# djangoku

A straightforward guide to deploying Django projects on Heroku

# Table of Contents

# djangoku

# Introduction

There are many tutorials on Django. But deployment is often not part of it.

So one is left with blog posts and documentation pages. The information is there, but it is cumbersome to connect all the dots.

With this book I try to provide a concise tutorial on how to deploy Django projects via GitLab to Heroku.

Every chapter ends with a checklist. Once you've completed all steps for a project, you can use the checklists as quick references.

> When I started learning programming, I often abandonded tutorials and books, because *my* code didn't work like the one in the book.

I want you to succeed and finish this book. If your code does not work as expected or you encounter a roadblock, don't hesitate to contact me and I will try to help out.

# What you need to start

To take the most of this book you need basic knowledge in Python, Django and Git. That said the setup you will encounter in the next chapters is kept as basic as possible, so even beginners can follow along.

This book will be based on the version stated in the list below. It is possible to follow the steps with different versions. The general steps will be the same, but there might be small differences.

## Prerequisites

- Python 3.8+
- Git
- PostgreSQL 11+
- GitLab OR GitHub Account
- Heroku Account

> This version of the book assumes you are using MacOS. You can follow this book on Windows and Linux as well by adjusting the commands accordingly.

## Python and Git

Make sure you have Python and Git installed on your system. The easiest way is to google `install python 3 on mac` and `install git on mac`. Chances are that Python and Git are already available on your system. While the Git version is not that important, your Python version should be version 3 or above.

## PostgreSQL

Generally speaking you can work with any database that Django and Heroku support. (Note: SQLite is not supported). This book is based on PostgreSQL, because you don't need any additional elements on Heroku and it is easy to setup on Django. Since we are only setting up the database and not do *specific* interactions with it, the PostgreSQL version you are using is not significant. To see if your version works with Heroku, see this list.

## GitLab account

To follow along you need an GitLab account.

Why GitLab and not Github?

Usually it does not matter *that much* if you use GitLab or GitHub. GitHub is comfortably integrated in Heroku. You can even create GitHub actions to create similar workflows for your Heroku deployment, as we do with GitLab in this book. I personally find working with the GitLab CI/CD pipeline more convenient.

## Heroku account and Heroku CLI

Last but not least, you obviously need an Heroku account to deploy your Django app on Heroku. The good thing about Heroku is, that it is free (how we use it) and scalable (if needed).

Please make also sure that you have installed the Heroku Commandline Interface (CLI).

## Checklist

### ✔ Python 3+ is installed

Terminal (any window)

```
python --version
```

→ Outputs python version 3+

> Depending on how you installed Python 3, you have to use `python3 --version` or even `python3.8 --version` instead.

## ✔ Git is installed

Terminal (any window)

```
git --version
```

→ Outputs any Git version

## ✔ PostgreSQL is installed

Terminal (any window)

```
postgres --version
```

→ Outputs any PostgreSQL version. Ideally 11+

## ✔ GitLab or GitHub account created

## ✔ Heroku account created

## ✔ Heroku CLI installed

Terminal (any window)

```
heroku --version
```

→ Outputs any Heroku CLI version

# Starting the project

Now that all the prerequisites are fullfilled it is time to start the *real* work. We will create our project folder, a local Git repository and a virtual environment. Then we will add Django to the mix.

## Beginning with the basics

Git will keep track of our file versioning and will be our gateway to distribute the files to GitLab and Heroku later. By using a virtual environment we have an isolated sandbox avoiding confusion with Python versions and modules that may be installed on the operating system already. Both Git and a virtual environment are mandatory for a streamlined Heroku workflow. But it is generally a good practice to start any Python projects that way.

> In this book the project will be called "*djangoku_dev*" and you will see that the root path to the project is `djangoku_dev`. Obviously you will have your custom path and maybe a more suitable project name – Keep in mind that you have to adjust some commands accordingly.

