To query JSON data in a Django model when the JSON data is not in the root, you can use the \_\_ syntax to traverse the JSON data. Here is an example query that returns the id and group\_id of the records in the folder\_table table where the JSON data in the data column has the first name “Allen” and the last name “Doe”:



This query uses the \_\_ syntax to traverse the master and basicdata keys in the JSON data, and the Q object to combine the two conditions with an AND operator. The values method is used to specify the fields to include in the result set.

In PostgreSQL, you can use the -> operator to extract values from the JSON data. Here is an example query that returns the id and group\_id of the records in the folder\_table table where the JSON data in the data column has the first name “Allen” and the last name “Doe”:



This query uses the -> operator to traverse the master and basicdata keys in the JSON data, and the ->> operator to extract the values of the firstName and lastName keys as strings. The LIKE operator is used to search for the values that match the search criteria.

<https://stackoverflow.com/questions/29939042/drf-how-to-integrate-django-rest-framework-jwt-to-djoser>

<https://forum.djangoproject.com/t/login-authentication-using-drf-api-backend/12772>

<https://stackoverflow.com/questions/41166704/how-to-make-centralized-login-server-with-django>

<https://www.django-rest-framework.org/api-guide/authentication/>

Authenticate Django Apps with Another Django App Hosted Separately

<https://python.plainenglish.io/authenticating-django-applications-with-another-django-app-hosted-separately-905949549fa2>

**Production**

Command to list items in the table “cs\_search\_requests” by connecting to default database in settings.



**Gunicorn config:** *sudo nano /etc/systemd/system/gunicorn.service*



**Config for NGINX:**



enable the file by linking

sudo ln -s /etc/nginx/sites-available/myproject /etc/nginx/sites-enabled

To enable HTTPS for your Django application running on a private network with Ubuntu 22.04 and NGINX and Gunicorn, you can follow these steps:

1. **Install OpenSSL**: Install OpenSSL on your Ubuntu server by running the following command in your terminal: sudo apt-get install openssl.
2. **Generate a self-signed SSL certificate**: Generate a self-signed SSL certificate using OpenSSL by running the following command in your terminal: sudo openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout /etc/ssl/private/nginx-selfsigned.key -out /etc/ssl/certs/nginx-selfsigned.crt. This command will generate a self-signed SSL certificate that is valid for 365 days.
3. **Configure Nginx to use HTTPS**: Update your Nginx configuration file to use HTTPS by adding the following lines:



## Best Practices: What to Include and Exclude

<https://clouddevs.com/django/version-control/>

Django projects have specific files that shouldn’t be included in version control, like `db.sqlite3` or the `\_\_pycache\_\_` directories. Thankfully, Git allows you to use a `.gitignore` file to specify patterns of files that it should overlook.

Example `.gitignore` for Django:



This setup excludes cache files, logs, databases, and the media directory, among other things.

**A Comprehensive Guide to Structuring Django Projects: Best Practices and Example**

.

└── backend/

├── app/

│ ├── api/

│ │ └── v1/

│ │ ├── serializers.py

│ │ ├── urls.py

│ │ └── views.py

│ ├── migrations/

│ │ └── \_\_init\_\_.py

│ ├── static/

│ │ └── app/

│ │ ├── css

│ │ ├── img

│ │ └── js

│ ├── templates/

│ │ └── app

│ ├── test/

│ │ ├── test\_models.py

│ │ ├── test\_setup.py

│ │ └── test\_views.py

│ ├── \_\_init\_\_.py

│ ├── actions.py

│ ├── admin.py

│ ├── apps.py

│ ├── constants.py

│ ├── exceptions.py

│ ├── filters.py

│ ├── forms.py

│ ├── managers.py

│ ├── messages.py

│ ├── models.py

│ └── views.py

├── backend/

│ ├── \_\_init\_\_.py

│ ├── asgi.py

│ ├── settings.py

│ ├── urls.py

│ └── wsgi.py

├── common/

│ ├── \_\_init\_\_.py

│ ├── constants.py

│ ├── custom\_logger.py

│ ├── exceptions.py

│ ├── filters.py

│ ├── generics.py

│ ├── messages.py

│ ├── mixins.py

│ ├── models.py

│ ├── pagination.py

│ ├── permissions.py

│ ├── swagger.py

│ ├── utils.py

│ └── viewsets.py

├── static/

│ ├── css

│ ├── img

│ └── js

├── templates

├── .dockerignore

├── .gitignore

├── .sample.env

├── Dockerfile

├── docker-compose.yaml

├── manage.py

└── requirements.txt

From <[*https://medium.com/@akshatgadodia/a-comprehensive-guide-to-structuring-django-projects-best-practices-and-example-afb77d8497d5*](https://medium.com/@akshatgadodia/a-comprehensive-guide-to-structuring-django-projects-best-practices-and-example-afb77d8497d5)>

