CONTENTS

Tutorial Series: How To Build Web Applications with Flask

7/13 How To Use a PostgreSQL ... 8/13 How To Use MongoDB in a ...

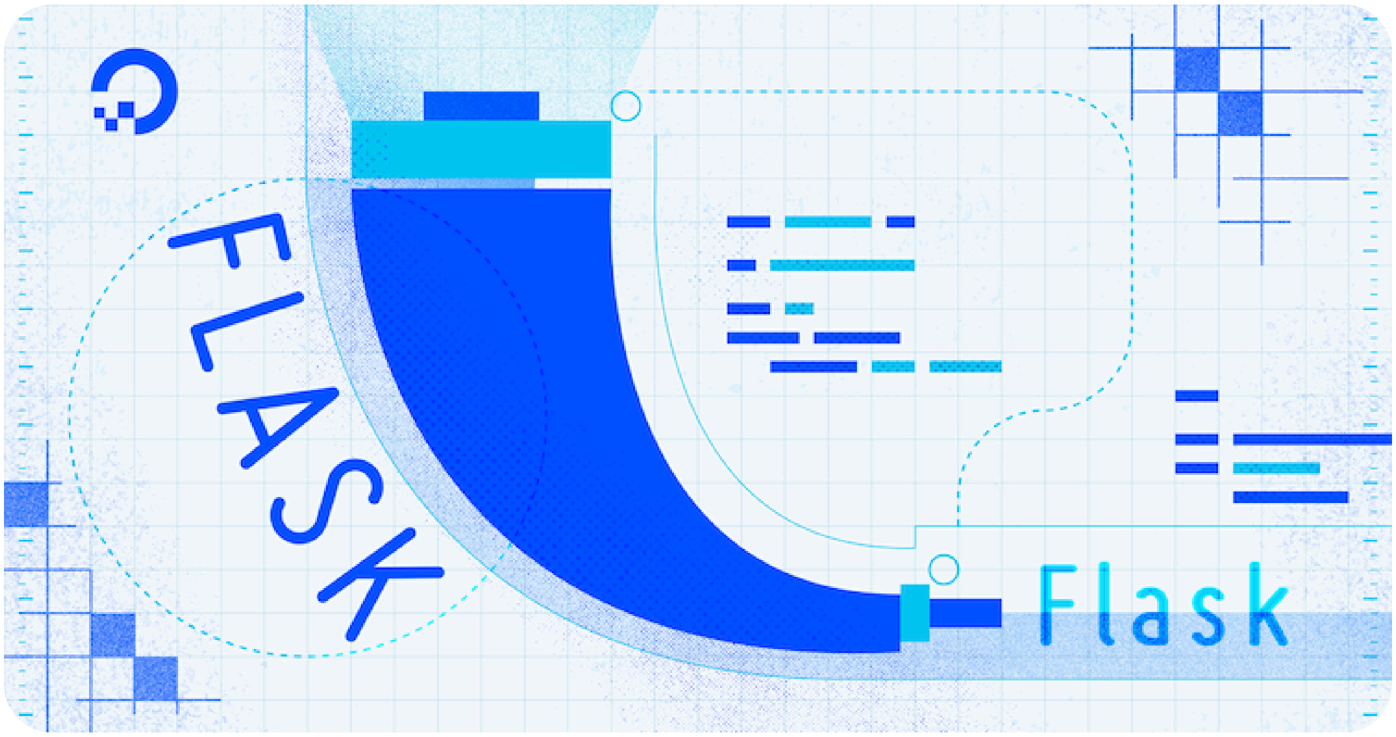
// TUTORIAL //

How To Use a PostgreSQL Database in a Flask Application

Published on January 25, 2022

Databases Python Python Frameworks Flask Development PostgreSQL

By Abdelhadi Dyouri



The author selected the Free and Open Source Fund to receive a donation as part of the Write for DOnations program.

Introduction

In web applications, you usually need a database, which is an organized collection of data. You use a database to store and maintain persistent data that can be retrieved and manipulated efficiently. For example, in a social media application, you have a database where user data (personal information, posts, comments, followers) is stored in a way that can be efficiently manipulated. You can add data to a database, retrieve it, modify it, or delete it, depending on different requirements and conditions. In a web application, these requirements might be a user adding a new post, deleting a post, or deleting their account, which might or might not delete their posts. The actions you perform to manipulate data will depend on specific features in your application. For example, you might not want users to add posts with no titles.

Flask is a lightweight Python web framework that provides useful tools and features for creating web applications in the Python Language. PostgreSQL, or Postgres, is a relational database management system that provides an implementation of the SQL querying language. It's standards-compliant and has many advanced features such as reliable transactions and concurrency without read locks.