## Create the project folder

Create a project folder and navigate with the terminal to it:

Terminal (any window)

```
mkdir djangoku_dev
cd djangoku_dev
```

> If you are on the Mac, there is an easy way to do this: Create the project folder in the Finder, open a new Terminal window, write `cd`, drag the folder from the Finder into the Terminal window and hit *Enter*

## Initiate a new Git repository

Initiate a new Git repository:

Terminal

```
git init
```

It is a good idea to create a Readme file at the beginning of a project. We will create this file later when we push our repository to GitLab. Feel free to create it now and add information along the way. Do not add any sensitive data to it.

## Create the virtual environment

If you are confused about virtual environments (as I still am sometimes), check out this great article on Real Python to get an overview.

In this book the virtual environment will be called *virtualenv*. You can name your virtual environment any way you want and adjust the commands accordingly.

Create a new virtualenvironment in your project folder and activate it:

Terminal

```
python3 -m venv virtualenv
source virtualenv/bin/activate
```

You can see that you have activated the virtual environment if the project foldername is between in parenthesis e.g. *(virtualenv)*

This virtual environment folder is needed only locally. GitLab and Heroku will create their own environments later. Therefore it is important not to track the virtual environment folder with Git.

Create a *.gitignore* file and add *virtualenv* to it:

Terminal

```
echo virtualenv > .gitignore
```

## Adding your first commit

Now that we have created the base layer for our project, we should do our first commit.

Stage all changed files – that's just the *.gitignore* file as we ignored everything else – and commit them:

Terminal

```
git add .
git commit -m "Start djangoku 🌱"
```

## Django. Finally!

After all this preparation and groundwork it is finally time to add Django.

## Install Django

Make sure the virtualenvironment is still activated by checking if the project folder is between in parenthesis.

Install the Django module:

Terminal

```
pip install django
```

## Start the Django project

Now that the Django module was installed, we can start the actual Django project. Note that the command has a `.` at the end – this will make sure we create the Django project in the current directory and not create a new one.

Start the Django project:

Terminal

```
django-admin.py startproject djangoku .
```

If you look into your root project folder you should see that a file called *manage.py* and a *djangoku* folder were added.

Run Django:

Terminal

```
python manage.py runserver
```

After you have run the command, you should see an address like this: `http://127.0.0.1:8000/`. Open this in your browser and see if Django is running successfully.

To stop the Django server press `Ctrl+c` while being in the Terminal.

## Ignore the SQLite database

Django creates an SQLite database automatically. That's the *db.sqlite3* file in the root folder. We will not need this file and will delete it later, because we will work with a PostgreSQL database. For now it is enough to not track the database file and add it to *.gitignore*. When deleted without declaring an alternative database at the same time, Django just recreates *db.sqlite3* everytime the server runs.

Note that the command has two `>`, because we want to append a line to the *.gitignore* file.

> There are other Django projects, where a SQLite database is good enough. Even then you should always ignore *.sqlite3* files.

Terminal

```
echo *.sqlite3 >> .gitignore
```

## Commit

Track the new files in Git:

Terminal

```
git add .
git commit -m "Add Django project 🤠"
```

## Freeze!

As long as we don’t push anything to the server we don’t have to bother that much about dependencies. Still this is a good moment to freeze the Python modules we are using at this stage into a *requirements.txt* file. With this file in our repository, GitLab and Heroku will automatically install the exact versions of Python modules we use in their environment.

Let’s create and fill the *requirements.txt* in one go:

Terminal

```
pip freeze > requirements.txt
```

Stage and commit *requirements.txt*:

Terminal

```
git add requirements.txt
git commit -m "Add current project requirements 📜"
```

## Checklist

## ✔ Project is tracked via Git

Terminal

```
git log
```

→ Outputs the last commit messages

## ✔ Virtual environment is activated

Terminal
→ The root project folder is wrapped in parenthesis in the Terminal.

## ✔ Virtual environment uses the correct Python

Terminal

```
which python
```

→ Outputs a path that leads into the virtual environment folder.

## ✔ Django is installed

Terminal

```
python -m django --version
```

→ Outputs Django version.

## ✔ Django works

Browser (any window)
→ Visit the URL that Django showed on startup (usually `http://127.0.0.1:8000`) and see if it shows the Django success message.

# Adding a page

Currently we see a placeholder page when we visit the Django project in the browser. As a wireframe for future additions, let’s create a minimal app in Django.

