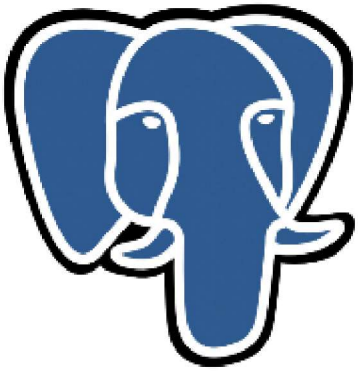
Get started Open in app

Dushan Kumarasinghe
73 Followers · About Follow

Create a web application with python + Flask + PostgreSQL and deploy on Heroku

Dushan Kumarasinghe Aug 29, 2018 · 9 min read



PostgreSQL

Creating an API or Web application using python has been made easy with Flask. It is a micro web framework written in Python.

Here you will create a python server using Flask, create database with PostgreSQL and deploy it on Heroku.

We will create a simple application to store details of books and get stored data to demonstrate database transactions with our python server here.

So here we use,

- python
- Flask
- PostgreSQL
- Heroku CLI
- git

Steps we follow here,

1. Install PostgreSQL to local machine
2. Install Heroku CLI
3. Create python virtual environment for the project
4. Create a sample code with Flask to check
5. Create database
6. Create configurations
7. Database migration
8. Finish the code
9. Commit changes using git and push to Heroku

1. Install PostgreSQL to local machine

Follow this step if you already haven't installed PostgreSQL on your machine.

Install PostgreSQL in Linux using the command,

```
sudo apt-get install postgresql postgresql-contrib
```

Now create a superuser for PostgreSQL

```
sudo -u postgres createuser --superuser name_of_user
```

And create a database using created user account

```
sudo -u name_of_user createdb name_of_database
```

You can access created database with created user by,

```
psql -U name_of_user -d name_of_database
```

> *Note that if you created the* **name_of_user** *and* **name_of_database** *as your user name on your machine, you can access that database with that user with* `psql` *command.*

2. Install Heroku CLI

Method 1: Heroku CLI can be installed using ***snap***.

```
sudo snap install --classic heroku
```

> *Make sure "* /snap/bin" *has been added to path. if not add to path using,* `export PATH="$PATH:/snap/bin"`

Method 2: You can install Heroku CLI using this shell script.

```
curl https://cli-assets.heroku.com/install.sh | sh
```

After installing Heroku CLI using any method above, Verify installation by

```
heroku --version
```

Create a Heroku account if you have not one already *here* and login to your Heroku account in CLI by

```
heroku login
```

3. Create python virtual environment for the project

> ***Why virtual environments?***
> *Virtual environments allow you to make isolated python environment for different projects. It is helpful when several projects need one package in different versions. With virtual environments you can manage packages on current project without considering affecting other projects or without affecting Linux operating system.*

To create virtual environments we need ***virtualenv*** package. Install python virtualenv package using,

```
pip install virtualenv
```

Then create a new directory for the project. Let's say ***books_server***

```
mkdir books_server
cd books_server
```

Create a virtual environment named ***env*** inside the created directory by,

```
virtualenv env
```

This will create the virtual environment named ***env*** inside the ***books_server.***
To activate this environment use this command inside ***books_server*** directory.

```
source env/bin/activate
```

> *You should activate virtual environment when you working with python in this directory for package installation and for running commands in the project directory. When you need to deactivate the virtual environment do it using* `deactivate` *command.*

At this moment we have created the required background for our web server creation and deployment. Let's check how Flask works with below section.

