



Tutorial Series: How To Build Web Applications with Flask



8/10 How To Use MongoDB in a ...

9/10 How to Use Flask-SQLAlch...



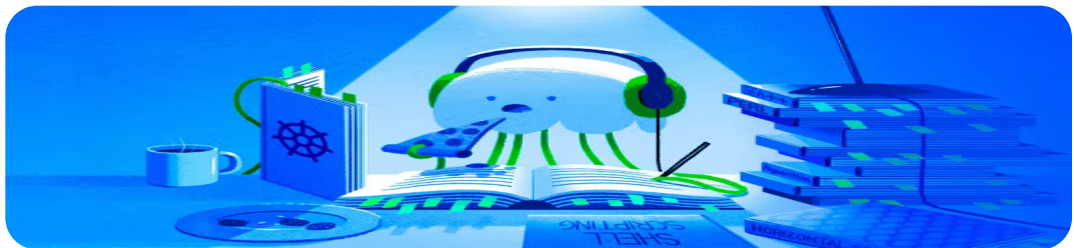
// Tutorial //

How To Use MongoDB in a Flask Application

Published on February 14, 2022

[Databases](#) [Development](#) [Flask](#) [MongoDB](#) [Python](#) [Python Frameworks](#)By [Abdelhadi Dyouri](#)

Developer and author at DigitalOcean.



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 may or may 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. [MongoDB](#) is a general-purpose, document-oriented, [NoSQL](#) database program that uses [JSON](#)-like documents to store data. Unlike tabular relations used in relational databases, JSON-like documents allow for flexible and dynamic schemas while maintaining simplicity. In general, NoSQL databases have the ability to scale horizontally, making them suitable for big data and real-time applications.

In this tutorial, you'll build a small todo list web application that demonstrates how to use the [PyMongo](#) library, a MongoDB database driver that allows you to interact with your MongoDB database in Python. You'll use it with Flask to perform basic tasks, such as connecting to a database server, creating

collections that store a group of documents in MongoDB, inserting data to a collection, and retrieving and deleting data from a collection.

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 we'll call our project directory `flask_app`.
- MongoDB installed on your local machine. Follow the [How To Install MongoDB on Ubuntu 20.04 guide](#) to set up your MongoDB database.
- 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.

Step 1 – Setting Up PyMongo and Flask

In this step, you will install Flask and the PyMongo library.

With your virtual environment activated, use `pip` to install Flask and PyMongo:

```
(env)sammy@localhost:~$ pip install Flask pymongo
```

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 click-8.0.3 it
```

Now that you've installed the required Python packages, you'll connect to your MongoDB Server and create a collection.

Step 2 – Connecting to the MongoDB Server and Creating a Collection

In this step, you'll use the PyMongo library to create a client you'll use to interact with your MongoDB server, create a database, and then create a collection to store your todos.

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

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

Copy

This file will import the necessary class and helpers from Flask and the PyMongo library. You'll interact with your MongoDB server to create a database and create a collection for todos. Add the following code to `app.py`:

flask_app/app.py

```
from flask import Flask
from pymongo import MongoClient

app = Flask(__name__)

client = MongoClient('localhost', 27017)

db = client.flask_db
todos = db.todos
```

Copy

Save and close the file.

Here you import the `Flask` class, which you use to create a Flask application instance called `app`.

You import the `MongoClient` which you use to create a client object for a MongoDB instance called `client`, which allows you to connect and interact with your MongoDB server. When you instantiate the `MongoClient()`, you pass it the host of your MongoDB server, which is `localhost` in our case, and the port, which is `27017` here.

Note:

It is **strongly** recommended that you harden your MongoDB installation's security by following our guide on [How To Secure MongoDB on Ubuntu 20.04](#). Once it's secured, you could then [configure MongoDB to accept remote connections](#).

Once you enable authentication in MongoDB, you'll need to pass additional `username` and `password` parameters when creating an instance of `MongoClient()` like so:

```
client = MongoClient('localhost', 27017, username='username', password='password'). Copy
```

You then use the `client` instance to create a MongoDB database called `flask_db` and save a reference to it in a variable called `db`.

Then you create a collection called `todos` on the `flask_db` database using the `db` variable. Collections store a group of documents in MongoDB, like tables in relational databases.

In MongoDB, databases and collections are created lazily. This means that even if you execute the `app.py` file, none of the code related to the database will actually be executed until the first document is created. You will create a small Flask application with a page that allows users to insert todo documents into your `todos` collection in the next step. Once the first todo document is added, the `flask_db` database and the `todos` collection will be created on your MongoDB server.

To get a list of your current databases, open a new terminal, and start the `mongo` shell using the following command:

```
(env)sammy@localhost:$ mongo Copy
```

A prompt will be opened, you can check your databases using the following command:

```
> show dbs Copy
```

The output, if this is a new installation of MongoDB, will list the `admin`, `config`, and `local` databases.

You'll notice that the `flask_db` doesn't exist yet. Leave the `mongo` shell running in a terminal window and continue to the next step.

Step 3 – Creating a Web Page for Adding and Displaying Todos

In this step, you'll create a web page with a web form that allows users to add todos, and display them on the same page.

With your programming environment activated, open your `app.py` file for editing:

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

First, add the following imports from `flask`:

```
flask_app/app.py
```

```
from flask import Flask, render_template, request, url_for, redirect
from pymongo import MongoClient Copy
```

```
# ...
```

Here, you import the `render_template()` helper function you'll use to render an HTML template, the `request` object to access data the user will submit, the `url_for()` function to generate URLs, and the `redirect()` function to redirect the user back to the index page after adding a todo.

Then add the following route at the end of the file:

```
flask_app/app.py
```

```
# ...
```

Copy

```
@app.route('/', methods=('GET', 'POST'))
def index():
    return render_template('index.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 `/` route using a GET request, a template file called `index.html` will be rendered. You will later edit this route to handle POST requests for when users fill and submit the web form for creating new todos.

Next create a templates folder in your `flask_app` directory, and the `index.html` template you referenced in the preceding route:

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

Copy

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

```
flask_app/templates/index.html
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>FlaskApp</title>
  <style>
    .todo {
      padding: 20px;
      margin: 10px;
      background-color: #eee;
    }
  </style>
</head>
<body>
  <h1>FlaskTODO</h1>
  <hr>
  <div class="content">
    <form method="post">
      <p>
        <b><label for="content">Todo content</label></b>
      </p>
      <p>
        <input type="text" name="content"
          placeholder="Todo Content"></input>
      </p>
      <p>
        <b><label for="degree">Degree</label></b>
      </p>
      <p>
        <input id="degree-0" name="degree" required type="radio" value="Important">
        <label for="degree-0">Important</label>
      </p>
    </form>
  </div>
```

Copy

```

<input id="degree-1" name="degree" required type="radio" value="Unimportant">
<label for="degree-1">Unimportant</label>
</p>
<button type="submit">Submit</button>
</form>
</div>
</body>
</html>

```

Save and close the file.

Here you have a basic HTML page with a title, some styles, a heading, and a web form. In the web form, you set the `method` attribute to `post` to indicate that the form will submit a POST request. You have a text input field for the todo content with the name `content`, which you'll use to access the title data in your `/` route. You also have two HTML radio buttons with the name `degree`, which allows the user to specify the degree of importance for each todo item: they can select either the **Important** option or the **Unimportant** option when creating a todo. Last, you have a **Submit** button at the end of the form.

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

Next, run the application:

```

(env)sammy@localhost:~$ flask run

```

Copy

Note: You might receive a `ModuleNotFoundError: No module named 'pymongo'` error when attempting to run the application. To fix this, deactivate your virtual environment and reactivate it. Then run the `flask run` command again.

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

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

You will see the index page with an input field for todo content, two radio buttons for the degree of importance, and a **Submit** button.

FlaskTODO

Todo content

Degree

☐ Important
 ☐ Unimportant

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](#).

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 `/` route.

Leave the server running and open a new terminal window.

Open `app.py` to handle the POST request the user submits, add them to the `todos` collection, and display them on the index page:

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

Copy

Edit the `/` route to look as follows:

```
flask_app/app.py
```

Copy

```
@app.route('/', methods=('GET', 'POST'))
def index():
    if request.method == 'POST':
        content = request.form['content']
        degree = request.form['degree']
        todos.insert_one({'content': content, 'degree': degree})
        return redirect(url_for('index'))

    all_todos = todos.find()
    return render_template('index.html', todos=all_todos)
```

Save and close the file.