The app will serve a welcome page when visiting the root of our project and contain some styling. With the welcome page we can see if Django works correctly even when the debug mode is deactivated. With the added styling, we will see if static files are served correctly.

## Create the app

The purpose of this app is to serve static pages – so we name it *pages*.

Create the *pages* app:

Terminal

```
python manage.py startapp pages
```

Add the *pages* app to `INSTALLED_APPS` in the settings file:

djangoku/settings.py (Around line 45)

```
INSTALLED_APPS = [
    # Other installed apps like django.contrib.*
    'pages',
]
```

## Add urls

Now that the *pages* app is present, we need to tell Django that we want to serve the *pages* urls on root level.

Include `pages.urls` to `urlpatterns` of the main *djangoku* urls file:

djangoku/urls.py

```
from django.contrib import admin
from django.urls import include, path

# Do not forget to import `include` from django.urls
urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('pages.urls'))
]
```

Create an urls file for the *pages* app:

Terminal

```
touch pages/urls.py
```

Add `welcome` path to *urls.py*:

pages/urls.py

```
from django.urls import path

from . import views

urlpatterns = [
    path('', views.welcome, name='welcome'),
]
```

## Add welcome view

The root of our project now directs to the *welcome* view. This view does not exist yet; let’s create it!

Add `welcome` function to *pages/views.py*

pages/views.py (Around line 4)

```
def welcome(request):
    content = {
        "welcome_msg": "Hello!",
        }
    template = "pages/welcome.html"
    return render(request, template, content)
```

## Add the welcome page template

The view currently tries to render a template that does not exist yet. To keep the modular structure of our project, we create it in a subdirectory of the *pages* app.

Create the subdirectory for *pages* templates and create the *welcome* template:

Terminal

```
mkdir -p pages/templates/pages
touch pages/templates/pages/welcome.html
```

> The repetition of the app name in the path of the template is to prevent confusion between templates of the same name in different apps.

Add content to the *welcome* template:

pages/templates/pages/welcome.html

```
{% load static %}

<! DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>djangoku</title>
    <link rel="stylesheet" href="{% static 'css/pages.css'

</head>

<body>
    <h1>{{welcome_msg}}</h1>
</body>
</html>
```

## Add the CSS file

As you can see we referenced a css file that does not exist yet.

Create *pages.css*:

Terminal

```
mkdir -p pages/static/css
touch pages/static/css/pages.css
```

Add basic styling to *pages.css*:

pages/static/css/pages.css

```
body {
    background-color: ghost;
}

h1 {
    text-align: center;
    font-family: monospace;
}
```

## Commit the code

Now we can stage and commit our changes:

Terminal

```
git add .
git commit -m "Add pages app 📒"
```

## Checklist

## ✔ App is created

## ✔ App shows a welcome page

## Handling static files

At one point or another you want to add assets like CSS files, JavaScript files or images to your project.

The advised solution is to store the files externally (like an AWS S3 bucket). For prototypes or smaller projects it is easier to let Django serve the static files on production as well. While Django handles static files locally just fine, the Heroku app needs a bit of configuration.

### Add `django-heroku`

With django-heroku Heroku provides a handy Python package that helps dealing with configuration settings for Heroku. Especially the handling of settings, static files and logging is much easier with it. All required dependencies are downloaded automatically for us.

Install `django-heroku`:

Terminal

```
pip install django-heroku
```

Add it to our *settings.py* file:

djangoku/settings.py (At the bottom of the file)

```
import django_heroku
django_heroku.settings(locals())
```

### Update our requirements

Now that we have two new modules in our project, let's update our requirements.

Update *requirements.txt*:

Terminal

```
pip freeze > requirements.txt
```

## Commit the code

If you run `git status` you will see, that we updated *requirements.txt* and *settings.py*.

Now we can stage and commit our changes:

Terminal

```
git add .
git commit -m "Configure static files handling 📂"
```

## Checklist

## ✔ Collectstatic works

## ✔ django-heroku is part of requirements

# Taking care of the Environment

Now that the basic local setup is in place, it is time to enhance it a bit. In this chapter we will start to make use of environment variables.