4. Create a sample code with Flask to check

For using Flask, first you need to install Flask. (Make sure that you have activated the virtual environment)

```
pip install Flask
```

Now create a file named ***app.py*** in ***books_server*** directory and put below code to test Flask before we move into real application development

```python
from flask import Flask, request

app = Flask(__name__)

@app.route("/")
def hello():
    return "Hello World!"

@app.route("/name/<name>")
def get_book_name(name):
    return "name : {}".format(name)

@app.route("/details")
def get_book_details():
    author=request.args.get('author')
    published=request.args.get('published')
    return "Author : {}, Published: {}".format(author,published)

if __name__ == '__main__':
    app.run()
```

app.py hosted with ♡ by GitHub view raw

to execute above code run

```
python app.py
```

or

```
FLASK_APP=app.py flask run
```

you can check the deployed server on ***http://127.0.0.1:5000/***

> *Here we have created 3 methods with 3 routes.*
>
> *1. first method is root URL.*
>
> **http://127.0.0.1:5000/** *will return* **Hello World!** *on your browser.*
>
> *another 2 methods are used to get inputs. here we have used 2 types of data input methods.*
>
> **http://127.0.0.1:5000/name**`/Twilight` *will return* **name : Twilight**
>
> **http://127.0.0.1:5000/details?author=**`Stephenie Meyer`**&published=**`2006`
> *will return* **Author : Stephenie Meyer, Published: 2006**
>
> *This* **app.py** *file won't be used in the project. We will create* **app.py** *later as required for the project*

From here let's move to create our book details storing application.

5. Create database

First create the database we need here for our application named ***books_store***

```
sudo -u name_of_user createdb books_store
```

Now you can check the created database with,

```
psql -U name_of_user -d books_store
```

You should log into ***books_store*** data base if above command was success.

6. Create configurations

We need to define configurations for deploying environments. create a file named ***config.py*** with below code.

```python
import os
basedir = os.path.abspath(os.path.dirname(__file__))

class Config(object):
    DEBUG = False
    TESTING = False
    CSRF_ENABLED = True
    SECRET_KEY = 'this-really-needs-to-be-changed'
    SQLALCHEMY_DATABASE_URI = os.environ['DATABASE_URL']


class ProductionConfig(Config):
    DEBUG = False


class StagingConfig(Config):
    DEVELOPMENT = True
    DEBUG = True


class DevelopmentConfig(Config):
    DEVELOPMENT = True
    DEBUG = True


class TestingConfig(Config):
    TESTING = True
```

config.py hosted with ♡ by GitHub view raw

running this in the terminal

```
export APP_SETTINGS="config.DevelopmentConfig"
```

Also add ***"DATABASE_URL"*** to environment variables. In this case our database URL is based on the created database. So, export the environment variable by this command in the terminal,

```
export DATABASE_URL="postgresql://localhost/books_store"
```

It should be returned when you execute `echo $DATABASE_URL` in terminal.
So, now our python application can get database URL for the application from the environment variable which is ***"DATABASE_URL"***

also put these 2 environment variables into a file called ***.env***

```
export APP_SETTINGS="config.DevelopmentConfig"
export DATABASE_URL="postgresql://localhost/books_store"
```

.env hosted with ♡ by GitHub view raw

7. Database migration

Now we need to use ***flask_sqlalchemy*** package for database manipulations. Install it by,

```
pip install flask_sqlalchemy
```

and in ***app.py*** create ***db*** variable using ***SQLAlchemy.***

also set ***app.config*** like below.

```
from flask import Flask, request
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)

app.config.from_object(os.environ['APP_SETTINGS'])
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
```

```
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
db = SQLAlchemy(app)

from models import Book

@app.route("/")
def hello():
    return "Hello World!"
```

app.py hosted with ♡ by GitHub view raw

defined ***db=SQLAlchemy(app)*** will be used to handle the database transactions.

Here we had to import ***Book*** from models. ***models.py*** is described below.

Create a file named ***models.py*** and there we define our tables. Here we define our table as ***books*** and we create the model class as ***Book.***

```
from app import db

class Book(db.Model):
    __tablename__ = 'books'

    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String())
    author = db.Column(db.String())
    published = db.Column(db.String())

    def __init__(self, name, author, published):
        self.name = name
        self.author = author
        self.published = published

    def __repr__(self):
        return '<id {}>'.format(self.id)

    def serialize(self):
        return {
            'id': self.id,
            'name': self.name,
            'author': self.author,
            'published':self.published
        }
```

models.py hosted with ♡ by GitHub view raw

We have defined ***books*** table with the columns ***[id, name, author, published]***