In this tutorial, you'll build a small book review web application that demonstrates how to use the library, a PostgreSQL database adapter that allows you to interact with your PostgreSQL database in Python. You'll use it with Flask to perform basic tasks, such as connecting to a database server, creating tables, inserting data to a table, and retrieving data from a table.

Prerequisites

- A local Python 3 programming environment. Follow the tutorial for your distribution in How To Install and Set Up a Local Programming Environment for Python 3 series. In this tutorial the project directory is called `flask_app`.
- An understanding of basic Flask concepts, such as routes, view functions, and templates. If you are not familiar with Flask, check out How to Create Your First Web Application Using Flask and Python and How to Use Templates in a Flask Application.
- An understanding of basic HTML concepts. You can review our How To Build a Website with HTML tutorial series for background knowledge.
- PostgreSQL installed on your local machine, and access to the PostgreSQL prompt. Follow How To Install and Use PostgreSQL on Ubuntu 20.04 to set up your PostgreSQL database.

Step 1 – Creating the PostgreSQL Database and User

In this step, you'll create a database called `flask_db` and a database user called `sammy` for your Flask application.

During the Postgres installation, an operating system user named `postgres` was created to correspond to the `postgres` PostgreSQL administrative user. You need to use this user to perform administrative tasks. You can use `sudo` and pass in the username with the `-iu` option.

Log in to an interactive Postgres session using the following command:

```
sammy@localhost:$ sudo -iu postgres psql
```

You will be given a PostgreSQL prompt where you can set up your requirements.

First, create a database for your project:

```
postgres=# CREATE DATABASE  flask_db ;
```

Copy

Note: Every Postgres statement must end with a semi-colon, so make sure that your command ends with one if you are experiencing issues.

Next, create a database user for our project. Make sure to select a secure password:

```
postgres=# CREATE USER sammy WITH PASSWORD 'password';
```

Copy

Then give this new user access to administer your new database:

```
postgres=# GRANT ALL PRIVILEGES ON DATABASE flask_db TO sammy;
```

Copy

To confirm the database was created, get the list of databases by typing the following command:

```
postgres=# \l
```

Copy

You'll see `flask_db` in the list of databases.

When you are finished, exit out of the PostgreSQL prompt by typing:

```
postgres=# \q
```

Copy

Postgres is now set up so that you can connect to and manage its database information via Python using the `psycopg2` library. Next, you'll install this library alongside the Flask package.

Step 2 – Installing Flask and psycopg2

In this step, you will install Flask and the `psycopg2` library so that you can interact with your database using Python.

With your virtual environment activated, use `pip` to install Flask and the `psycopg2` library:

```
(env)sammy@localhost:$ pip install Flask psycopg2-binary
```

Copy

Once the installation is successfully finished, you'll see a line similar to the following at the end of the output:

```
Output

Successfully installed Flask-2.0.2 Jinja2-3.0.3 MarkupSafe-2.0.1 Werkzeug-2.0.2
```

You now have the required packages installed on your virtual environment. Next, you'll connect to and set up your database.

Step 3 — Setting up a Database

In this step, you'll create a Python file in your `flask_app` project directory to connect to the `flask_db` database, create a table for storing books, and insert some books with reviews into it.

First with your programming environment activated, open a new file called `init_db.py` in your `flask_app` directory.

```
(env)sammy@localhost:$ nano init_db.py
```

Copy

This file will open a connection to the `flask_db` database, create a table called `books`, and populate the table using sample data. Add the following code to it:

flask_app/init_db.py

```
import os
import psycopg2

conn = psycopg2.connect(
        host="localhost",
        database="flask_db",
        user=os.environ['DB_USERNAME'],
        password=os.environ['DB_PASSWORD'])
```

Copy

```
# Open a cursor to perform database operations
cur = conn.cursor()

# Execute a command: this creates a new table
cur.execute('DROP TABLE IF EXISTS books;')
cur.execute('CREATE TABLE books (id serial PRIMARY KEY,'
                                 'title varchar (150) NOT NULL,'
                                 'author varchar (50) NOT NULL,'
                                 'pages_num integer NOT NULL,'
                                 'review text,'
                                 'date_added date DEFAULT CURRENT_TIMESTAMP);'
                                 )

# Insert data into the table

cur.execute('INSERT INTO books (title, author, pages_num, review)'
            'VALUES (%s, %s, %s, %s)',
            ('A Tale of Two Cities',
             'Charles Dickens',
             489,
             'A great classic!')
            )


cur.execute('INSERT INTO books (title, author, pages_num, review)'
            'VALUES (%s, %s, %s, %s)',
            ('Anna Karenina',
             'Leo Tolstoy',
             864,
             'Another great classic!')
            )

conn.commit()

cur.close()
conn.close()
```

Save and close the file.

In this file, you first import the module you'll use to access environment variables where you'll store your database username and password so that they are not visible in your source code.