In these changes, you handle POST requests inside the `if request.method == 'POST'` condition. You extract the todo content and degree of importance the user submits from the `request.form` object.

You use the `insert_one()` method on the `todos` collection to add a todo document into it. You provide todo data in a Python dictionary, setting the `'content'` to the value the user submitted in the text field for the todo content, and setting the `'degree'` key to the value of the radio button the user selects. You then redirect to the index page, which will refresh the page and display the newly added todo item.

To display all the saved todos, you use the `find()` method outside the code responsible for handling POST requests, which returns all the todo documents available in the `todos` collection. You save the todos you get from the database in a variable called `all_todos`, and then you edit the `render_template()` function call to pass the list of todo documents to the `index.html` template, which will be available in the template in a variable called `todos`.

If you refresh the index page, you might receive a message from the browser asking you to confirm form resubmission. If you accept, the todo item you previously submitted before handling POST requests will then be added to the database, because the code for handling forms now exists in the route.

Because the index page has no code to display todo items yet, the item you added will not be visible. If you allowed the browser to resubmit your form, you can see the newly added data by opening your mongo shell, and connecting to the `flask_db` database using the following command:

```
> use flask_db
```

Copy

Then use the `find()` function to get all the todo items in the database:

```
> db.todos.find()
```

Copy

If any data was resubmitted, you'll see it here in your output.

Next, open the `index.html` template to display the contents of the `todos` list you passed to it:

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

Copy

Edit the file by adding an `<hr>` break and a [Jinja for loop](#) after the form, so that the file looks as follows:

```
flask_app/templates/index.html
```

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

Copy

CONTENTS

Introduction

Prerequisites

Step 1 — Setting Up PyMongo and Flask

Step 2 — Connecting to the MongoDB Server and Creating a Collection

Step 3 — Creating a Web Page for Adding and Displaying Todos

Step 4 — Deleting Todos

Conclusion

RELATED

How To Install MongoDB on Ubuntu 12.04

[Tutorial](#)

How To Securely Configure a Production MongoDB Server

[Tutorial](#)

```

<meta charset="UTF-8">
<title>FlaskApp</title>
<style>
    .todo {
        padding: 20px;
        margin: 10px;
        background-color: #eee;
    }
</style>
</head>
<body>
    <h1>FlaskTODO</h1>
    <hr>
    <div class="content">
    <form method="post">
        <p>
            <b><label for="content">Todo content</label></b>
        </p>
        <p>
            <input type="text" name="content"
                placeholder="Todo Content"></input>
        </p>

        <p>
            <b><label for="degree">Degree</label></b>
        </p>
        <p>
            <input id="degree-0" name="degree" required type="radio" value="Important">
            <label for="degree-0">Important</label>
        </p>
        <p>
            <input id="degree-1" name="degree" required type="radio" value="Unimportant">
            <label for="degree-1">Unimportant</label>
        </p>
        <button type="submit">Submit</button>
    </form>
    <hr>
    {% for todo in todos %}
        <div class="todo">
            <p>{{ todo['content'] }} <i>{{ todo['degree'] }}</i></p>
        </div>
    {% endfor %}

</div>
</body>
</html>

```

Save and close the file.

In this file, you add an `<hr>` tag to separate the web form and the list of todos.

You use a `for` loop in the line `{% for todo in todos %}` to go through each todo item in the `todos` list. You display the todo content and the degree of importance inside a `<p>` tag.

Now refresh your index page, fill in the web form, and submit it. You'll see the todo you added below the form. Next, you'll add a button to allow users to delete existing todos.

Step 4 – Deleting Todos

In this step, you'll add a route that allows users to delete todos using a button.

First, you'll add a new `/id/delete` route that accepts POST requests. Your new `delete()` view function will receive the ID of the todo to be deleted from the URL, then use that ID to delete it.

To delete a todo, you get its ID as a string, and you must convert it to an [ObjectId](#) before passing it to the collection's delete method. So you need to import the `ObjectId()` class from the [bson module](#), which handles BSON (Binary JSON) encoding and decoding.

