**How to use Python and Flask to build a web app — an in-depth tutorial**



Locust walk at the University of Pennsylvania. The home of our clubs fair which inspired the example App I build in this article (credit to the [Daily Pennsylvanian](https://s3.amazonaws.com/media.dpn/51491_locustf.jpg))

Python is an incredibly versatile language. It’s considered to be a staple of modern development. It’s used for the simplest of scripts to incredibly complex machine learning and neural network training algorithms.

But perhaps the less-known usage of Python is its use as a web server. Overshadowed by more popular frameworks like as Node/Express and Ruby on Rails, Python is often overlooked as a web server choice for most developers.

Having a backend written in Python is incredibly useful for several reasons, among which are:

* It’s incredibly easy to step up from learning Python as a regular scripting language to using it to make a backend.
* It’s best to use if you plan on serving parts of your application that are already written in Python (For example - submitting a form, evaluating input via a Tensorflow model, and returning the output to a use)
* It has an incredibly diverse ecosystem of packages and tools to help you with development, not to mention a great community of developers (since the language has been around so long)

The purpose of this article is to demonstrate how Python can be used to create a full stack web application. In this tutorial, I will be using Flask, a Python “microframework” to developing a web application.

I would be remiss not to mention that there are other more popular Python frameworks out there such as Django, but Flask is incredibly useful for the budding developer since it is bare bones and requires developers to create/utilize the components they need within the App based on their requirement (rather than calling some command line tool that generates 20 files automatically… look in’ at you Ruby on Rails). Of course, I won’t be going through how to start a Web App completely from scratch, rather I’ll give you an intro to Flask and then move onto how you can use a project called flask-base to get upto speed quickly in the future.

**Intro to Flask**

Flask is a microframework (read as: It doesn’t come with much) for web development in Python. Before we do a deep(ish) dive, let’s cover some basic concepts of backend development.



Soure: flask.pocoo.org

Routes

Let’s imagine you’re visiting apple.com and want to go to the Mac section at apple.com/mac/. How do Apple’s servers know to serve you the specific page that shows the details about Mac devices. It is most likely because they have a web app running on a server that knows when someone looks up apple.com and goes to the /mac/ section of the website, handle that request and send some pages back. The logic behind figuring out what to do when someone goes to /mac/ is done by a route.

So when I visit apple.com (implied apple.com/), the / route handles what is shown. If I go to apple.com/purchase, there is a /purchase route. If I go to apple.com/purchase/1 where 1 is some item identifier, there most likely is a generic route handler /purchase/<int:item-id> that handles that request. Routes can handle both GET and POST requests as well.

**Basic App**

So how do we make a basic Flask app that has routes? Well, let’s take a look at the docs. Create a Python file called hello.py that contains the following.

from flask import Flask

app = Flask(\_\_name\_\_)

@app.route("/")

def hello():

return "Hello World!"

if \_\_name\_\_ == '\_\_main\_\_':

app.run()

Let’s break down what’s happening here.

1. We import our Flask dependency
2. We create an instance of a Flask App. The argument passed into the Flask instantiator (\_\_name\_\_) evaluates to a string that "names" the Flask App. When run from the command line, \_\_name\_\_ == "\_\_main\_\_". You can set the first argument to whatever you want.
3. We set up a route [/](http://blah.com/) on our App that executes the [hello(](http://blah.com/)) function immediately below it when that route is visited. Note that the function must return a string or a rendered template.