> *Note that* **serialize** *method here is not needed for database migration but it will be useful when we need to return book objects in response as JSON.*

another requirement for database migration is ***manage.py*** file. Create a file named ***manage.py***

```
from flask_script import Manager
from flask_migrate import Migrate, MigrateCommand

from app import app, db

migrate = Migrate(app, db)
manager = Manager(app)

manager.add_command('db', MigrateCommand)


if __name__ == '__main__':
    manager.run()
```

manage.py hosted with ♡ by GitHub view raw

In ***manage.py*** file we use 2 more packages ***flask_script, flask_migrate.***
Also we need ***psycopg2-binary*** package. Install them by,

```
pip install flask_script

pip install flask_migrate

pip install psycopg2-binary
```

Now we can start migrating database. First run,

```
python manage.py db init
```

This will create a folder named ***migrations*** in our project folder. To migrate using these created files, run

```
python manage.py db migrate
```

This should give a result like this.

```
INFO  [alembic.runtime.migration] Context impl PostgresqlImpl.
INFO  [alembic.runtime.migration] Will assume transactional DDL.
INFO  [alembic.autogenerate.compare] Detected added table 'books'
  Generating /home/dushan/testing/books_server/migrations/versions/e6d049f35de3_.py ... done
```

Now apply the migrations to the database using

```
python manage.py db upgrade
```

This will create the table books in ***books_store*** database. you can check it in PostgreSQL command line inside ***books_store*** database by `\dt` command.

```
books_store=# \dt
              List of relations
 Schema |      Name       | Type  | Owner
--------+-----------------+-------+--------
 public | alembic_version | table | dushan
 public | books           | table | dushan
(2 rows)
```

> *In a case of migration fails to be success try droping auto generated* **alembic_version** *table by* `drop table alembic_version;`

You can check the columns of the ***books*** table by `\d books` command.

```
books_store=# \d books
                                  Table "public.books"
  Column   |       Type        | Collation | Nullable |              Default
-----------+-------------------+-----------+----------+-----------------------------------
 id        | integer           |           | not null | nextval('books_id_seq'::regclass)
 name      | character varying |           |          |
 author    | character varying |           |          |
 published | character varying |           |          |
Indexes:
    "books_pkey" PRIMARY KEY, btree (id)
```

8. Finish the code

At this moment we have created the required database configurations and database. So now we can focus on database transactions in our python server. Here we will focus on *app.py*

```
import os
from flask import Flask, request, jsonify
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)

app.config.from_object(os.environ['APP_SETTINGS'])
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
db = SQLAlchemy(app)

from models import Book

@app.route("/")
def hello():
    return "Hello World!"

@app.route("/add")
def add_book():
    name=request.args.get('name')
    author=request.args.get('author')
    published=request.args.get('published')
    try:
        book=Book(
            name=name,
            author=author,
            published=published
        )
        db.session.add(book)
        db.session.commit()
        return "Book added. book id={}".format(book.id)
    except Exception as e:
            return(str(e))

```

```
@app.route("/getall")
def get_all():
    try:
        books=Book.query.all()
        return  jsonify([e.serialize() for e in books])
    except Exception as e:
            return(str(e))

@app.route("/get/<id_>")
def get_by_id(id_):
    try:
        book=Book.query.filter_by(id=id_).first()
        return jsonify(book.serialize())
    except Exception as e:
            return(str(e))

if __name__ == '__main__':
    app.run()
```

method we created in ***Book*** class in ***models.py*** is used here to provide ***book*** objects as serialized.

Also, as now we have created ***manage.py*** now we can run our server locally by,

```
python manage.py runserver
```

And this will automatically refresh the local server with code changes.

> **http://127.0.0.1:5000/add?name=*Twilight*&author=*Stephenie Meyer*&published=*2006*** *will add the book "Twilight" to the database.*
>
> ***http://127.0.0.1:5000/get/1*** *will return the book details of the book which id is 1 as a JSON.*
>
> ***http://127.0.0.1:5000/getall*** *will return every book that have been stored our database.*

At this moment database + python server is complete for our project as a REST API. But if you need to handle ***html*** files with this application we can further improve it by adding

a folder named ***templates*** to our project root directory and add ***html*** files there.