## Create and ignore an *.env* file

Important data should not live in your code or be part of the version control. That's where environment variables come in handy. Another advantage is, that we can set different values in different environments (hence the name).

> To learn more about best practices for web apps, check out the twelve-factor methodology

Locally we will store the environment variables in an *.env* file. Later on GitLab and Heroku, those variables can be defined in the settings.

Create an *.env* file:

Terminal

```
touch .env
```

Let's ignore the *.env* file to prevent Git from tracking the content of it.

Add the *.env* file to *.gitignore*:

Terminal

```
echo .env >> .gitignore
```

## Load necessary modules

To easily load and get environment variables in Django, we use the `python-dotenv` module.

Add `python-dotenv` to the project:

Terminal

```
pip install python-dotenv
```

Import `os` and `python`dotenv` to the Django settings and load it:

djangoku/settings.py (Beginning of the file)

```
import os
from dotenv import load_dotenv

load_dotenv()
```

## Create an easy DEBUG switch

The Django debug mode should be turned off in production. Locally or in a staging environment it might be useful to see the Django debug messages. That's why we want to set the debug mode for the environment separately.

Change the value of `DEBUG` from `True` to this:

djangoku/settings.py::23

```
DEBUG = os.getenv('DEBUG', default=False) == "True"
```

This statement does look a bit odd, so let's see what it does:

1. `os.getenv` tries to get the `DEBUG` env variable.
2. If `DEBUG` is not set as an env variable, `DEBUG` defaults to `False`.
3. If the env variable is set to `"True"`, `DEBUG` is set to `True`.

> Mind that environment variables are read as strings. Therefore we must check for a `"True"` string, not a boolean.

When you run Django server now you will see that Django does not run in debug mode. (You may even see this: `CommandError: You must set settings.ALLOWED_HOSTS if DEBUG is False.` ) Locally we want to continue to see debug messages.

Add `DEBUG` variable to the *.env* file and set it to `True` :

'.env'

```
echo "export DEBUG=True" >> .env
```

When you re-run the Django server you will see that Django does run in debug mode again.

## Add another secret to the *.env* file

> The secret key must be a large random value and it must be kept secret.

— Django Documentation, Deployment Checklist

> See this (*stack overflow* answer) to learn more about the purpose of the secret key.

When you have a look in the settings file around *line 23* you see the current value of Django's `SECRET_KEY` .

djangoku/settings.py::23

```
# SECURITY WARNING: keep the secret key used in production
SECRET_KEY = 'mp#qzy1(u!!)ch!s7-d@4rybwz2-r5^l%cy699=v=uyq
```

Since we already added this file to Git, we need to create a new `SECRET_KEY` , add it to our *.env* file as a variable and reference it in the settings file.

Start the Django shell in the terminal:

Terminal

```
python manage.py shell
```

Terminal

```
from django.core.management.utils import get_random_secret_

get_random_secret_key()
```

Copy the output (e.g. `'NEWLY%G3GENERATEDsecret_KEY!'`) and add it to the *.env* file:

'.env'

```
export SECRET_KEY='NEWLY%G3GENERATEDsecret_KEY!'
```

Replace the `SECRET_KEY` string with a reference to the environment variable:

djangoku/settings.py (Beginning of the file)

```
SECRET_KEY = os.getenv('SECRET_KEY')
```

Run `python manage.py runserver` to see if the server starts as expected.

## Commit updates

When you run `git status` you will see, that we changed *.gitignore* and *djangoku/settings.py*.

> You should not see the *.env* file listed when you run `git status`. If you do, check if it was added correctly to *.gitignore*.

Before we commit our changes, we have to update *requirements.txt*:

Terminal

```
pip freeze > requirements.txt
```

When you run `git status` you should see that *requirements.txt* is part of the modified files list.

