css. In this file add the following lines:

```
@tailwind base;
@tailwind components;
@tailwind utilities;
```

7. In the package.json file, under scripts, add the following definitions:

```
    "dev": "tailwindcss -i static/css/main.css -o
static/css/main.min.css --watch --verbose",
    "build": "tailwindcss -i static/css/main.css -o
static/css/main.min.css",
```

8. You can now run "npm run dev" to compile main.min.css.

9. Add the link line to main.min.css in your base.html file. Like so:

```
<link rel="stylesheet" href="{% static 'css/main.min.css' %}">
```

10. Tailwindcss styles should now be applied to your pages.

Adding HTMX to your Installation

htmx gives you access to AJAX, CSS Transitions, WebSockets and Server Sent Events directly in HTML, using attributes, so you can build modern user interfaces with the simplicity and power of hypertext

1. Use npm to install htmx using the following command:

```
npm install -D htmx.org
```

2. Modify your dev, and build scripts to copy htmx.min.js into your static/js folder:

```
    "dev": "tailwindcss -i static/css/main.css -o
static/css/main.min.css --watch --verbose; cp
node_modules/htmx.org/dist/htmx.min.js static/js/htmx.min.js",
    "build": "tailwindcss -i static/css/main.css -o
static/css/main.min.css; cp node_modules/htmx.org/dist/htmx.min.js
static/js/htmx.min.js"
```

3. Add a script link to your base.html file, like so:

```
    <script src="{% static 'js/htmx.min.js' %}"></script>
```

TODO: Add alpine.js to the project.

TODO: Add steps to actually minify the css output file.

(back to top)

Setup Django-Allauth

Django-allauth can be used in two configurations, with social websites configured to provide authentication, and used to enhance the existing Django authentication system. These notes will document that later.

The package for django-allauth needs to be included in the requirements.in/requirements.txt file. Once the django-allauth package is installed, you must carry out the following configuration changes"

```
        INSTALLED_APPS = [
            ...
            "django.contrib.sites",
            # Applications
            "accounts.apps.AccountsConfig",
            "tasks.apps.TasksConfig",
            # 3rd Party Libraries
            "allauth",
            "allauth.account",
            ...
        ]

        MIDDLEWARE = [
            ...
            "allauth.account.middleware.AccountMiddleware",
        ]

        # Django-allauth Settings
        AUTH_USER_MODEL = "accounts.CustomUser"
        SITE_ID = 1
        LOGIN_REDIRECT_URL = "home"
        LOGOUT_REDIRECT_URL = "home"
        ACCOUNT_SESSION_REMEMBER = True
        ACCOUNT_SIGNUP_PASSWORD_ENTER_TWICE = False
        ACCOUNT_USERNAME_REQUIRED = False
        ACCOUNT_AUTHENTICATION_METHOD = "email"
        ACCOUNT_EMAIL_REQUIRED = True
        ACCOUNT_UNIQUE_EMAIL = True

        AUTHENTICATION_BACKENDS = [
            # Needed to login by username in Django admin, regardless of
`allauth`
            'django.contrib.auth.backends.ModelBackend',
            # `allauth` specific authentication methods, such as login by
email
```

```
        'allauth.account.auth_backends.AuthenticationBackend',
    ]

    EMAIL_BACKEND = "django.core.mail.backends.console.EmailBackend"
```

These settings will enable allauth to enhance the existing authentication system.

1. It will set the email address as the authenticator rather than the username.
2. It will send a verification email to all new signup's hence the EMAIL_BACKEND being set to console during development.

- [ ] At this point you should run:

```
python manage.py migrate
```

- [ ] Update the projects urls.py file and replace all lines with reference to "accounts" with:

```
path('accounts/', include('allauth.urls')),
```

- [ ] Under the templates folder you need to create a new folder named account (not plural - singular), and move the login.html, and signup.html files from templates/registration to templates/account. You can then remove the registration folder from the templates directory.
- [ ] At this same time all links to login, signup, or logout need to be changed to account_login, account_signup, and account_logout.
- [ ] It's now time to create a customized logout template:

```
{% extends '_base.html' %}

{% load crispy_forms_tags %}

{% block title %}| Logout{% endblock title %}

{% block content %}

<h1>Logout</h1>
<p class="lead">Are you sure you want to log out?</p>
<form action="" method="post">
    {% csrf_token %}
    {{ form|crispy }}
    <button class="btn btn-danger" type="submit">Log out</button>
</form>

{% endblock content %}
```

(back to top)

Setup Django-debug-toolbar

The django-debug-toolbar is a module that can help you analyze the performance of your queries when the page loads. Knowing how long it takes a task or a post to load will help you design an application that users will be happy to use because it is responsive when they click on a link.

The package can be added to either requirements.in, or requirements-dev.in. It's not important since the configuration of the debug_toolbar will only allow it to be seen when running against the localhost address (127.0.0.1).

In requirements.in add the following line to the file:

```
    django-debug-toolbar==<latest_version>
```

The run the command:

```
    pip-compile requirements.in && pip install -r requirements.txt
```

- [ ] In settings.py ensure that under INSTALLED_APPS that 'django.contrib.staticfiles' is listed, and that STATIC_URL is correctly configured. Under TEMPLATES make sure the BACKEND includes "django.template.backends.django.DjangoTemplates", and that APP_DIRS is set to True.
- [ ] You can now add "debug_toolbar", to the list of INSTALLED_APPS.
- [ ] In the projects urls.py file add the path for the debug urls:

```
    path("__debug__/", include("debug_toolbar.urls")),
```

- [ ] Under MIDDLEWARE, add the following line:

```
    "debug_toolbar.middleware.DebugToolbarMiddleware",
```

NOTE: The order of MIDDLEWARE is important. You should include the Debug Toolbar middleware as early as possible in the list. However, it must come after any other middleware that encodes the response's content, such as GZipMiddleware.

- [ ] Now add the setting INTERNAL_IPS, this will limit when the toolbar can be seen.

```
    INTERNAL_IPS = [
        "127.0.0.1",
    ]
```

(back to top)