**Chapter 09: Environment Variables**

Environment variables are variables that can be loaded into the operating environment of a project at run time as opposed to hard coded into the codebase itself. They are considered an integral part of the popular Twelve-Factor App Design methodology and a Django best practice because they allow a greater level of security and simpler local/production configurations.

Why greater security? Because we can store truly secret information–database credentials, API keys, and so on–separate from the actual code base. This is a good idea because using a version control system, like git, means that it only takes one bad commit for credentials to be added in there forever. Which means that anyone with access to the codebase has full control over the project. This is very, very dangerous. It’s much better to limit who has access to the application and environment variables provide an elegant way to do so.

A secondary advantage is that environment variables make it much easier to switch between local and production code environments. As we will see, there are a number of setting configurations that Django uses by default intended to make local development easier, but which must be changed once the same project is ready for production.

**Environs**

There are many different ways to work with environment variables in Python but for this project we’ll use the environs package, which has a Django-specific option that installs a number of additional packages that help with configuration.

We will install it by first updating the requirements.txt file.

requirements.txt

asgiref==3.5.2

Django==4.0.4

psycopg2-binary==2.9.3

sqlparse==0.4.2

django-crispy-forms==1.14.0

crispy-bootstrap5==0.6

django-allauth==0.50.0

environs[django]==9.5.0 # new

marshmallow==3.20.1 # new and update version

Then spin down the currently running Docker container, rebuild our Docker image which will contain the new software package, and start up the container again.

docker-compose down

docker-compose up -d –build

In the django\_project/settings.py file, there are three lines of imports to add at the top of the file, just under the import of Path.

# django\_project/settings.py

from pathlib import Path

from environs import Env # new

env = Env() # new

env.read\_env() # new

But is not going to work because .venv is on chapter 4 so the word “environ” is going to have a Red Squiggle underneath so to fix it you have to:

1. Press Ctrl + Shift + P

2. Type: Python: Select Interpreter → Press Enter

3. Click: Enter interpreter path

4. Then click: Find...

5. Navigate to this path:

C:\Users\Jean-Marc H\Documents\Django for professionals\Chapter 04. Bookstore Project\.venv\Scripts

6. python.exe

7. click on “Select Interpreter”

8. popup confirms VS Code has detected the correct Python environment (.venv from Chapter 4), and it's offering to reload the terminal to apply it properly. That’s exactly what we want click on it.

(if the pop up window on vs code disappear or still have the red Suiggle on setting restart the process).

If everything work the way it should be you’re all set.

**SECRET\_KEY**

For our first environment variable we’ll set the SECRET\_KEY, a randomly generated string used for cryptographic signing and created whenever the startproject command is run. It is very important that SECRET\_KEY actually be kept, well, secret.

In my django\_project/settings.py file, it has the following value:

# django\_project/settings.py

SECRET\_KEY = "django-insecure-hv1(e0r@v4n4m6gqdz%dn(60o=dsy8&@0\_lbs8p-v3u^bs4)xl"

Note that the double quotes ("") around the SECRET\_KEY make it a Python string. The double quotes are not actually part of the SECRET\_KEY value itself, which is an easy mistake to make.

There is a two-step process to switching over to environment variables:

• add the environment variable to the docker-compose.yml file

• update django\_project/settings.py to point to the variable

In the docker-compose.yml file, add a section called environment under the web service. It will be a variable that we’ll call DJANGO\_SECRET\_KEY with the value of our existing SECRET\_KEY. This is what the updated file looks like:

docker-compose.yml

services:

web:

build: .

command: python /code/manage.py runserver 0.0.0.0:8000

volumes:

- .:/code

ports:

- 8000:8000

depends\_on:

- db

environment:

- DJANGO\_SECRET\_KEY = 'django-insecure-kdhq8uv91b4(ae==g+v6dd41)=re54dovd^1xzv+7d\*t6m0d=('

db:

image: postgres:13

volumes:

- postgres\_data:/var/lib/postgresql/data/

environment:

- POSTGRES\_HOST\_AUTH\_METHOD=trust

volumes:

postgres\_data:

|  |
| --- |
| Note that if your SECRET\_KEY includes a dollar sign, $, then you need to add an additional dollar sign, $$. This is due to how docker-compose handles variable substitution. Otherwise you will see an error! |

The second step is to update the SECRET\_KEY configuration within django\_project/settings.py.

