sakyum Documentation

Release latest

Aug 18, 2023

CONTENTS

WARNING This project has been totally renamed to flask-unity , official repository is https://github.com/usmanmusa1920/flask-unity

An extension of flask web framework that erase the complexity of structuring flask project blueprint, packages, and other annoying stuffs.

The main reason behind the development of *sakyum* is to combine flask and it extensions in one place to make it ease when developing an application without the headache (worrying) of knowing the tricks on how to connect those extensions with flask, or import something from somewhere to avoid some errors such as circular import and other unexpected errors. Also structuring flask application is a problem at some cases to some people, *sakyum* take care of all these so that you only focus on writing your application views the way you want.

Sakyum depends on (come with) the following flask popular and useful extensions, these include: flask-admin for building an admin interface on top of an existing data model (where you can manage your models in the admin page), flask-bcrypt it provides bcrypt hashing utilities for your application, flask-login it provides user session management for Flask. It handles the common tasks of logging in, logging out, and remembering your users' sessions over extended periods of time, flask-sqlalchemy It simplifies using SQLAlchemy with Flask by setting up common objects and patterns for using those objects, such as a session tied to each web request, models, and engines, flask-wtf Simple integration of Flask and WTForms, including CSRF, file upload, and reCAPTCHA. And possibly some other extensions / libraries.

When using sakyum, don't be confuse with the the concept of **project** and **app**.

Project is the entire folder that contain your flask application, when you create project with the command:

```
python -c "from sakyum import project; project('schoolsite')"
```

It will create a parent directory with the name *schoolsite* also inside the *schoolsite* directory there is a directory with thesame name of the parent directory *schoolsite* this directory is the one that most of configurations, registering and other thing that are going to be implemented inside it.

Within that parent directory *schoolsite* it also generate a file called *thunder.py* this is the file that you will be running along side with some positional arguments and flags. Also it will generate a directory called *auth* this directory contains admin system utilities. Lastly it will generate *templates* and *static* directory for your site pages and it styles respectively. At a glance, it will create *schoolsite, thunder.py, templates, static, auth* all in *schoolsite*.

App (application) is like to say a blueprint which greatly simplify how large applications work and provide a central means for Flask extensions to register operations on applications. Read blueprint about flask.

Don't worry if you didn't get the concept of *project* and *app*, surely you will get it if we dive deep by the help of the schoolsite project.

CHAPTER

ONE

TABLE OF CONTENT

1.1 Quick start

First we recomend you to create a virtual environment to avoid conflict (upgrade/downgrade of some of your system libraries) when installing sakyum, this is just a recomendation, it still work even if you install it without using virtual environment

Install and update the latest release from pypi. Basically the library was uploaded using **sdist** (Source Distribution) and **bdist_wheel** (Built Distribution), this software (library) as from **v0.0.9** it is compatible and also tested with **windows OS** and others as well, such as **linux**, **macOS** and possibly some others too!.

You will notice we use **–upgrade** in the installation command, this will make sure it install the latest release from pypi (in case you have a version which is not the latest), you can still ommit the *–upgrade* and use the version you want then wait for the installation to finish.:

```
pip install --upgrade sakyum
```

This **quick start** will walk you through creating project called **schoolsite** and a basic application called **exam** in the project. User will be able to register, login/logout, create exam questions/choices, and edit or delete their own question/choices. All using *sakyum*, you will be able to clone it on github. it is located inside example directory of the base repository.

1.1.1 Create flask project using sakyum

Now after the installation, let create a project called **schoolsite** to do so paste the following command on your termianl:

```
python -c "from sakyum import project; project('schoolsite')"
```

or create a file and paste the below codes which is equivalent of the above, and then run the file

```
from sakyum import project

project("schoolsite")
```

Both the command you type on terminal or the code you paste in a file (after running the file) will create a project called **schoolsite** now cd into the **schoolsite** directory, if you do **ls** within the directory you just enter you will see a module called **thunder.py** and some directories (some in the form of package) **media**, **static**, **templates** and a directory with thesame name of your parent directory which is **schoolsite**.