Open `app.py` for editing:

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

Copy

First, add the following import at the top of the file:

```
flask_app/app.py
```

```
from bson.objectid import ObjectId

# ...
```

Copy

This is the `ObjectId()` class you'll use to convert string IDs to `ObjectId` objects.

Then add the following route at the end:

```
flask_app/app.py
```

```
# ...

@app.post('/<id>/delete/')
def delete(id):
    todos.delete_one({'_id': ObjectId(id)})
    return redirect(url_for('index'))
```

Copy

Save and close the file.

Here, instead of using the usual `app.route` decorator, you use the `app.post` decorator introduced in [Flask version 2.0.0](#), which added shortcuts for common HTTP methods. For example, `@app.post("/login")` is a shortcut for `@app.route("/login", methods=["POST"])`. This means that this view function only accepts POST requests, and navigating to the `/ID/delete` route on your browser will return a 405 Method Not Allowed error, because web browsers default to GET requests. To delete a todo, the user clicks on a button that sends a POST request to this route.

The function receives the ID of the todo document to be deleted. You pass this ID to the `delete_one()` method on the `todos` collection, and you convert the string ID you receive to an `ObjectId` using the `ObjectId()` class you imported earlier.

After deleting the todo document, you redirect the user to the index page.

Next, edit the `index.html` template to add a **Delete Todo** button:

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

Copy

Edit the `for` loop by adding a new `<form>` tag:

```
flask_app/templates/index.html
```

```
{% for todo in todos %}
<div class="todo">
  <p>{{ todo['content'] }} <i>({{ todo['degree'] }})</i></p>
  <form method="POST" action="{{ url_for('delete', id=todo['_id']) }}" >
    <input type="submit" value="Delete Todo "
      onclick="return confirm('Are you sure you want to delete this entry?')>
  </form>
</div>
{% endfor %}
```

Copy

Save and close the file.

Here, you have a web form that submits a POST request to the `delete()` view function. You pass `todo['_id']` to specify the todo that will be deleted. You use the `confirm()` method available in web browsers to display a confirmation message before submitting the request.

Now refresh your index page and you'll see a **Delete Todo** button below each todo item. Click on it, and confirm the deletion. You'll be redirected to the index page, and the todo will no longer be there.

You now have a way of deleting unwanted todos from your mongoDB database in your Flask application.

To confirm the deletion, open your mongo shell and use the `find()` function:

```
> db.todos.find()
```

[Copy](#)

You should see that the items you've deleted are no longer in your `todos` collection.

Conclusion

You built a small Flask web application for managing todos that communicates with a MongoDB database. You learned how to connect to a MongoDB database server, create collections that store a group of documents, insert data to a collection, and retrieve and delete data from a collection.

If you would like to read more about Flask, check out the other tutorials in the [How To Create Web Sites with Flask](#) series.

For more on MongoDB, see our [How To Manage Data with MongoDB](#) tutorial series.

Next in series: [How to Use Flask-SQLAlchemy to Interact with Databases in a Flask Application](#) →

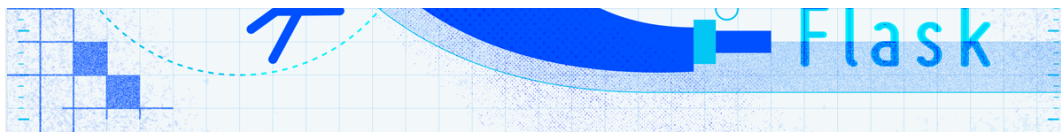
Want to learn more? Join the DigitalOcean Community!

Join our DigitalOcean community of over a million developers for free! Get help and share knowledge in our Questions & Answers section, find tutorials and tools that will help you grow as a developer and scale your project or business, and subscribe to topics of interest.

[Sign up](#) →

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](#) [Development](#) [Flask](#) [MongoDB](#) [Python](#) [Python Frameworks](#)

Browse Series: 10 articles

[1/10 How To Create Your First Web Application Using Flask and Python 3](#)[2/10 How To Use Templates in a Flask Application](#)[3/10 How To Handle Errors in a Flask Application](#)[Expand to view all](#)

About the authors



[Abdelhadi Dyouri](#) Author
Developer and author at DigitalOcean.