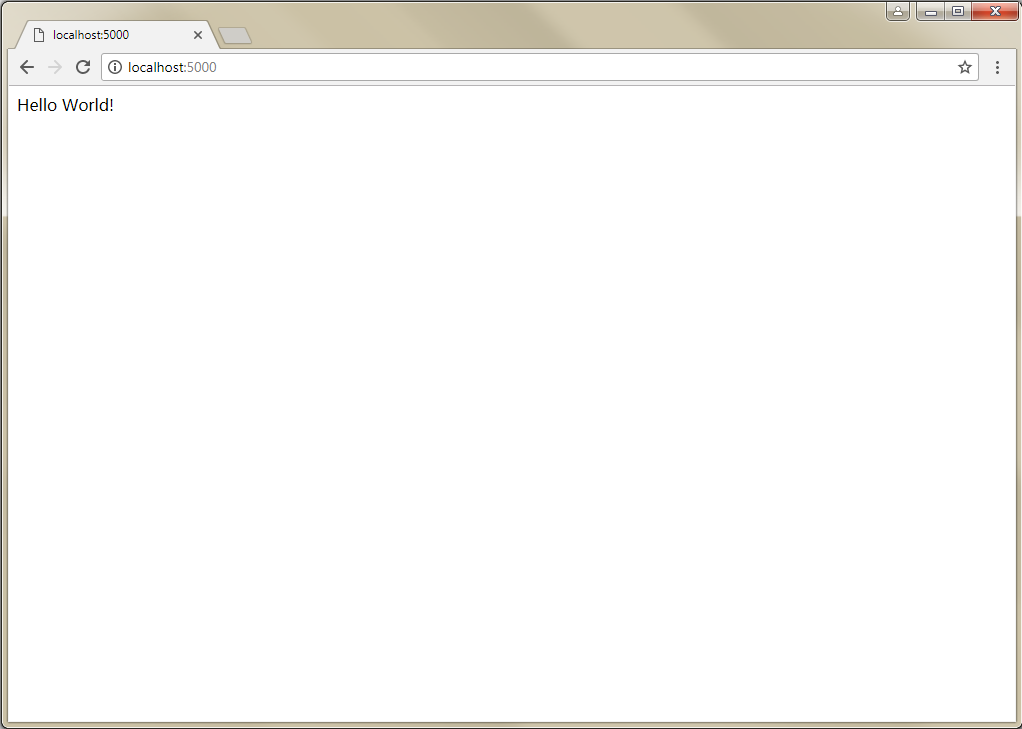
On the command line, let’s set up something called a virtual environment (it will help us isolate our development environment and package installations from the rest of our system).

1. If you haven’t done so already, install pip via easy\_install pip (you may need to run sudo in front of this if you are on a Mac.
2. Run pip install virtualenv to install virtualenv using pip
3. In the directory of your App, create your virtual environment by running virtualenv venv (this creates a virtual environment in a folder called venv of the current directory).
4. Run source venv/bin/activate to activate the virtual environment. This is specifically required to install packages into it. You can deactivate the virtual environment by running deactivate from the command line. Pretty simple.

Now that our virtual environment is installed and activated, let’s install Flask. It’s really simple, just run pip install Flask. We can then run the example from earlier by writing the following in our command line.

FLASK\_APP=hello.py flask run

You should see something like \* Running on http://localhost:5000/ in your terminal. And if you visit that link in your browser, you'll see a page with just Hello World!.



**Example App: Penn Club Review**

*Note: The code for this project can be found at* [*this repository on GitHub*](https://github.com/abhisuri97/penn-club-ratings)*.*

Now let’s figure out some project to create in order to demonstrate the full capabilities of Flask. One recent project I came up with is a club rating app called “PennClubReview”.

**PennClubReview**

I’m currently attending the University of Pennsylvania. One of the most common problems that freshmen on campus face is choosing which clubs to join on campus. This process is further complicated by the fact that some clubs are incredibly competitive to get into, have multiple interview rounds, and require a large time commitment. Often, none of these aspects of clubs are discussed during club information sessions.



UPenn club fair

So, in order to combat this issue, we can create an app where

* An administrator can set survey questions for users to answer about clubs.
* Users can view average ratings for each survey question for each club
* Users can view individual responses for clubs. If a user chooses to submit another review, their previous response is overwritten.
* Users can suggest clubs for administrators to edit/approve to show in public (administrators should be notified via email when this happens)
* A user or admin needs to be able to edit their own account information.
* An administrator should have the ability to add/remove users, survey questions, club categories and clubs from the system.

**Breaking down App components**

In order to develop this app, we’ll need to have some more components in addition to Flask, such as a backing database, a login management system, some way to organize routes, and handle emailing. We could code this from scratch. But, there is already an amazing boilerplate that can give you a great starting place.

**Enter Flask-Base**



Flask-base is a project that my friends and I developed as part of a student run nonprofit called [Hack4Impact](https://hack4impact.org/). We work with nonprofits over a semester to develop technical projects that help them accomplish their mission.