Now we can stage and commit all changes:

Terminal

```
git add .
git commit -m "Work with env variables 🔐"
```

## Checklist

**Objective**: The project works with an **_.env** file to load environment variables.

### ✔ *.env* exists

Terminal

```
cat .env
```

→ Outputs the contents of the *.env* file including `DEBUG` and `SECRET_KEY`

### ✔ *.env* is not tracked in Git

Terminal

```
git ls-files --error-unmatch .env
```

→ Outputs `error: pathspec '.env' did not match any file(s) known to git`

### ✔ *.env* is ignored by Git

Terminal

```
git check-ignore .env
```

→ Outputs `.env`

## ✔ Django can access the env variables

Terminal

```
python manage.py shell
```

Terminal

```
import os
from dotenv import load_dotenv
from django.conf import settings

load_dotenv()

settings.SECRET_KEY == os.getenv('SECRET_KEY')
```

→ Outputs `True`

## Connecting the database

Every environment will have their own database. On Heroku, the database will be created automatically. Locally we have to create the database on our own.

### Creating the local PostgrSQL database

As described before, we ignored the SQLite database to create our own PostgreSQL database. So let’s do this.

Open postgres shell:

Terminal

```
psql
```

> If the `psql` command fails, you may have forgotten to start the Postgres server.

You can use your own database name. I like to use a database name that is a combination of the project name and the environment. That way it is easy to find it when there are multiple databases. And it is clear which environment the database is meant for.

Create the database:

Terminal

```
CREATE DATABASE djangoku_local;
```

> Do not forget the `;` at the end

Quit postgres shell:

Terminal

```
\q
```

## Adding database modules

For Django and Heroku to be able work with PostgrSQL databases, we need two additional modules.

`psycopg2` is a database adapter, that enables Django to *talk* to the PostgreSQL database. `dj-database-url` allows us to use database urls. This is not important in our local environment, but will be on Heroku. Heroku stores the database url in an environment URL, that we access in the Django settings.

Let’s install both packages:

Terminal

```
pip install psycopg2 dj-database-url
```

And update our *requirements.txt* file:

Terminal

```
pip freeze > requirements.txt
```

## Connecting Django and the database

Now that we have the prerequisites for our PostgreSQL database in place, it is time to clean up a bit.

## Deleting the old ...

Finally we can get rid of the *db.sqlite3* database. Just make sure that you first follow the steps to add the PostgreSQL databas below, before restarting the Django server. If you don’t Django will just recreate the file.

Terminal

```
rm db.sqlite3
```

## … adding the new

The data Django needs to connect to the PostgreSQL database is sensitive information. That's one reason why we will store them in our *.env* file. Another advantage of this is that we can connect to different databases on different environments.

We will add a `DATABASE_URL` variable to the *.env* file, that contains all information to connect to the database. It will look like this: `DATABASE_URL=postgres://{user}:{password}@{host_name}:{port}/{database_name}`

Before we add it to the *.env* file, let's look at the segments of the `DATABSE_URL` :

| Segment | Info | Default value |
|---|---|---|
| `user` | The user you want to connect with to the database | `postgres` |
| `password` | The password for the user | \` \` *(no password)* |
| `host_name` | The database host | `localhost` |
| `port` | The database port | `5432` |
| `database_name` | The database name you chose before | `djangoku_local` |

Let's combine the segments to a functional url and add it to *.env*:

.env

```
export DATABASE_URL=postgres://postgres@localhost:5432/dja
```

Now we need to tell Django to use this database instead of the standard SQLite database by importing `dj_database_url` into `djangoku/settings.py` and adjusting the `DATABASES` variable.

Import `dj_database_url`:

djangoku/settings.py (At the beginning of the file)

```
import dj_database_url
```

Replace the current value of `DATABASES` with this:

djangoku/settings.py (Around line 78)

```
DATABASES = {
    'default': dj_database_url.config(conn_max_age=600)
}
```

Try `python manage.py runserver` to see if Django starts as expected.

> If you deleted *db.sqlite3* before starting the server, it *should not* be recreated by Django automatically on launch. If it does, you need to check your settings again.

## Commit updates

If you run `git status` you will see, that we changed *requirements.txt* and *djangoku/settings.py*.

Now we can stage and commit all changes:

Terminal

```
git add .
git commit -m "Connect PostgreSQL database 🗄️"
```

## Checklist

### ✔ db.sqlite3 is deleted

Terminal

```
test -f db.sqlite3 || echo "deleted"
```

→ Outputs `"deleted"`

### ✔ PostgreSQL database is present

Terminal

```
psql --list
```

→ `djangoku_local` is present in the database list output.