# django\_project/settings.py

SECRET\_KEY = env("DJANGO\_SECRET\_KEY")

If you refresh the website now you’ll likely see an error. Django requires a SECRET\_KET to work and in this case, our value is being stored within docker-compose.yml, which is used to start the container. It won’t be applied until we restart the current container which we’ll do so now.

docker-compose down

docker-compose up -d

Refresh the page again and everything works as before, which is what we want. Astute readers may notice that even though we are now using an environment variable the actual value of SECRET\_KEY is still visible in our source code as it’s merely moved to docker-compose.yml which is tracked by Git. This is true! However, when we configure our website for production, we will create a separate file for production purposes–docker-compose-production.yml–and immediately add it to our .gitignore file so it is not tracked in the source code.

For now though, the goal of this chapter is to start using environment variables locally for the values that need to be either truly secret or switched in a production context.

**NOTE:** Because of versions conflict since the book is from 2020 you have to do it that way to make it work:

✅ Finalized Summary – Fixing Django .env Setup & Docker Compatibility (Chapter 9)

Updated SECRET\_KEY handling

Moved SECRET\_KEY from settings.py to .env using environs library (Env()).

Adjusted settings.py to load environment variables using environs[django].

Initial Rebuild Attempt

Ran docker-compose down and docker-compose up --build -d.

Expected web app to still work — but it failed (browser error or 500).

Compatibility Issues Due to Outdated Book Stack

Book uses older Python/Django/Docker images.

Your Docker and VS Code setup are newer, leading to version conflicts.

marshmallow vs environs Version Conflict

environs==9.5.0 expects marshmallow<4.

You had marshmallow==4.0.0, which breaks compatibility.

✅ Fix: Pin marshmallow==3.20.1 in requirements.txt.

PostgreSQL Version Mismatch

Django 5.2+ requires PostgreSQL 14+.

The book uses postgres:13, which caused an error.

✅ Fix: Updated to postgres:15 in docker-compose.yml.

Rebuild Clean Docker Environment

Used docker-compose down -v to remove old containers & volumes.

Then rebuilt everything: docker-compose up --build -d.

Migrations Missing (Critical)

Because volumes were wiped, the DB was empty (no schema).

App crashed with errors like relation "django\_session" does not exist.

✅ Fix: Re-run migrations after rebuild:

docker-compose exec web python manage.py migrate

Final Check

Web app loads as expected.

Sign-up/login pages now work (or fail appropriately if expected).

You’re ready to continue to the next step (e.g., DEBUG, ALLOWED\_HOSTS).

✅ Bonus Notes (Optional but Helpful to Document):

Always check the Django and library version compatibility in the official docs.

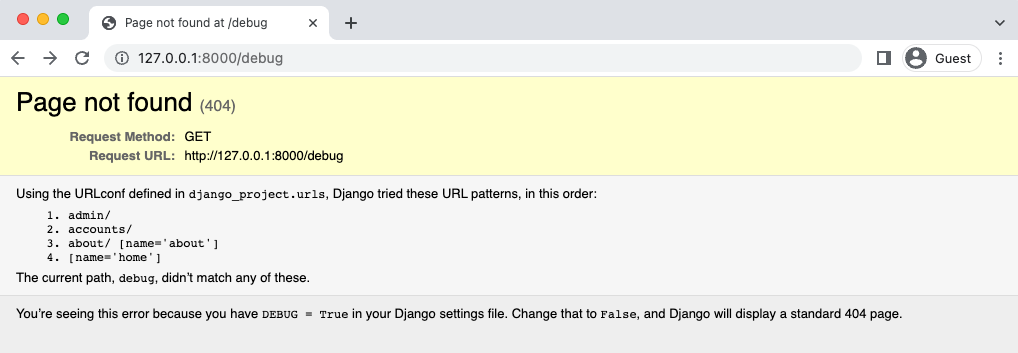
Use pip freeze to pin all working versions after fixing everything.