While working on so many projects, we found out that we would often be repeating the same code across all of our applications. So we decided to create a single code base containing the most common parts that any App we made would need. This code base would include:

* User authentication scheme
* Account management
* Blueprints (to handle routes)
* Backing database
* Emailing (with a redis queue)

It recently became fairly popular, garnering 1200+ GitHub stars over the course of a few months. This codebase is perfect for what we are trying to set up. You can find the GitHub repo containing the code for Flask base [here](https://github.com/hack4impact/flask-base).

**App dev setup**

First let’s clone flask-base. Follow the instructions on the README.md page. In a nutshell run the following.

git clone https://github.com/hack4impact/flask-base.git

cd flask-base

virtualenv venv

source venv/bin/activate

pip install -r requirements.txt

python manage.py recreate\_db

python manage.py setup\_dev

OK. I’ll elaborate on what we’ve done here.

* Clone the repository from GitHub (i.e. download it) and then go into its directory.
* Create a new virtual environment and activate it.
* Read the package dependencies in the requirements.txt file and install all of them via pip.
* Instantiate the database (recreate it) and also insert an administrator rule (via setup\_dev).

Additionally, let’s create a running database migration. This will keep track of changes in our database models without needing to recreate our database (i.e. remove all the information and then rebuild the database from scratch). Migrations allow us to preserve information. We can do this via the command below.

python manage.py db init && python manage.py db migrate && python manage.py db upgrade

To run the App, run honcho start -f Local (you'll need to install Honcho if you haven't already). If you have any issues, chances are they have been addressed in the README of flask-base already. Now you can visit localhost:5000 and pull up a running flask-base application.

To log into the App as an administrator, go to the login link and type in for the username [flask-base-admin@example.com](mailto:flask-base-admin@example.com) with a password password. You can then invite new users into the application from the administrator screen. Note that before you do so, you'll need to create a config.env file that contains the following two variables:

MAIL\_PASSWORD=someSendGridPassword  
MAIL\_USERNAME=someSendGridUsername

Upon creation of a user account, the account remains unconfirmed until the new invited user clicks a link sent to their email. Additionally, a user can register for the App and will go through a similar authentication flow with regards to confirmation.

Look through the flask-base documentation to get a better sense of some of the capabilities of flask-base out of the box. For now, we’re going to move on to how we can use it to make our App.

**Databases!**



All our database logic is wrapped by the SQLAlchemy ORM so we don’t have to make very verbose database statements to run queries or add/delete records. All the database *models* (think of them as classes) are contained within the app/models folder. Let's think of some models that are needed for the application itself.

So we need to have a Club model that contains the name of the club (Datatype: String), a club description (Datatype: Text) and a variable is\_confirmed (Datatype: Boolean) to keep track of whether a club that is suggested has been approved by an administrator to be shown. Additionally, we want some way to refer to the categories of a club, and another way to refer to the question answers that belong to a club.

Let’s think about how Clubs and Club Categories should relate to each other. We can think of it as follows. A club has many categories (e.g. a club can be a Social Impact and Tech club) and a club category can belong to many clubs (e.g. there can be many Tech clubs on campus). The only attribute this ClubCategory has a category\_name (Datatype: String).

We can create this relationship (a many to many relationship), via an association table.

**Club and Club Categories (Many to Many)**

Now how do we encode that logic into flask-base? First, create a file called club.py in app/models. First let's create the Club and ClubCategory models.

from .. import db

# we are going to put something here later

class Club(db.Model):

\_\_tablename\_\_ = 'clubs'

id = db.Column(db.Integer, primary\_key=True)

name = db.Column(db.String(1000))

description = db.Column(db.Text)

is\_confirmed = db.Column(db.Boolean)

class ClubCategory(db.Model):

\_\_tablename\_\_ = 'club\_categories'

id = db.Column(db.Integer, primary\_key=True)

category\_name = db.Column(db.String(1000))

So now we have two models, but they aren’t connected to each other. Each of them have individual attributes, but neither can be explicitly connected to each other. We make the connection via an association as I mentioned earlier. After the db import, add the following lines.

club\_category\_assoc = db.Table('club\_category\_association', db.Model.metadata,

db.Column('club\_id', db.Integer,db.ForeignKey('clubs.id')),

db.Column('club\_category\_id', db.Integer,db.ForeignKey('club\_categories.id')))