Let's create ***getdata.html*** with a form and note that the form method is ***POST.***
I have used Bootstrap to make it a smooth web page.

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Document</title>
    <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstra
        crossorigin="anonymous">
</head>

<body>
    <div class="container">

        <div class="container">
            <br>
            <br>

            <div class="row align-items-center justify-content-center">
                <h1>Add a book</h1>
            </div>
            <br>

            <form method="POST">

                <label for="name">Book Name</label>
                <div class="form-row">
                    <input class="form-control" type="text" placeholder="Name of Book" id="name"
                </div>
                <br>
                <div class="form-row">
                    <label for="author">Author</label>
                    <input class="form-control" type="text" placeholder="Author Name" id="author
                </div>
                <br>
                <div class="form-row ">
                    <label for="published">Published</label>
```

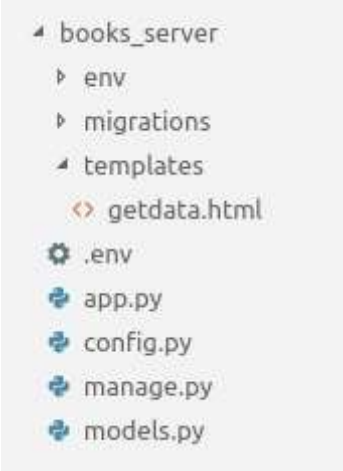
```
                    <input class="form-control " type="date" placeholder="Published" id="publish
                </div>

                <br>
                <button type="submit " class="btn btn-primary " style="float:right ">Submit</but

            </form>
            <br><br>
        </div>
    </div>

    <script src="https://code.jquery.com/jquery-3.3.1.slim.min.js " integrity="sha384-q8i/X+965D
        crossorigin="anonymous "></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.3/umd/popper.min.js " int
        crossorigin="anonymous "></script>
    <script src="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/js/bootstrap.min.js " integr
        crossorigin="anonymous "></script>
</body>

</html>
```

```
import os
from flask import Flask, request, jsonify, render_template
from flask_sqlalchemy import SQLAlchemy


app = Flask(__name__)

app.config.from_object(os.environ['APP_SETTINGS'])
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
db = SQLAlchemy(app)

from models import Book

@app.route("/")
def hello():
    return "Hello World!"

@app.route("/add")
def add_book():
    name=request.args.get('name')
```

```
    author=request.args.get('author')
    published=request.args.get('published')
    try:
        book=Book(
            name=name,
            author=author,
            published=published
        )
        db.session.add(book)
        db.session.commit()
        return "Book added. book id={}".format(book.id)
    except Exception as e:
            return(str(e))

@app.route("/getall")
def get_all():
    try:
        books=Book.query.all()
        return  jsonify([e.serialize() for e in books])
    except Exception as e:
            return(str(e))

@app.route("/get/<id_>")
def get_by_id(id_):
    try:
        book=Book.query.filter_by(id=id_).first()
        return jsonify(book.serialize())
    except Exception as e:
            return(str(e))

@app.route("/add/form",methods=['GET', 'POST'])
def add_book_form():
    if request.method == 'POST':
        name=request.form.get('name')
        author=request.form.get('author')
        published=request.form.get('published')
        try:
            book=Book(
                name=name,
                author=author,
                published=published
            )
            db.session.add(book)
            db.session.commit()
            return "Book added. book id={}".format(book.id)
```

```
66          except Exception as e:
67              return(str(e))
68      return render_template("getdata.html")
69
70  if __name__ == '__main__':
71      app.run()
```

Project structure at the moment should look like below.



9. Commit changes using git and push to Heroku

We need to use ***gunicorn*** package here for Heroku. Either we don't need to use ***gunicorn*** in our local machine, as we can add every package we installed in our project to a file using pip freeze command let's install ***gunicorn*** to our project so we can add it to required packages list for Heroku.