You import the `psycopg2` library. Then you open a connection to the `flask_db` database using the `psycopg2.connect()` function. You specify the host, which is the localhost in this case. You pass the database name to the `database` parameter.

You provide your username and password via the `os.environ` object, which gives you access to environment variables you set in your programming environment. You will store the database username in an environment variable called `DB_USERNAME` and the password in an environment variable called `DB_PASSWORD`. This allows you to store your username and password outside your source code, so that your sensitive information is not leaked when the source code is saved in source control or uploaded to a server on the internet. Even if an attacker gains access to your source code, they will not gain access to the database.

You create a cursor called `cur` using the method, which allows Python code to execute PostgreSQL commands in a database session.

You use the cursor's `execute()` method to delete the `books` table if it already exists. This avoids the possibility of another table named `books` existing, which might result in confusing behavior (for example, if it has different columns). This isn't the case here, because you haven't created the table yet, so the SQL command won't be executed. Note that this will delete all of the existing data whenever you execute this `init_db.py` file. For our purposes, you will only execute this file once to initiate the database, but you might want to execute it again to delete whatever data you inserted and start with the initial sample data again.

Then you use `CREATE TABLE books` to create a table named `books` with the following columns:

- `id`: An ID of the `serial` type, which is an autoincrementing integer. This column represents a *primary key* you specify using the `PRIMARY KEY` keywords. The database will assign a unique value to this key for each entry.
- `title`: The book's title of the `varchar` type, which is a character type of variable length with a limit. `varchar (150)` means that the title can be up to 150 characters long. `NOT NULL` signifies that this column can't be empty.
- `author`: The book's author, with a limit of 50 characters. `NOT NULL` signifies that this column can't be empty.
- `pages_num`: An integer representing the number of pages the book has. `NOT NULL` signifies that this column can't be empty.
- `review`: The book review. The `text` type signifies that the review can be text of any length.
- `date_added`: The date the book was added to the table. `DEFAULT` sets the default value of the column to `CURRENT_TIMESTAMP`, which is the time at which the book was added to the database. Just like `id`, you don't need to specify a value for this column as it will be automatically filled in.

After creating the table, you use the cursor's `execute()` method to insert two books into the table, *A Tale of Two Cities* by Charles Dickens, and *Anna Karenina* by Leo Tolstoy. You use the `%s` placeholder to pass the values to the SQL statement. `psycopg2` handles the insertion in the background in a way that prevents SQL Injection attacks.

Once you finish inserting book data into your table, you use the
method to commit the transaction and apply the changes to the database. Then you clean things up by closing the cursor with `cur.close()`, and the connection with `conn.close()`.

For the database connection to be established, set the `DB_USERNAME` and `DB_PASSWORD` environment variables by running the following commands. Remember to use your own username and password:

```
(env)sammy@localhost:$ export DB_USERNAME=" sammy "
(env)sammy@localhost:$ export DB_PASSWORD=" password "
```

Copy

Now, run your `init_db.py` file in the terminal using the `python` command:

```
(env)sammy@localhost:$ python init_db.py
```

Copy

Once the file finishes execution with no errors, a new `books` table will be added to your `flask_db` database.

Log in to an interactive Postgres session to check out the new `books` table.

```
sammy@localhost:$ sudo -iu postgres psql
```

Copy

Connect to the `flask_db` database using the `\c` command:

```
postgres=# \c flask_db
```

Copy

Then use a `SELECT` statement to get the titles and authors of books from the `books` table:

```
postgres=# SELECT title, author FROM books;
```

Copy

You'll see an output like the following:

```
        title          |      author
-----------------------+------------------
 A Tale of Two Cities | Charles Dickens
 Anna Karenina        | Leo Tolstoy
```

Quit the interactive session with `\q`.

Next, you'll create a small Flask application, connect to the database, retrieve the two book reviews you inserted into the database, and display them on the index page.

Step 4 — Displaying Books

In this step, you'll create a Flask application with an index page that retrieves the books that are in the database, and display them.

With your programming environment activated and Flask installed, open a file called `app.py` for editing inside your `flask_app` directory:

```
sammy@localhost:$ nano app.py
```

Copy

This file will set up your database connection and create a single Flask route to use that connection. Add the following code to the file:

flask_app/app.py

Copy

```
import os
import psycopg2
from flask import Flask, render_template

app = Flask(__name__)

def get_db_connection():
    conn = psycopg2.connect(host='localhost',
                            database=' flask_db ',
                            user=os.environ['DB_USERNAME'],
                            password=os.environ['DB_PASSWORD'])
    return conn
```

```
@app.route('/')
def index():
    conn = get_db_connection()
    cur = conn.cursor()
    cur.execute('SELECT * FROM books;')
    books = cur.fetchall()
    cur.close()
    conn.close()
    return render_template('index.html', books=books)
```

Save and close the file.