What this does is create a new association table (an intermediary between the Club and ClubCategory model). There are two columns in this table club\_id and club\_category\_id which refer to the respective id’s of their respective models (note that the id attribute is a Primary Key within each model, i.e. the thing that is unique for each record). But within the association table, we refer to these Primary Keys as Foreign Keys (because they are refering to other tables). Additionally, we need to add a line to the Club model at the bottom.

categories = db.relationship('ClubCategory', secondary=club\_category\_assoc, backref='clubs')

And this actually creates the bidirectional relationship between the Club and ClubCategory models and sets up a relationship between Club and ClubCategory using the club\_category\_assoc association table. The backref tells the ClubCategory model how to refer to the Club models. So, with a given club club, you can run club.categories to get an array of category object backs. With a given ClubCategory called category, you can get all the clubs in that category by doing category.clubs.

You can see this in action by doing the following:

In app/models/\_\_init\_\_.py add the line

from .club import \* # noqa

And then run python manage.py shell. Run the following commands to interact with your database models (note that >>> indicates an input you put in).

Importing environment from .env file

SECRET KEY ENV VAR NOT SET! SHOULD NOT SEE IN PRODUCTION

THIS APP IS IN DEBUG MODE. YOU SHOULD NOT SEE THIS IN PRODUCTION.

>>> from app import models, db

>>> club = models.Club(name="hello world club")

>>> category1 = models.ClubCategory(category\_name="test1")

>>> category2 = models.ClubCategory(category\_name="test2")

>>> club2 = models.Club(name="hello world club 2")

>>> club.categories.append(category1)

>>> club.categories.append(category2)

>>> category1.clubs.append(club2)

>>> [x.category\_name for x in club.categories]

['test1', 'test2']

>>> [x.category\_name for x in club2.categories]

['test1']

>>> [x.name for x in category1.clubs]

['hello world club', 'hello world club 2']

>>> [x.name for x in category2.clubs]

['hello world club']

**Questions and Answers (Many to One)**

Great! We now have a working Club and ClubCategory model. Now let’s move onto the Question and Answer models. For a question, we need to keep track of the content of the question which will be a **String** containing the text of the question itself. We will also include a **max\_rating** attribute that will contain the maximum rating an individual can give for the question. For example, if the question content is "Rate the community of the club 10 is the best", we could set **max\_rating** to be 10. Additionally, we'll keep track of a Boolean free\_response to determine whether we will allow people to include an optional extra response that is long form. Lastly, we will need to have a relation to the **Answer** model. This will be a one to many relation because a question can have multiple answers but an answer can only have one question.

The **Answer** model will have the following attributes:

* an **answer** attribute corresponding the the free text response of an answer (if the question allows a free text response)
* answer：该属性与答案的自由文本响应有关（如果问题允许自由文本响应）
* rating：评分范围从1到问题的最大评分
* user\_id：评论用户的ID（一个用户可以有很多问题，但是一个问题只有一个用户）
* question\_id：指的是answer所属的question
* club\_id：指的是answer所属的club。

Let’s create a file **question.py**

from .. import db

class Question(db.Model):

\_\_tablename\_\_ = 'questions'

id = db.Column(db.Integer, primary\_key=True)

content = db.Column(db.String(1000))

max\_rating = db.Column(db.Integer)

free\_response = db.Column(db.Boolean, default=False)

answers = db.relationship("Answer", backref="question")

Most of the stuff in here is fairly straightforward except for the last line. The last line connects the **Question** and **Answer** models. It says to set up a relationship with the **Answer** model which can refer to the **Question** model via the keyword **question**. Given an answer **a**, you can get the question via **a.question** and given a question **q**, you can get the answer associated with it via **q.answers**. Let's now set up the **Answer** model. Create a new file called answer.py in the models folder and paste in the following.

from .. import db

class Answer(db.Model):

\_\_tablename\_\_ = 'answers'

id = db.Column(db.Integer, primary\_key=True)

answer = db.Column(db.Text)

rating = db.Column(db.Integer)

user\_id = db.Column(db.Integer, db.ForeignKey('users.id'))

question\_id = db.Column(db.Integer, db.ForeignKey('questions.id'))

club\_id = db.Column(db.Integer, db.ForeignKey('clubs.id'))