Updating Docker images or Python versions may trigger issues in older projects.

**DEBUG and ALLOWED\_HOSTS**

As the Django deployment checklist notes, there are a number of settings that must be updated before a website can be deployed safely in production. Chief among them are DEBUG and ALLOWED\_HOSTS.

When DEBUG is set to True, Django displays a lengthy message and detailed bug report whenever an error occurs. For example, try visiting a page that does not exist such as /debug.



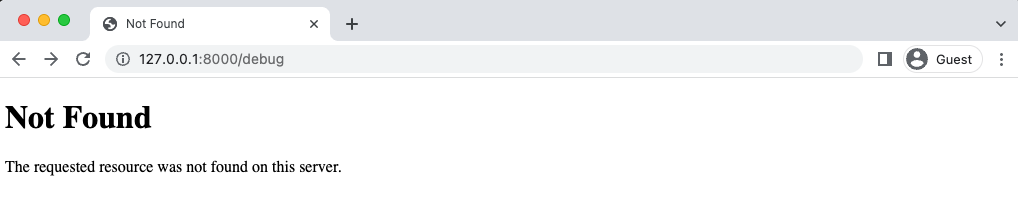
Debug Page

This is great for our purposes as developers, but it is also a roadmap for a hacker in a production setting. When DEBUG is set to False it is required to add a setting for ALLOWED\_HOSTS, which controls the specific hosts or domains that can access the website. We’ll add the two local ports–localhost and 127.0.0.1–as well as .herokuapp.com, which will be used by Render for our production website.

Update the django\_project/settings.py file with two new settings:

d

Then refresh the web page.



Debug Page Not Found

This is the behavior we want for our production site: no information, just a generic message. When we deploy the website we’ll use an elegant way to toggle between the two settings, but for now change DEBUG to an environment variable called DJANGO\_DEBUG.

# django\_project/settings.py

DEBUG = env.bool("DJANGO\_DEBUG") # new

Then proceed to update docker-compose.yml so DJANGO\_DEBUG is set to True.

**DATABASES**

When we installed environs[django] earlier, the Django “goodies” included the elegant djdatabase-url package, which takes all the database configurations needed for our database, SQLite or PostgreSQL. This will be very helpful later on in production.

For now, we can set it to use PostgreSQL locally by adding a default value. Update the existing DATABASES configuration with the following:

# django\_project/settings.py

DATABASES = {

"default": env.dj\_db\_url("DATABASE\_URL",

default="postgres://postgres@db/postgres")

}

The environment variable, DATABASE\_URL, will be created by Heroku when we deploy. More on that later. Refresh the website to confirm everything still works properly.

**Git**

We made a number of important changes in this chapter so make sure to commit the code updates with Git.

If any issues crop up, compare your files against the official source code on Github.

git

git status

git add "Chapter 09. Environment Variables/"

git commit -m "Chapter 09: Environment Variables"

**To push to GitHub if you have an account:**

1. run: cd ..

Your path should be where you root depository is(in my case Bookstore-by-Chapter):

PS C:\Users\computer’s username\Documents\your main folder\Bookstore-by-Chapter>

Yours would be different like:

PS C:\Users\computer’s username\Documents\your main folder\title of your subfolder>

git add "Chapter 09. Environment Variables"

git commit -m "Chapter 09. Environment Variables"

git push

**Create a back up:**

Copy-Item -Recurse -Path "C:\Users\Jean-Marc H\Documents\Django for professionals\Bookstore-by-Chapter\Chapter x. chapter’s title" -Destination "C:\Users\Jean-Marc H\Documents\Django for professionals\Bookstore-by-Chapter\ Chapter x. chapter’s title - Backup"

**Conclusion**

Adding environment variables is a necessary step for any truly professional Django project. They take some getting used to but are invaluable for switching between local and production environments as we’ll do later on in the book. In the next chapter we’ll fully configure our email settings and add password reset functionality.

The end.