Here, you import the `os` module, the `psycopg2` library, and the `Flask` class and the `render_template()` from the `flask` package. You make a Flask application instance called `app`.

You define a function called `get_db_connection()`, which opens a connection to the `flask_db` database using the user and password you store in your `DB_USERNAME` and `DB_PASSWORD` environment variables. The function returns the `conn` connection object you'll be using to access the database.

Then you create a main `/` route and an `index()` view function using the `app.route()` decorator. In the `index()` view function, you open a database connection using the `get_db_connection()` function, you create a cursor, and execute the `SELECT * FROM books;` SQL statement to get all the books that are in the database. You use the `fetchall()` method to save the data in a variable called `books`. Then you close the cursor and the connection. Lastly, you return a call to the `render_template()` function to render a template file called `index.html` passing it the list of books you fetched from the database in the `books` variable.

To display the books you have in your database on the index page, you will first create a base template, which will have all the basic HTML code other templates will also use to avoid code repetition. Then you'll create the `index.html` template file you rendered in your `index()` function. To learn more about templates, see How to Use Templates in a Flask Application.

Create a `templates` directory, then open a new template called `base.html`:

```
sammy@localhost:$ mkdir templates
sammy@localhost:$ nano templates/base.html
```
Copy

Add the following code inside the `base.html` file:

flask_app/templates/base.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title> {% block title %} {% endblock %} - FlaskApp</title>
    <style>
        nav a {
            color: #d64161;
            font-size: 3em;
            margin-left: 50px;
            text-decoration: none;
        }

        .book {
            padding: 20px;
            margin: 10px;
            background-color: #f7f4f4;
        }

        .review {
                margin-left: 50px;
                font-size: 20px;
        }

    </style>
</head>
<body>
    <nav>
        <a href=" {{ url_for('index') }} ">FlaskApp</a>
        <a href="#">About</a>
    </nav>
    <hr>
    <div class="content">
         {% block content %} {% endblock %}
    </div>
```

```
</body>
</html>
```

Save and close the file.

This base template has all the HTML boilerplate you'll need to reuse in your other templates. The `title` block will be replaced to set a title for each page, and the `content` block will be replaced with the content of each page. The navigation bar has two links, one for the index page where you use the `url_for()` helper function to link to the `index()` view function, and the other for an **About** page if you choose to include one in your application.

Next, open a template called `index.html`. This is the template you referenced in the `app.py` file:

```
sammy@localhost:$ nano templates/index.html
```

Copy

Add the following code to it:

flask_app/templates/index.html

Copy

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

{% block content %}
    <h1>{% block title %} Books {% endblock %}</h1>
    {% for book in books %}
        <div class='book'>
            <h3>#{{ book[0] }} - {{ book[1] }} BY {{ book[2] }}</h3>
            <i><p>({{ book[3] }} pages)</p></i>
            <p class='review'>{{ book[4] }}</p>
            <i><p>Added {{ book[5] }}</p></i>
        </div>
    {% endfor %}
{% endblock %}
```

Save and close the file.

In this file, you extend the base template, and replace the contents of the `content` block. You use an `<h1>` heading that also serves as a title.

You use a Jinja loop in the line `{% for book in books %}` to go through each book in the `books` list. You display the book ID, which is the first item using `book[0]`. You then display the book title, author, number of pages, review, and the date the book was added.

While in your `flask_app` directory with your virtual environment activated, tell Flask about the application (`app.py` in this case) using the `FLASK_APP` environment variable. Then set the `FLASK_ENV` environment variable to `development` to run the application in development mode and get access to the debugger. For more information about the Flask debugger, see How To Handle Errors in a Flask Application. Use the following commands to do this:

```
(env)sammy@localhost:$ export FLASK_APP=app
(env)sammy@localhost:$ export FLASK_ENV=development
```

Copy

Make sure you set the `DB_USERNAME` and `DB_PASSWORD` environment variables if you haven't already:

```
(env)sammy@localhost:$ export DB_USERNAME=" sammy "
(env)sammy@localhost:$ export DB_PASSWORD=" password "
```

Copy

Next, run the application:

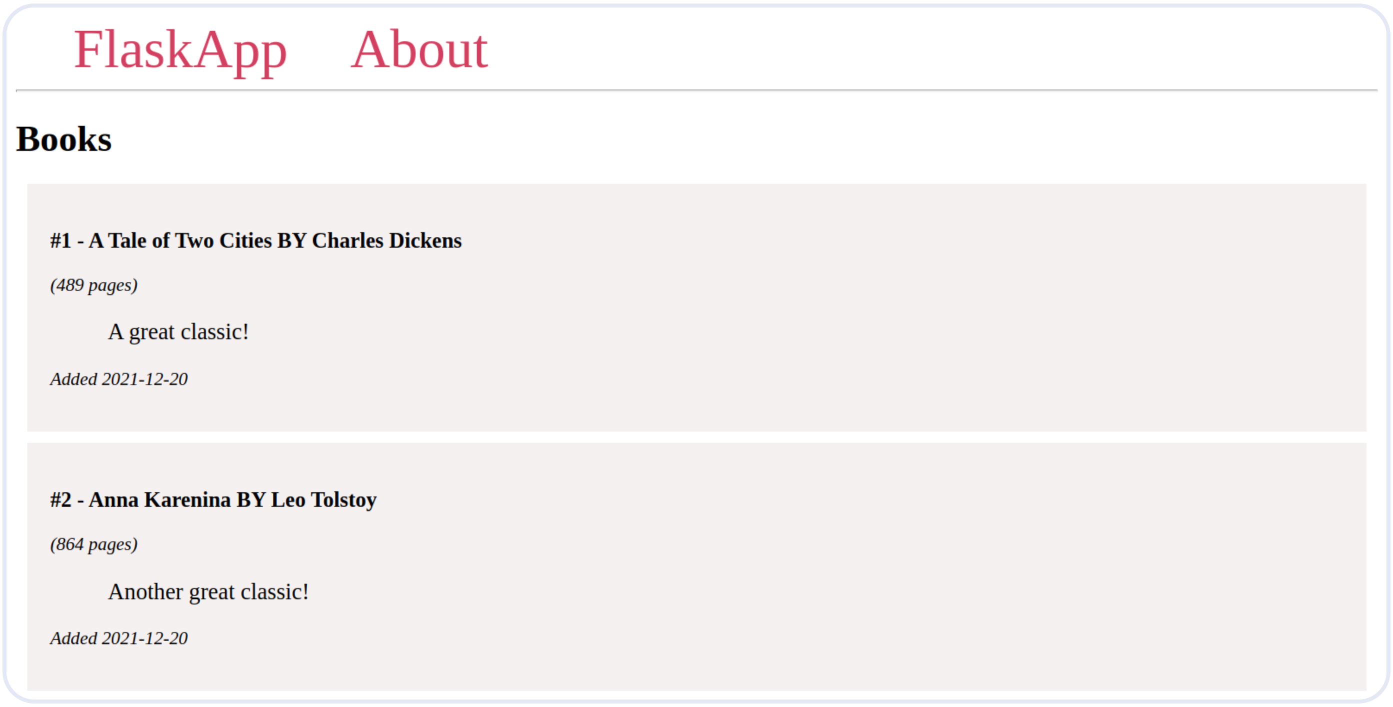
```
(env)sammy@localhost:$ flask run
```

Copy

With the development server running, visit the following URL using your browser:

```
http://127.0.0.1:5000/
```

You'll see the books you added to the database on the first initiation.



Books

#1 - A Tale of Two Cities BY Charles Dickens

(489 pages)

A great classic!

Added 2021-12-20

#2 - Anna Karenina BY Leo Tolstoy

(864 pages)

Another great classic!

Added 2021-12-20

You’ve displayed the books in your database on the index page. You now need to allow users to add new books. You’ll add a new route for adding books in the next step.

Step 5 — Adding New Books

In this step, you’ll create a new route for adding new books and reviews to the database.

You’ll add a page with a web form where users enter the book title, book author, the number of pages, and the book review.

Leave the development server running and open a new terminal window.

First, open your `app.py` file:

```
(env)sammy@localhost:$ nano app.py
```

Copy

For handling the web form, you’ll need to import a few things from the `flask` package:

- The global object to access submitted data.
- The function to generate URLs.
- The function to redirect users to the index page after adding a book to the database.

Add these imports to the first line in the file:

flask_app/app.py

```
from flask import Flask, render_template , request, url_for, redirect

# ...
```

Then add the following route at the end of the `app.py` file:

flask_app/app.py

```
# ...


@app.route('/create/', methods=('GET', 'POST'))
def create():
    return render_template('create.html')
```

Save and close the file.

In this route, you pass the tuple `('GET', 'POST')` to the `methods` parameter to allow both GET and POST requests. GET requests are used to retrieve data from the server. POST requests are used to post data to a specific route. By default, only GET requests are allowed. When the user first requests the `/create` route using a GET request, a template file called `create.html` will be rendered. You will later edit this route to handle POST requests for when users fill and submit the web form for adding new books.