@staticmethod

def newAnswer(answer, rating, user\_id, question\_id, club\_id):

a = Answer.query.filter\_by(

user\_id=user\_id, club\_id=club\_id, question\_id=question\_id).first()

if a is not None:

db.session.delete(a)

db.session.commit()

a\_new = Answer(

answer=answer,

rating=rating,

user\_id=user\_id,

question\_id=question\_id,

club\_id=club\_id)

db.session.add(a\_new)

db.session.commit()

So this file is much longer, but recall that there are many things an answer is related to. Let’s start at the beginning, note that **question\_id** refers to the **Question** model via the foreign key **questions\_id** (the **id** column of the **questions** table (which contains records of instances of the **Question** model).

Note that we also have a **user\_id** column that refers to a user. Let's go into **user.py** within the **app/models** folder and add the following line after the **role\_id** declaration.

answers = db.relationship('Answer', backref='user')

This statement uses very similar syntax to that of the Question model.

Also note that there is a **club\_id** attribute that refers to the club the answer is associated with. Edit the **club.py** file to include the following line as the last attribute of the **Club** model.

answers = db.relationship('Answer', backref='club')

Finally, add these two lines to **\_\_init\_\_.py** in **app/models**

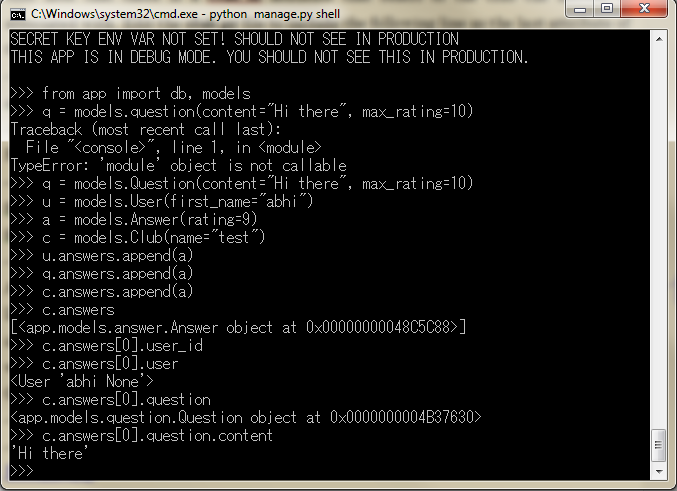
from .question import \* # noqa

from .answer import \* # noqa

And now we should be able to play around with our databases as follows.

Lastly, let’s address the **newAnswer** method. This method is used to insert new answers into the database while making sure that if a user has already answered that question, we delete it and insert the new response.

Once again, we can run **python manage.py shell**



There, we are now done with the models :)

至此，我们完成了相关模块。

**Views**

Now the database stuff is out of way, let’s create the way for users to interact with the application itself. First let’s set up some blueprints.

现在，数据库内容还是独立的，让我们创建方法允许用户和应用程序本身进行交互。首先，让我们建一些模板（蓝图）。

**Blueprints**

Blueprints are a great way to organize you flask application. It allows you to mount all routes that are associated with each other in a single file. For example, for all actions associated with an account, such as account management, user password reset, forgot password, etc. would be included in the **account** blueprint.

Each blueprint has a folder associated with it under **app**. For example, there is an **account**/ folder and a folder under **templates** containing the actual html templates that will be rendered to the user.

Let’s add some blueprints. Before the **return app** line of **app/\_\_init\_\_.py** add the following.

from .club import club as club\_blueprint

app.register\_blueprint(club\_blueprint, url\_prefix='/club')

from .question import question as question\_blueprint

app.register\_blueprint(question\_blueprint, url\_prefix='/question')

from .category import category as category\_blueprint

app.register\_blueprint(category\_blueprint, url\_prefix='/category')

These calls create blueprints mounted at the url prefixes **/club**, **/question**, and **/category** respectively. Let's create the folders **club**, **question**, and **category** for each of the blueprints. Within each of the folders create the files **\_\_init\_\_.py**, **forms.py**, and **views.py**.

**Club Forms and Views**

I’ll walk through how to set up the views/templates for **club** blueprint. The other views are fairly easy to understand from the code.