# How to configure your Django project for multiple environments?

Learn how to configure Django project for multiple environments to write manageable code with fewer duplications and more accurate settings. If you'd like to just take a quick glance at the code, take a look at this SlideShare presentation. Below, I'll explain how to configure Django project step by step.

[The Guidelines to Adjusting Django Projects to Multiple Environments (apptension.com)](https://www.apptension.com/blog-posts/how-to-configure-your-django-project-for-multiple-environments)

## How to Use Multiple Settings Files in Django Project

This is done effectively by breaking down the settings.py module into multiple files

Let's take a brand new Django project structure, which looks like the tree structure below. Here's how you can split the settings file into multiple configuration files



1. **Create a directory named “settings” within the core directory**

The first thing we want to do is to create a folder named settings.

1. **Add a \_\_init\_\_.py file to the settings directory**

The \_\_init\_\_.py will make the settings directory a python module.

1. **Rename the previous settings.py file to base.py and move it into the settings directory**

The base.py will be responsible for providing the settings configuration that is common among all environments such as development, production, staging, etc.

Here is how the current project structure should look after the three steps above.



1. **Create a settings.py module for each environment.**

You should create settings.py for all common environments using the name of the environment as the file name.

Some common use cases are:

* ci.py :This file will contain configurations of continuous integration (CI/CD) or tests. You can use tests.py in place of ci.py.
* development.py: This file will contain configurations of the Development version alone. It is sometimes named local.py or dev.py.
* production.py : This file will contain configurations of the production environment. It is sometimes named prod.py.
* staging.py : This file will contain configurations of the staging environment.

It is important to know that the file names are merely conventional and not enforced. You can call it whatever you like, as long it adheres to python's file naming rules.

Here's how the file structure would look like after creating all the files representing the various environments above:



How to Configure Settings for Different File Environments

Here's how you would go about setting the configurations for the development, production, and base settings.

To do this, we will use the decouple package.

Decouple helps you to organize your settings so that you can change parameters without having to redeploy your app.

Decouple adheres to the following principles, which we will follow:

1. Store parameters in ini or .env files.
2. Define comprehensive default values.
3. Convert values to the correct data type.
4. Have only one configuration module to rule all your instances.

Install the decouple package using the code below

pip install python-decouple

First, let's start with the base.py module in settings/base.py.

We will only define a handful of settings, so the example does not get too big.



Next, we will create a development.py configuration by first extending our base.py settings module.

In settings/development.py,



Next, we will define a production.py configuration, which also extends the base.

In settings/production.py,



When working with Python, it is generally advised to avoid using star import, which indicates, importing everything.

This is because star imports may put lots of unnecessary stuff in the namespace, which in some cases can cause issues.

However, importing all from Django base settings is one of the few cases where it is accepted.

When importing everything from the base, it is important to know that

Also bear in mind that even though we are using different files for development and production, you still have to protect sensitive data.

Keep passwords and secret keys in environment variables or use a library like Python-Decouple which is used in this example.

## How to Set Up Django Server to Use Different Settings in Different Environments

Since we no longer have a single settings.py file in the project root, running the command: python manage.py runserver will no longer work unless we make some changes.

There are three ways you can do this:

**Method One: Run the server with the settings you want to use**

Here, you have to pass the settings.py module you want to use in the command line:



Or



**Method Two: Configure it in manage.py**

You can edit the manage.py to set the default settings module to your development.py module.

To do that, simply edit the manage.py file, like this:

In manage.py,



So, basically, we changed the line from:

# 7 Critical Django Production Server Settings to Configure Before Going Live

<https://tonyteaches.tech/django-production-server-settings/>

1. Hide the Secret Key
2. 2. Install an SSL Certificate for HTTPS
3. Django HTTPS Settings
4. Django HSTS Settings
5. Only Serve Your Domain Name
6. Turn Off Debug Mode
7. Copy Static Files
8. Automated Django Deployment Check

# Run project with React + Vite