Open the new `create.html` template:

```
(env)sammy@localhost:$ nano templates/create.html
```

Add the following code to it:

flask_app/templates/create.html

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

{% block content %}
    <h1>{% block title %} Add a New Book {% endblock %}</h1>
    <form method="post">
        <p>
            <label for="title">Title</label>
            <input type="text" name="title"
                   placeholder="Book title">
            </input>
        </p>

        <p>
            <label for="author">Author</label>
            <input type="text" name="author"
                   placeholder="Book author">
            </input>
        </p>

        <p>
            <label for="pages_num">Number of pages</label>
            <input type="number" name="pages_num"
                   placeholder="Number of pages">
            </input>
        </p>
        <p>
        <label for="review">Review</label>
        <br>
        <textarea name="review"
                  placeholder="Review"
                  rows="15"
                  cols="60"
                  ></textarea>
        </p>
        <p>
            <button type="submit">Submit</button>
        </p>
    </form>
{% endblock %}
```

Save and close the file.

You extend the base template, set a heading as a title, and use a `<form>` tag with the attribute `method` set to `post` to indicate that the form will submit a POST request.

You have a text field with the name `title`, which you'll use to access the title data in your `/create` route.

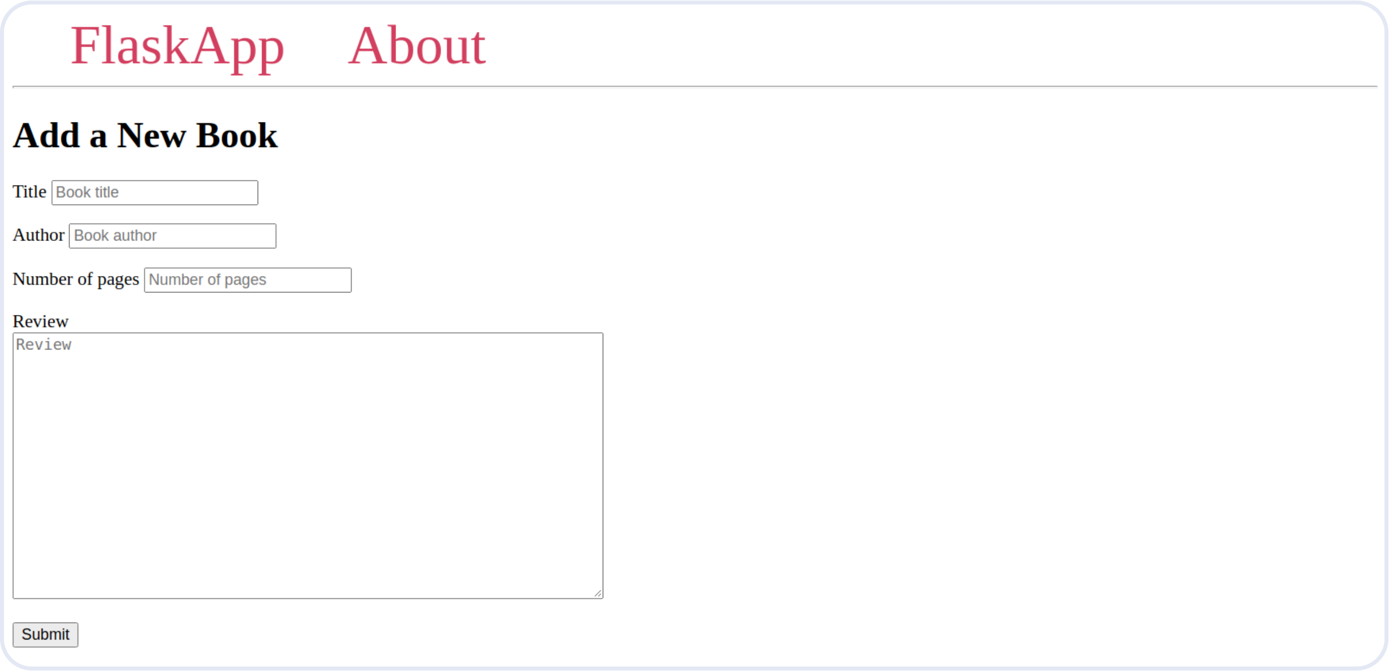
You have a text field for the author, a number field for the number of pages, and a text area for the book review.

Last, you have a **Submit** button at the end of the form.

Now, with the development server running, use your browser to navigate to the `/create` route:

```
http://127.0.0.1:5000/create
```

You will see an **Add a New Book** page with an input field for a book title, one for its author, and one for the number of pages the book has, a text area for the book's review, and a **Submit** button.



If you fill in the form and submit it, sending a POST request to the server, nothing happens because you did not handle POST requests on the `/create` route.

Open `app.py` to handle the POST request the user submits:

```
(env)sammy@localhost:$ nano app.py
```
Copy

Edit the `/create` route to look as follows:

flask_app/app.py

```
# ...