So within the club view, we want to have a few different things to show

1. If you are an administrator, you should be able to create a club and give it a name, description, and categories.
2. If you are an administrator, you should be able to view all the clubs, including ones that aren’t confirmed.
3. If you are an administator or user, you should be able to view an individual club’s information.
4. If you are an administator, you should be able to edit a club’s information and delete a club.

Let’s first create a couple of forms within **forms.py** that we will then pass to our views, specifically the view that handles create a new club and the one that edits club information.

In **forms.py** for **club** add the following lines:

from flask\_wtf import Form

from wtforms.ext.sqlalchemy.fields import QuerySelectMultipleField

from wtforms.fields import StringField, SubmitField, TextAreaField, SelectField

from wtforms.validators import InputRequired

from .. import db

from ..models import ClubCategory

class NewClubForm(Form):

name = StringField('Please input the name of the club')

desc = TextAreaField('Please input the description of the club')

categories = QuerySelectMultipleField(

'Add categories for club',

validators=[InputRequired()],

get\_label='category\_name',

query\_factory=

lambda: db.session.query(ClubCategory).order\_by('category\_name'))

submit = SubmitField('Create Club')

class EditClubForm(NewClubForm):

is\_confirmed = SelectField(

'Please indicate whether this club entry should be shown',

choices=[('True', 'Yes'), ('False', 'No')])

submit = SubmitField('Edit Club')

Flask-base uses **wtforms** to create forms. **wtforms** allows us to create forms in an object oriented manner where each form is a class.

So we create two forms, one called **NewClubForm** that extends the base **wtforms** **Form** class, and has 3 fields – **name** (Datatype: Text), **desc**(Datatype: Text) containing the description of the club, and **categories** (a multiple select dropdown). With the categories field, we query the **ClubCategory** model with a **Lambda** function (which is basically an anonymous function) for the category names and populate the category select field options with the results from that query.

Lastly, we have a **submit** field, so the submit button can be rendered.

Next, we have an **EditClubForm** which extends the **NewClubForm** field set by adding a new field called **is\_confirmed**. Recall that **is\_confirmed** in our **Club** model determines whether the given club instance can be shown or not shown to the public. We will be adding the function for a club to be suggested by users, and by default, suggested clubs are hidden until approved by an admin. We also overwrite the **submit** field to display the text "Edit Club".

In **views.py** under **club/**, we create a few routes.

* **/new-club** (GET, POST) LOGIN PROTECTED: The renders and accepts data from form for creating a new club.
* **/clubs** (GET) ADMIN PROTECTED: Renders all the clubs
* **/<int:club\_id>/(:info)** (GET) LOGIN PROTECTED: Will render out info for a given club instance with **id = club\_id** and can access the route at **/club/1** or **/club/1/info**.
* **/<int:club\_id>/change-club-details** (GET, POST) ADMIN PROTECTED: Render and accept data from form for editing club information.
* **/<int:club\_id>/delete** (GET) ADMIN PROTECTED: Render page to delete club
* **/<int:club\_id>/\_delete** (GET) ADMIN PROTECTED: Delete club with club id.

For the first route **/new-club**, we want to also allow regular users to create a new club, which is why we only login protect it. Let's see how we can make a route for this.

@club.route('/new-club', methods=['GET', 'POST'])

@login\_required

def new\_club():

"""Create a new club."""

form = NewClubForm()

if form.validate\_on\_submit():

club = Club(

name=form.name.data,

description=form.desc.data,

is\_confirmed=current\_user.is\_admin(),

categories=form.categories.data)

db.session.add(club)

db.session.commit()

link = url\_for(

'club.change\_club\_details', club\_id=club.id, \_external=True)

if (current\_user.is\_admin() == False):

for r in Role.query.filter\_by(name='Administrator').all():

for a in r.users:

get\_queue().enqueue(

send\_email,

recipient=a.email,

subject='A new club was suggested by {}'.format(

current\_user.first\_name),

template='club/email/suggested\_club',

club=club,

link=link)

action = 'created' if current\_user.is\_admin() else 'suggested'

flash('Club {} successfully {}'.format(club.name, action),

'form-success')

return render\_template('club/new\_club.html', form=form)

[原文链接地址](https://medium.freecodecamp.org/how-to-use-python-and-flask-to-build-a-web-app-an-in-depth-tutorial-437dbfe9f1c6)