Tree structure of the project using **tree .** command look like:

```
.
├── media
│   └── default_img.png
├── schoolsite
│   ├── config.py
│   ├── __init__.py
│   ├── routes.py
│   └── secret.py
├── static
│   └── schoolsite
│       ├── index.js
│       ├── media
│       └── style.css
├── templates
│   ├── admin
│   │   └── index.html
│   └── schoolsite
│       └── index.html
└── thunder.py

8 directories, 10 files
```

Boot up the flask server by running the below command:

```
python thunder.py boot
```

Now visit the local url **http://127.0.0.1:5000** this will take you to the index page of your project with some links in the page.

1.1.2 Create flask project app using sakyum

Since we create a project, let create an app within the project. To start an app within the project (**schoolsite**) shutdown the flask development server by pressing (CTRL+C). If you do **ls** in that same directory you will see it create a **default.db** file (an sqlite file) which is our default database. Now run the following command in other to create your app, by giving the name you want your app to be, in our case we will call our app **exam**:

```
python thunder.py create_app -a exam

# or

python thunder.py create_app --app exam
```

this will create an app (a new package called **exam**) within the project (**schoolsite**), the **-a** flag is equivalent to **–app** which is a flag for the app name in this example it is called **exam**

Now the **tree .** structure of the project after creating **exam** app look like:

```
.
├── default.db
├── exam
│   ├── admin.py
│   ├── forms.py
│   ├── __init__.py
```

(continues on next page)

(continued from previous page)

```
    ├── models.py
    └── views.py
├── media
    └── default_img.png
├── schoolsite
    ├── config.py
    ├── __init__.py
    ├── routes.py
    └── secret.py
├── static
    ├── exam
        ├── index.js
        ├── media
        └── style.css
    └── schoolsite
        ├── index.js
        ├── media
        └── style.css
├── templates
    ├── admin
        └── index.html
    ├── exam
        └── index.html
    └── schoolsite
        └── index.html
└── thunder.py

12 directories, 19 files
```

You notice it create a package name with thesame name of the app (**exam**) with some files in it, also a directory named **exam** inside **templates** and **static** folder with default html page together with css and js files (in static folder)

1.1.3 Register an app

Once the app is created it is time to register the app, to do so open a file **schoolsite/routes.py** and import your **exam** app blueprint which is in (**exam/views.py**), default name given to an app blueprint, is the app name so our **exam** app blueprint name is **exam**, after importing it, append (register) the app blueprint in a list called **reg_blueprints** in that same file of **schoolsite/routes.py**

```
**WARNING** don't ommit the registered blueprint you see in the `reg_blueprints` list
**(blueprint.default, blueprint.errors, blueprint.auth, base)** blueprints just append
your app blueprint
```

importing blueprint

```
from exam.views import exam
```

after that, append it in the list **reg_blueprints** provided in the **routes.py** file by

registering blueprint

```
reg_blueprints = [
  blueprint.default,
```

(continues on next page)

(continued from previous page)

```
  blueprint.errors,
  blueprint.auth,
  base,
  exam,
]
```

once you register the app, boot up the flask webserver again by:

```
python thunder.py boot
```

This will bring the flask development server on port **5000** you can give it a different port by including a flag **-p** or **–port** flag which is for port number:

```
python thunder.py boot -p 7000

# or

python thunder.py boot --port 7000
```

The above command will bring the development serve on port **7000** visit the localhost url with the port number, it will show you your project **index page** (schoolsite). To get to the app *(exam)* default page, visit the url with your app name in our case:

http://127.0.0.1:7000/exam

this will take you to the app (exam) **index page**, and you can also vist the admin page with this url **http://127.0.0.1:7000/admin**

Also, you can give your desire ip address/host by using **-H** or **–host** flag, e.g:

```
python thunder.py boot -p 7000 -H 0.0.0.0

# or

python thunder.py boot --port 7000 --host 0.0.0.0
```

For development server, you can give a debug value to True by specifying **-d** flag or **–debug** e.g:

```
python thunder.py boot -p 7000 -d True

# or

python thunder.py boot --port 7000 --debug True
```

You can change your default profile picture by moving to http://127.0.0.1:5000/admin/change_profile_image/ and select your new picture from your file system.