@app.route('/create/', methods=('GET', 'POST'))
def create():
    if request.method == 'POST':
        title = request.form['title']
        author = request.form['author']
        pages_num = int(request.form['pages_num'])
        review = request.form['review']

        conn = get_db_connection()
        cur = conn.cursor()
        cur.execute('INSERT INTO books (title, author, pages_num, review)'
                    'VALUES (%s, %s, %s, %s)',
                    (title, author, pages_num, review))
        conn.commit()
        cur.close()
        conn.close()
        return redirect(url_for('index'))

    return render_template('create.html')
```

Save and close the file.

You handle POST requests inside the `if request.method == 'POST'` condition. You extract the title, author, number of pages, and the review the user submits from the `request.form` object.

You open a database using the `get_db_connection()` function, and create a cursor. Then you execute an `INSERT INTO` SQL statement to insert the title, author, number of pages, and review the user submitted into the `books` table.

You commit the transaction and close the cursor and connection.

Lastly, you redirect the user to the index page where they can see the newly added book below the existing books.

With the development server running, use your browser to navigate to the `/create` route:

```
http://127.0.0.1:5000/create
```

Fill in the form with some data and submit it.

You'll be redirected to the index page where you'll see your new book review.

Next, you'll add a link to the Create page in the navigation bar. Open `base.html`:

```
(env)sammy@localhost:$ nano templates/base.html
```

Copy

Edit the file to look as follows:

flask_app/templates/base.html

Copy

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>{% block title %} {% endblock %} - FlaskApp</title>
    <style>
        nav a {
            color: #d64161;
            font-size: 3em;
            margin-left: 50px;
            text-decoration: none;
        }

        .book {
            padding: 20px;
            margin: 10px;
            background-color: #f7f4f4;
        }

        .review {
                margin-left: 50px;
                font-size: 20px;
        }
    </style>
</head>
<body>
    <nav>
        <a href="{{ url_for('index') }}">FlaskApp</a>
         <a href="{{ url_for('create') }}">Create</a >
        <a href="#">About</a>
    </nav>
    <hr>
    <div class="content">
        {% block content %} {% endblock %}
    </div>
</body>
</html>
```

Save and close the file.

Here, you add a new `<a>` link to the navigation bar that points to the Create page.

Refresh your index page and you'll see the new link in the navigation bar.

You now have a page with a web form for adding new book reviews. For more on web forms, see How To Use Web Forms in a Flask Application. For a more advanced and more secure method of managing web forms, see How To Use and Validate Web Forms with Flask-WTF.