### ✔ DATABASE_URL is parsed correctly

Terminal

```
python manage.py shell
```

Terminal

```
from django.conf import settings

settings.DATABASES['default']
```

→ Outputs a dictionary with correct values for `NAME` , `USER` , `HOST` , `PORT`

### ✔ Django runs correctly

Terminal

```
python manage.py runserver
```

→ Development server starts

## Preparing Heroku

What we have so far is a basic Django project. With everything we did so far, we have a decent basis to extend the project by adding a superuser, apps, models, views, templates, etc.

### Create a Heroku app

Verify that you are logged in with the correct Heroku user:

Terminal

```
heroku auth:whoami
```

Choose a unqiue name and region for your Heroku app.

Then create your Heroku app with the *Heroku CLI*:

Terminal

```
# Important: Change {appname} to a unique app name and {re
heroku apps:create {appname} --region {region}
```

### Add a new secret key to Heroku

Heroku relies on env variables the same way we did on your local system. Heroku will set the URL to the database itself. But `SECRET_KEY` must be set by us.

It is a good idea to create a separate `SECRET_KEY` for each environment.

Let’s create a new one the same way as before.

Start the Django shell in the terminal:

Terminal

```
python manage.py shell
```

Terminal

```
from django.core.management.utils import get_random_secret_

get_random_secret_key()
```

Copy the output (e.g. `'NEWLY%G3GENERATEDsecret_KEY!'` ).

Terminal

```
# Important:
# * Change {appname} to your app name
# * CHange {secretkey} to your secret key
heroku config:set -a {appname} SECRET_KEY={secretkey}
```

> You may also set the `DEBUG` env variable. The way we configured it in *settings.py* it defaults to `False` when not set – which is recommended for the app on Heroku.

## Create a Procfile with Gunicorn

All Heroku applications require a *Procfile*. This file must be located in the root of the project. It declares how Heroku should run your application.

For Django we will tell Heroku to run a *Gunicorn* server for us.

Create a *Procfile*:

Terminal

```
echo web: gunicorn djangoku.wsgi > Procfile
```

> If you named your project differently, you need to adjust the command above accordingly.

Install `gunicorn`:

Terminal

```
pip install gunicorn
```

## Expand allowed hosts

Django checks for the *HTTP_HOST header* of requests. It rejects requests coming from urls that are not part of the `ALLOWED_HOSTS` list in the settings.

You need to add the url of your Heroku app and any custom domains that you may use for the project.

Add the url of the Heroku app to *settings.py*:

djangoku/settings.py (Around line 33)

```
# Important:
# Replace {appname} with your Heroku app name

ALLOWED_HOSTS = [
    "{appname}.herokuapp.com",
]
```

## Update our requirements

Now that we have a new module in our project, let’s update our requirements.

Update *requirements.txt*:

Terminal

```
pip freeze > requirements.txt
```

## Commit the code

If you run `git status` you will see, that we updated *requirements.txt* and *settings.py*, and added *Procfile*.

Now we can stage and commit our changes:

Terminal

```
git add .
git commit -m "Prepare Heroku app 🔌"
```

## Checklist

### ✔ Heroku app exists

Terminal

```
heroku apps
```

→ Your app is part of the Heroku apps list

### ✔ SECRET_KEY is set on Heroku

### ✔ Procfile is present

### ✔ Gunicorn is part of requirements

# Pushing to GitLab

So far we have only worked locally. In this chapter we will create a remote repository on GitLab to push our code to.

## Creating the GitLab project

GitLab has the functionality to create a new project by non-existant repositories by *just pushing to them*. The repository is set to *private* by default. You can change that in the GitLab project settings.

> If you feel more comfortable to use the GitLab website, that’s fine too. Just make sure the remote repository is empty, so we can push to it without pulling it first and maybe even running into merge conflicts.

Terminal

```
## Important: Change {username} to your GitLab username
git push --set-upstream git@gitlab.com:{username}/djangoku
```

This command created the project on GitLab and pushed the current state of our local repository to the remote repository. However, it did not *add* the the remote repository.