With this, you can do many and many stuffs now! From here you are ready to keep write more views in the app *views.py* as well as in the project *routes.py* and do many stuffs just like the way you do if you use flask only.

Source code for this *quick start* is available at official github repository of the project.

1.2 Flags

Some useful flags that you can use along side, when your are running your application (project) along side with *thunder.py* file are as follows:

Flags associated with `create_app` positional argument:

Use *-a* or *–app* if you are about to create app in your project, that will capture the app name:

```
python thunder.py create_app -a blog

# or

python thunder.py create_app --app blog
```

Flags associated with `boot` positional argument:

Use *-p* or *–port* if you want to give your desire port number instead of the default one which is *5000* It is use only if you are about to bring up the server, after the positional argument of *boot*:

```
python thunder.py boot -p 7000

# or

python thunder.py boot --port 7000
```

Use *-H* or *–host* if you are to give a different host, in the case of deployment. Also, it is use if you are about to bring up the server, after the positional argument of *boot*:

```
python thunder.py boot -H 0.0.0.0

# or

python thunder.py boot --host 0.0.0.0
```

Use *-d* or *–debug* if you want your app in debug mode. That mean ifyou make change, you need not to shutdown the server and reload it again, it will do that automatically once you set it to *True* Also, it is use if you are about to bring up the server, after the positional argument of *boot*:

```
python thunder.py boot -d True

# or

python thunder.py boot --debug True
```

Flags associated with `create_user` positional argument:

Use *-u* or *–username* and then the username beside it, if you do not specify it, you will see a prompt saying *Enter username:*:

```
python thunder.py create_user -u network-engineer

# or

python thunder.py create_user --username network-engineer
```

Use *-e* or *–email* and then the email beside it, if you do not specify it, you will see a prompt saying *Enter email:*:

```
python thunder.py create_user -e network-engineer@datacenter.com

# or

python thunder.py create_user --email network-engineer@datacenter.com
```

Use *-p* or *–password* and then the password beside it, if you do not specify it, you will see a prompt saying *Enter password:*:

```
python thunder.py create_user -p my-secret-pass

# or

python thunder.py create_user --password my-secret-pass
```

1.3 Error pages

There are 12 default error pages that have been implemented in sakyum. These error pages include:

Error page of *400* for bad request

Error page of *401* for unauthorized

Error page of *403* for forbidden

Error page of *404* for not found

Error page of *406* for ot acceptable

Error page of *415* for unsupported media type

Error page of *429* for too many requests

Error page of *500* for internal server error

Error page of *501* for not implemented

Error page of *502* for bad gateway

Error page of *503* for service unavailable

Error page of *504* for gateway timeout

This tutorial will be a continuation of the quick start. As we see in the *quick start* we create a project called **schoolsite** and an app inside the project called **exam**. Taking from there let continue by creating models in our **exam** app.

Note that, you can write your views or models like the way you usually write them when using flask without sakyum. It work great, nothing change.