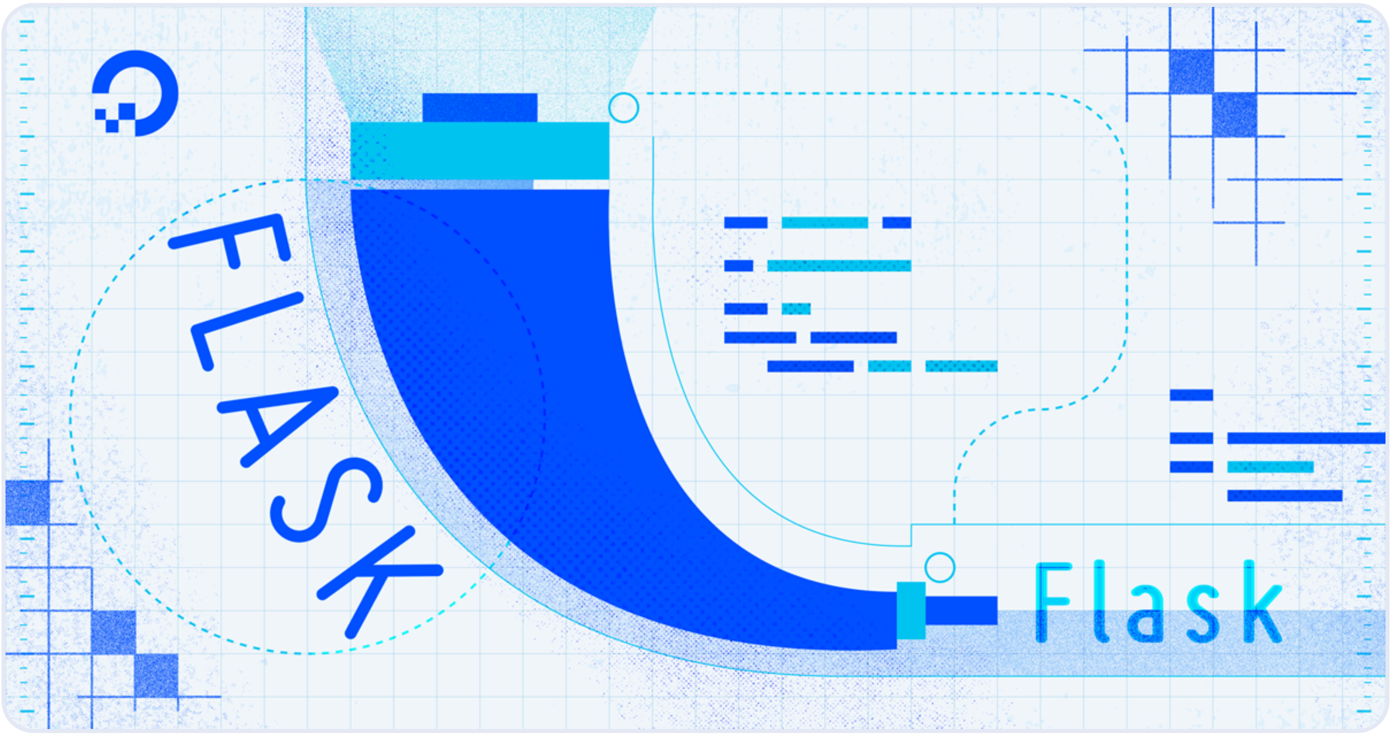
Conclusion

You built a small web application for book reviews that communicates with a PostgreSQL database. You have basic database functionality in your Flask application, such as adding new data to the database, retrieving data, and displaying it on a page.

If you would like to read more about Flask, check out the other tutorials in the Flask series.

Next in series: How To Use MongoDB in a Flask Application →

Tutorial Series: How To Build Web Applications with Flask



Flask is a lightweight Python web framework that provides useful tools and features for creating web applications in the Python Language. It gives developers flexibility and is an accessible framework for new developers because you can build a web application quickly using only a single Python file. Flask is also extensible and doesn’t force a particular directory structure or require complicated boilerplate code before getting started. Learning Flask will allow you to quickly create web applications in Python. You can take advantage of Python libraries to add advanced features to your web application, like storing your data in a database, or validating web forms.

Subscribe

Databases Python Python Frameworks Flask Development PostgreSQL

Browse Series: 13 articles

Expand to view all

About the authors

Abdelhadi Dyouri Author

Still looking for an answer? Ask a question

Search for more help

Was this helpful? Yes No

Comments

1 Comments

Leave a comment...

This textbox defaults to using `Markdown` to format your answer.

You can type `!ref` in this text area to quickly search our full set of tutorials, documentation & marketplace offerings and insert the link!

Sign In or Sign Up to Comment

sidietz • June 6, 2022

Do you really think using plain SQL in a flask application is still a thing in 2022?

Wouldn’t it be better to use a ORM like sqlalchemy instead, would it?

Show replies Reply

Try DigitalOcean for free

Click below to sign up and get **$200 of credit** to try our products over 60 days!

Sign up

Popular Topics

Ubuntu

Linux Basics

JavaScript

Python

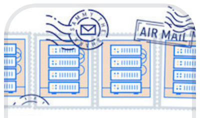
MySQL

Docker

Kubernetes

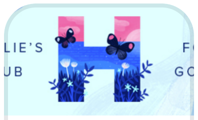
All tutorials →

Talk to an expert →



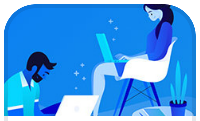
Get o biwe news

Sign up for Infrastru as a Newslett



Hollie Hub for Good

Working on improvin health and educatio reducing inequalit and spurring economi growth? We'd like to help.



Beco contr

Get paid to write technica tutorials and select a tech-focused charity to receive a matching donation

Sign up →

Learn more →

Learn more →

Featured on Community

Kubernetes Course Learn Python 3 Machine Learning in Python

Getting started with Go Intro to Kubernetes

DigitalOcean Products

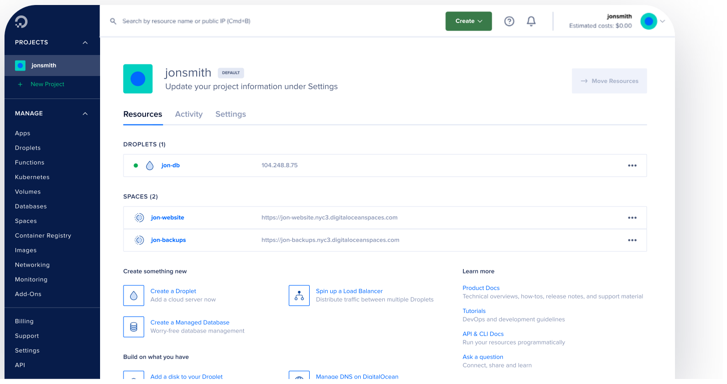
Cloudways Virtual Machines Managed Databases Managed Kubernetes

Block Storage Object Storage Marketplace VPC Load Balancers

Welcome to the developer cloud

DigitalOcean makes it simple to launch in the cloud and scale up as you grow — whether you're running one virtual machine or ten thousand.

Learn more



Get started for free

Sign up and get $200 in credit for your first 60 days with DigitalOcean.

Get started

This promotional offer applies to new accounts only.

Company

Products

Community

Solutions

Contact