[Brian MacDonald](#) Editor
Senior Acquisitions Editor

Editor at DigitalOcean, former book editor at Pragmatic, O'Reilly, and others. Occasional conference speaker. Highly nerdy.

Still looking for an answer?



[Ask a question](#)[Search for more help](#)

Was this helpful?

[Yes](#)[No](#)

Comments

Leave a comment

B I U ↶ ↷ ⚡ H₁ H₂ H₃ ≡ 1. „, ⓘ ☐ <>  

Leave a comment...

Login to Comment

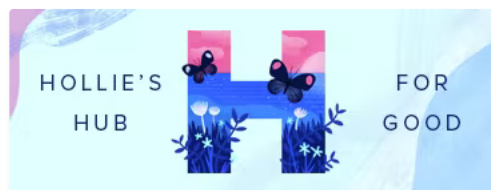


This work is licensed under a Creative Commons Attribution-NonCommercial- ShareAlike 4.0 International License.



GET OUR BIWEEKLY NEWSLETTER

Sign up for Infrastructure as a Newsletter.



HOLLIE'S HUB FOR GOOD

Working on improving health and education, reducing inequality, and spurring economic growth? We'd like to help.



BECOME A CONTRIBUTOR

You get paid; we donate to tech nonprofits.

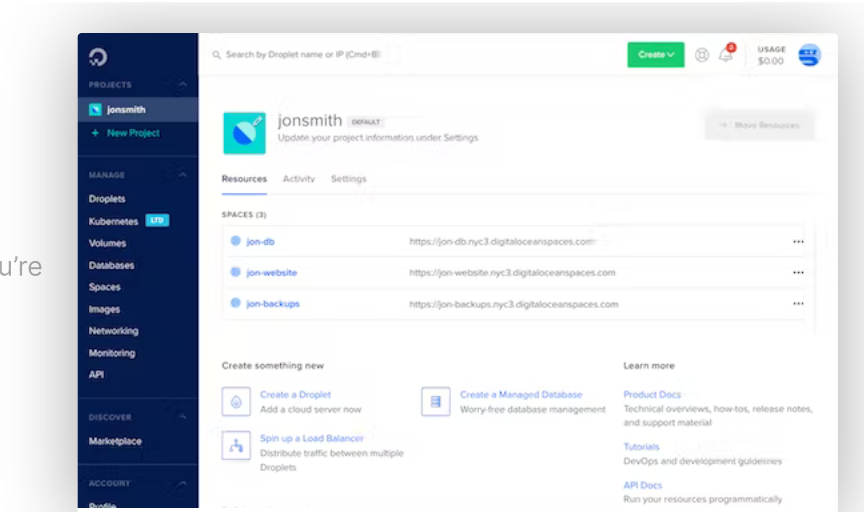
Featured on [Community](#) [Kubernetes Course](#) [Learn Python 3](#) [Machine Learning in Python](#) [Getting started with Go](#) [Intro to Kubernetes](#)

DigitalOcean Products [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](#)



Company

- About
- Leadership
- Blog
- Careers
- Customers
- Partners
- Referral Program
- Press
- Legal
- Trust Platform
- Investor Relations
- DO Impact

Products

- Products Overview
- Droplets
- Kubernetes
- App Platform
- Functions
- Managed Databases
- Spaces
- Marketplace
- Load Balancers
- Block Storage
- Tools & Integrations
- API
- Pricing
- Documentation
- Release Notes

Community

- Tutorials
- Meetups
- Q&A
- CSS-Tricks
- Write for DONations
- Droplets for Demos
- Hatch Startup Program
- Shop Swag
- Research Program
- Currents Research
- Open Source
- Code of Conduct
- Newsletter Signup

Solutions

- Web & Mobile Apps
- Website Hosting
- Game Development
- Streaming
- VPN
- Startups
- SaaS Solutions
- Agency & Web Dev Shops
- Managed Cloud Hosting
- Providers
- Big Data
- Business Solutions
- Cloud Hosting for Blockchain

Contact

- Support
- Sales
- Report Abuse
- System Status
- Share your ideas

© 2022 DigitalOcean, LLC. All rights reserved.