1.4 schoolsite project

1.4.1 App models

Now we are going to create models for our **exam** app, the models are going to be two *ExamQuestionModel* and *ExamChoiceModel*

To create these two models we have to go into our **exam** app models.py **exam/models.py**. We will notice some default import:

```
from datetime import datetime
from schoolsite.config import db
```

Now below we are to start defining our model, let start with **ExamQuestionModel** model which will look like:

```
class ExamQuestionModel(db.Model):
  """ Exam default Question model """
  id = db.Column(db.Integer, primary_key=True)
  date_posted = db.Column(db.DateTime, nullable=False, default=datetime.utcnow)
  # the user field is the user who create the question and he is in the `User` models of␣
↪auth
  user = db.relationship('User', backref='user')
  user_id = db.Column(db.Integer, db.ForeignKey('user.id'))
  question_text = db.Column(db.Text, nullable=False)
  choices = db.relationship('ExamChoiceModel', backref='selector', lazy=True)

  def __str__(self):
    return f'{self.question_text}'

  def __repr__(self):
    return f'{self.question_text}'

  # the `ExamChoiceModel` is the choice model class below
  # the `selector` is the attribute that we can use to get selector who choose the choice
  # the `lazy` argument just define when sqlalchemy loads the data from the database
```

Now let define the **ExamChoiceModel** model which will look like:

```
class ExamChoiceModel(db.Model):
  """ Exam default Choice model """
  id = db.Column(db.Integer, primary_key=True)
  date_posted = db.Column(db.DateTime, nullable=False, default=datetime.utcnow)
  question_id = db.Column(db.Integer, db.ForeignKey('exam_question_model.id'),␣
↪nullable=False)
  # you can pass a keyword argument of `unique=True` in the below choice_text field
  # that will make it unique across the entire table of choice
  choice_text = db.Column(db.String(100), nullable=False)

  def __str__(self):
    return f'{self.choice_text}'

  def __repr__(self):
    return f'{self.choice_text}'
```

After pasting them, save the file. From here we can now create a migration for our *ExamQuestionModel and ExamChoiceModel* models using alembic, check how to create migration using alembic in sakyum, but we are going to skip this and just play with *api*.

Play with api

Before we move further let us play with the model api. This is the continuation from the last tutorial where we stop, when we make debug value to be *True* after registering the app (last tutorial)

From there shutdown the development server and go into the python **shell** (python interpreter), make sure you are within that directory you boot up the server by typing **python**, once you are in the interpreter, start by importing your **db** and **bcrypt** (for password hash) instance from project package (schoolsite), and also import the models you create for your app in *exam/models.py* and the default User model located in *auth.models.py*:

```
from sakyum.contrib import bcrypt
from sakyum.auth.models import User
from schoolsite.config import db
from exam.models import ExamQuestionModel, ExamChoiceModel
```

Next call the *create_all()* method of **db** that will create the tables of our models and database (if it doesn't create db file). Run the below command.:

```
db.create_all()
```

After that let us create three users instance, that will be able to create question and choice of the **ExamQuestionModel** and **ExamChoiceModel** model:

```
user1_hashed_pwd = bcrypt.generate_password_hash('123456').decode('utf-8')
user1 = User(username='backend-developer', email='developer@backend.com', password=user1_
↪hashed_pwd)

user2_hashed_pwd = bcrypt.generate_password_hash('123456').decode('utf-8')
user2 = User(username='front-developer', email='developer@front.com', password=user2_
↪hashed_pwd)

user3_hashed_pwd = bcrypt.generate_password_hash('123456').decode('utf-8')
user3 = User(username='quantum-developer', email='developer@quantum.com', password=user3_
↪hashed_pwd)
```

Now we are to add and commit those users in our database:

```
db.session.add(user1)
db.session.add(user2)
db.session.add(user3)
db.session.commit()
```

To make sure our users have been added in our database let query the entire User model of our project by:

```
User.query.all()
# [User('backend-developer', 'developer@backend.com', User('front-developer',
↪'developer@front.com', User('quantum-developer', 'developer@quantum.com']
```