```
pip install gunicorn
```

At the moment we have installed all the required packages for this simple project. We need to specify these packages in ***requirements.txt*** to identify and install when we push our project to Heroku. It can be simply done by,

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

Above command will create the file ***requirements.txt*** in project root directory and add each package we have installed in our virtual environment.

Now we need to create ***Procfile*** to specify our application to Heroku.

```
1  web: gunicorn app:app
```

Procfile hosted with ♡ by GitHub view raw

Also to specify which python version you need to use, create ***runtime.txt***

```
1  python-3.6.5
```

runtime.txt hosted with ♡ by GitHub view raw

Now to push our project into Heroku we should initialize a git repository for the project. Create an git repository by,

```
git init
```

And create a file named ***.gitignore*** in the project root directory.

```
1  .env
2  __pycache__/
3  env/
4  .gitignore
```

.gitignore hosted with ♡ by GitHub view raw

Now add project files to git and commit.

```
git add .

git commit -m "initial commit"
```

At this stage our project file structure should look like this.

Now it is time to create an application on Heroku and push our code. To create an application according to your project use the below command.

```
heroku create name_of_your_application
```

> *This name should be a unique one. So you may have to try several times if the name has already taken by someone. Also if you have 5 applications already running on Heroku at the moment you cannot create another application with the free account. Either you have to delete an existing application from Heroku or upgrade your account.*
>
> *Here I created the application named* ***books-example***

```
Creating ⬢ books-example... done
https://books-example.herokuapp.com/ | https://git.heroku.com/books-example.git
```

The link here in green color is the URL for our application and yellow color link is the git remote link on Heroku.

Now we can add this git remote link to our local git repository. Let's name our remote as ***prod*** for the meaning of production.

```
git remote add prod heroku_git_url
```

> *In my case it is ,*
>
> *git remote add prod https://git.heroku.com/books-example.git*

Now set the configurations for Heroku application through Heroku CLI.

First set "***APP_SETTINGS***" environment variable.

```
heroku config:set APP_SETTINGS=config.ProductionConfig --remote prod
```

Then add Postgres addon to Heroku server by,

```
heroku addons:create heroku-postgresql:hobby-dev --app
name_of_your_application
```

> *In my case it is,*
>
> *heroku addons:create heroku-postgresql:hobby-dev --app books-example*

You can check variables you set for Heroku server by,

```
heroku config --app name_of_your_application
```

> *In my case it is,*
>
> *heroku config --app books-example*
>
> *And it should return something like this with both **"APP_SETTINGS"** and **"DATABASE_URL"**.*

```
=== books-example Config Vars
APP_SETTINGS: config.ProductionConfig
DATABASE_URL: postgres://ysadtycylwgnln:4f25cb04
```

Now it is time to push our code into Heroku. It is done by using git.

```
git push prod master
```

This will install required packages and deploy your application on Heroku. You can check deployed application through the URL Heroku provided. But still there is one more thing to do. You cannot add or get data through database. That's should happen because we still didn't migrate the database in Heroku.

Run migrations by,

```
heroku run python manage.py db upgrade --app name_of_your_application
```

This should give an output like below

```
INFO  [alembic.runtime.migration] Context impl PostgresqlImpl.
INFO  [alembic.runtime.migration] Will assume transactional DDL.
INFO  [alembic.runtime.migration] Running upgrade  -> da7e8baa30a9, empty message
```

> *If the output gives errors make sure that your migrations folder contains the required files. In a situation like this you can*
> ***delete migrations folder=> drop books and alembic_version tables=>rerun the database init,migration and upgrade commands locally .Then add,commit and push changed migrations*** *folder to Heroku prod.*

That's all for this tutorial.
Deployed project is here. https://books-example.herokuapp.com/

You can find the source code for the project here.

dushan14/books-store

Contribute to dushan14/books-store development by creating an

Contribute to dushan14/books-store development by creating an

account on GitHub.
github.com

Python Flask Postgresql Heroku

About Help Legal

Get the Medium app

Download on the App Store GET IT ON Google Play