So, let’s do this:

Terminal

```
## Important: Change {username} to your GitLab username
git remote add origin git@gitlab.com:{username}/djangoku.g
```

## Creating a Readme file

Since our project is still missing a Readme file, now is finally the time to create one. Feel free to add any content you want to the Readme file. Do not add any sensitive data to it, as it will be tracked by Git and may be exposed to the public.

Create a *Readme.md* file:

Terminal

```
echo "# djangoku" > Readme.md
```

This blog post by Raphael Campardou lists what makes a good readme file.

## Commit and push updates

If you run `git status` you will see, that we added *Readme.md*.

Now we can stage, commit and push it to GitLab:

Terminal

```
git add .
git commit -m "Add Readme 🐌"
git push origin master
```

## Checklist

### ✔ Remote repository is added

Terminal

```
git remote -v
```

→ Shows `origin` with url as *fetch* and *push* remote.

### ✔ Remote repository is up to date

Terminal

```
git diff origin/master
```

→ Outputs nothing.

# Building Our Pipeline

To quickly release new features, we need to concentrate on our deployment pipeline. When we push our code to our remote repository, we want GitLab to autodeploy our project to Heroku. In order to do this, we must enhance our Django project to a proper Heroku app and add a *.gitlab-ci.yml* file.

## Adding GitLab CI instructions

Now that our Heroku requirements are fulfilled, we need to tell GitLab to deploy our code to Heroku on push. We do this by adding a CI/CD pipeline to our remote repository. GitLab creates this pipeline automatically when a `.gitlab-ci.yml` file is present in the root of the repository.

Our `.gitlab-ci.yml` contain instructions about connecting with Heroku. The authorization will be done by an env variable named `HEROKU_PRODUCTION_API_KEY`.

## Get your Heroku API key

You can find your Heroku API key (aka. auth token) in your Heroku account settings. You can also retrieve it with the commandline.

Copy the API key to your clipboard:

Terminal

```
heroku auth:token | pbcopy
```

## Add the Heroku API key to GitLab

To add custom env variables to your GitLab repo, we must go to the settings of our repository on gitlab.com.

Browser (any window)

1. Go to your project's **Settings > CI/CD** and expand the **Variables** section.

2. Click the **Add Variable** button.

3. Add a variable named `HEROKU_API_KEY` with your Heroku API key (see above) as value.

> You may need to delete the newline the auth token, when you used `pbcopy` to add it to your clipboard.

## Create the *.gitlab-ci.yml* file

Back in the terminal, create the *gitlab-ci.yml* file:

Terminal

```
touch -gitlab-ci.yml
```

> Don't forget the `.` at the beginning of the file name.

Add the Heroku deploy stage to the *.gitlab-ci.yml* file:

.gitlab-ci.yml

```
---
// Important: Change {appname} to your Heroku app name
heroku:
  type: deploy
  script:
  - apt-get update -qy
  - apt-get install -y ruby-dev
  - gem install dpl
  - dpl --provider=heroku --app={appname} --api-key=$HEROK
---
```

> This is a very basic example of a *.gitlab-ci.yml* file. Feel free to let it run tests or only deploy if you push to a specific branch.

See the GitLab CI/CD documentation for more information.

## Commit the code

If you run `git status` you will see, that we updated *requirements.txt*, and added *Procfile* and *.gitlab-ci.yml*.

Now we can stage and commit our changes:

Terminal

```
git add .
git commit -m "Add deployment pipeline ☁"
```

We did not push our code yet. There will be a special chapter for this.

## Checklist

### ✔ .gitlab-ci.yml is present

## Autodeploy to Heroku

We prepared everything, so our production pipeline to Heroku is triggered by a push to our remote Git repository. So we will only do one thing and then sit and watch.

Push the changes to GitLab:

Terminal

```
git push origin master
```

Now you can visit *CI/CD → Pipeline* on your GitLab project page to see the pipelines running. After that check out *Activity* on your Heroku app page to see the build process.

If everything worked out, you should see the Django placeholer page when you click the *Open app* button on Heroku or visit your project url directly.

### It is live!

Congratulations! You successfully deployed your Django app via GitLab to Heroku.

Now you can work on your project and quickly ship new features — deployment won't be holding you back.