Yes, our users are in the database, good jod. The next thing now is to start creating our Questions and commit them to our database:

```
q1 = ExamQuestionModel(question_text='At which year Neil Armstrong landed in the moon?',␣
↪user=user1)
q2 = ExamQuestionModel(question_text='What is odd in the choice?', user=user2)
q3 = ExamQuestionModel(question_text='What is not related to quantum?', user=user3)

db.session.add(q1)
db.session.add(q2)
db.session.add(q3)
db.session.commit()
```

To make sure our *questions* are in the database let query them to see by:

```
ExamQuestionModel.query.all()
# [At which year Neil Armstrong landed in the moon?, What is odd in the choice?, What is␣
↪not related to quantum?]
```

Yes, our questions are in the database, good jod. We are to capture our questions *id* (q1, q2 and q3) since they are the once we are going to link to each choice:

```
the_q1 = ExamQuestionModel.query.get_or_404(1)
the_q2 = ExamQuestionModel.query.get_or_404(2)
the_q3 = ExamQuestionModel.query.get_or_404(3)

# choices for our first question
c1_1 = ExamChoiceModel(choice_text='In 1969', question_id=the_q1.id)
c1_2 = ExamChoiceModel(choice_text='In 1996', question_id=the_q1.id)
c1_3 = ExamChoiceModel(choice_text='In 2023', question_id=the_q1.id)
c1_4 = ExamChoiceModel(choice_text='In 2007', question_id=the_q1.id)

# choices for our second question
c2_1 = ExamChoiceModel(choice_text='python', question_id=the_q2.id)
c2_2 = ExamChoiceModel(choice_text='java', question_id=the_q2.id)
c2_3 = ExamChoiceModel(choice_text='linux', question_id=the_q2.id)
c2_4 = ExamChoiceModel(choice_text='ruby', question_id=the_q2.id)

# choices for our third question
c3_1 = ExamChoiceModel(choice_text='qubit', question_id=the_q3.id)
c3_2 = ExamChoiceModel(choice_text='entanglement', question_id=the_q3.id)
c3_3 = ExamChoiceModel(choice_text='bit', question_id=the_q3.id)
c3_4 = ExamChoiceModel(choice_text='superposition', question_id=the_q3.id)

# Now let add and commit the choice into database::
db.session.add(c1_1)
db.session.add(c1_2)
db.session.add(c1_3)
db.session.add(c1_4)

db.session.add(c2_1)
db.session.add(c2_2)
db.session.add(c2_3)
db.session.add(c2_4)

db.session.add(c3_1)
```

(continues on next page)

(continued from previous page)

```
db.session.add(c3_2)
db.session.add(c3_3)
db.session.add(c3_4)

db.session.commit()
```

We can see choices related to our question number one (1) by:

```
ExamQuestionModel.query.get_or_404(1).choices
# [In 1969, In 1996, In 2023, In 2007]
```

To see many other method related to our *ExamQuestionModel.query* by passing it into *dir()* function:

```
dir(ExamQuestionModel.query)
```

To see all choices in our database:

```
ExamChoiceModel.query.all()
# [In 1969, In 1996, In 2023, In 2007, python, java, linux, ruby, qubit, entanglement,␣
↪bit, superposition]
```

Also like the *ExamQuestionModel.query* we see above, we can see many other method related to our *ExamChoiceModel.query* by passing it into *dir()* function:

```
dir(ExamChoiceModel.query)
```

Lastly let us make a loop over all question and print each question choices:

```
for question in ExamQuestionModel.query.all():
  question
  for choice in question.choices:
    print('\t', f'{choice.id}: ', choice)

# At which year Neil Armstrong landed in the moon?
#     1:  In 1969
#     2:  In 1996
#     3:  In 2023
#     4:  In 2007
# What is odd in the choice?
#     5:  python
#     6:  java
#     7:  linux
#     8:  ruby
# What is not related to quantum?
#     9:  qubit
#     10:  entanglement
#     11:  bit
#     12:  superposition
```

Since we insert something into the database, let move on, on how we can make those record to be display in the admin page (by registering the models), because if now we logout from the python interpreter and boot up the server **python thunder.py boot -d True** then navigate to admin page we won't be able to see those models. We can do so below:

Register our models to admin

In other to register our model, we are to open a sub project folder and open the **config.py** file we see there **(schoolsite/config.py)**, within create_app function in the file, we are to import our app models (**ExamQuestionModel**, **ExamChoiceModel**) that we want to register, above the method that will create the tables **db.create_all()** and we will see a commented prototype above it:

```
""" You will need to import models themselves before issuing `db.create_all` """
from sakyum.auth.models import User
from sakyum.auth.admin import UserAdminView
from exam.models import ExamQuestionModel, ExamChoiceModel
# from <app_name>.admin import <admin_model_view>
db.create_all() # method to create the tables and database
```

then we will append the models in the **reg_models = []** list within **admin_runner** function (inner function of the create_app function):

```
# rgister model to admin direct by passing every model that you
# want to manage in admin page in the below list (reg_models)
reg_models = [
  # User,
  ExamQuestionModel,
  ExamChoiceModel,
]
```

That will register our model in the admin page and we will be able to see it if we visit the admin page now! But this kind of registering admin model is not convenient, the convenient way is to use what is called admin model view.

Register model in the form of admin model view

We can register our model in the form of model view by grouping models that are related.

To create these model view we have to go into our app admin.py **exam/admin.py**. We will notice some default import:

```
from flask_login import current_user
from flask import redirect, request, url_for
from flask_admin.contrib.sqla import ModelView
```

Now below we are to start defining our model view, I will call the model view **QuestionChoiceAdminView** which will look like:

```
class QuestionChoiceAdminView(ModelView):
  can_delete = True  # enable model deletion
  can_create = True  # enable model deletion
  can_edit = True  # enable model deletion
  page_size = 50  # the number of entries to display on the list view

  def is_accessible(self):
    return current_user.is_authenticated

  def inaccessible_callback(self, name, **kwargs):
    # redirect to login page if user doesn't have access
    return redirect(url_for('auth.adminLogin', next=request.url))
```

The *is_accessible* method will check if a user is logged in, in other to show the *QuestionChoiceAdminView* model in the admin page, else it just show the plain admin page without the *QuestionChoiceAdminView*.

The *inaccessible_callback* method will redirect user (who is not logged in) to the login page of the admin.

In other to register our model view, open the *config.py* file (schoolsite/config.py) and import our admin model view (*QuestionChoiceAdminView*) below the import of our *ExamQuestionModel* and *ExamChoiceModel* which look like:

```
""" You will need to import models themselves before issuing `db.create_all` """
from sakyum.auth.models import User
from sakyum.auth.admin import UserAdminView
from exam.models import ExamQuestionModel, ExamChoiceModel
from exam.admin import QuestionChoiceAdminView
db.create_all() # method to create the tables and database
```

Now comment the **ExamQuestionModel** and **ExamChoiceModel** in the *reg_models* list, just like the way we comment the *User* in the list, because if we didn't comment it and we register our *QuestionChoiceAdminView* that mean we register *ExamQuestionModel and ExamChoiceModel* twice and that will trow an error:

```
# rgister model to admin direct by passing every model that you
# want to manage in admin page in the below list (reg_models)
reg_models = [
  # User,
  # ExamQuestionModel,
  # ExamChoiceModel,
]
```

go below the function we call **adminModelRegister** in (within admin_runner function) after registering our *UserAdminView* and call the admin method called **add_view** and then pass your model view class as an argument, also pass an arguments in the model view class, the first argument is the model class, the second is the **db.session**, and then last give it a category (key word argument) in our case we will call it **category='Question-Choice' like:

```
admin.add_view(QuestionChoiceAdminView(ExamQuestionModel, db.session, name='Questions',␣
↪category='Question-Choice'))
admin.add_view(QuestionChoiceAdminView(ExamChoiceModel, db.session, name='Choices',␣
↪category='Question-Choice'))
```

Save the file, that will register your related model in the admin page and you will see them if you vist the admin page *http://127.0.0.1:5000/admin*, only if you are logged in because of *is_accessible* method.

Now let navigate to *http://127.0.0.1:5000/login* and login using one of the user credential, we created when we were in the python interpreter (shell), the one (user credential) that we are going to use is for the *backend-developer* (username: **backend-developer**, password: **123456**).

After we logged in, now if we navigate to *http://127.0.0.1:5000/admin* we are able to see our *QuestionChoiceAdminView* view in the form of drop-down menu, if we click it, it will show list containing *Questions and Choices* only, since the are the only once associated with that mode admin view. Now click the *Questions* this will show list of questions we have inserted in the python shell.

Source code for the *app models* is available at official github repository of the project.

See more on how to write model view class at Flask-Admin documentation.

1.4.2 HTML forms

We can instead of using html file to write our forms, we can use this form feature that will represent our form template in the form of class and some methods.

To create forms we have to go into our app forms.py **exam/forms.py**. We will notice some default import:

```
from flask_wtf import FlaskForm
from wtforms import StringField, SubmitField, TextAreaField
from wtforms.validators import DataRequired, Length
```

Now below we are to start defining our forms, I will first start with **QuestionForm** form which will look like:

```
class QuestionForm(FlaskForm):
  """ Exam default Question form """
  question_text = TextAreaField('Question_Text', validators=[DataRequired()])
  submit = SubmitField('create')
```

Now I will define the **ChoiceForm** model which will look like:

```
class ChoiceForm(FlaskForm):
  """ Exam default Choice form """
  question_id = StringField('Question_Id', validators=[DataRequired()])
  choice_text = StringField('Choice_Text', validators=[DataRequired(), Length(min=2,␣
↪max=20)])
  submit = SubmitField('create')
```

Next is to go to our app views.py file **exam/views.py** and import the forms, make sure your follow the order of the import (you will see a prototype commented in your app views.py file above) which look like:

```
from flask import (render_template, Blueprint)
from sakyum.utils import footer_style, template_dir, static_dir
# from <project_name>.config import db
# from .models import <app_models>
# from .forms import <model_form>
```

Uncomment the fifth line, like:

```
from flask import (render_template, Blueprint)
from sakyum.utils import footer_style, template_dir, static_dir
# from <project_name>.config import db
# from .models import <app_models>
from .forms import QuestionForm, ChoiceForm
```

1.4.3 Advance tutorial

In this advance tutorial, we will expand our so long project (**schoolsite**), by creating another app called **result** in total making our apps to be two in number (**exam** and **result**). It is an add-on on top of the previous one.

Source code for the *quick start* is available at official github repository of the project.

1.5 Admin user

There are basically two ways in which you can create admin user. One is by using flags, second one is by prompt, and the other one is by using (combining both the two *prompt or flags*).

Let say you start a project, and an app inside the project by the following command:

```
python -c "from sakyum import project; project('schoolsite')" && cd schoolsite && python␣
↪thunder.py create_app -a exam
```

Admin user using flags:

This can be done by given the *create_user* position argument and flags together with their values e.g:

```
python thunder.py create_user -u network-engineer -e network-engineer@datacenter.com -p␣
↪my-secret-pass

# or

python thunder.py create_user --username network-engineer --email network-
↪engineer@datacenter.com --password my-secret-pass
```

`Warning:` don't use the *-p* flag to specify user password, do so only if you are testing (not in production) by just giving the user username, and email address, then enter, where as the password will be prompt to enter it, like:

```
python thunder.py create_user -u network-engineer -e network-engineer@datacenter.com
```

Admin user using prompt:

This can be done by only given the *create_user* position argument and then hit tab, e.g:

```
python thunder.py create_user
```

once you run it, a prompt will come up to input admin user information, these include *username*, *email*, and *password*

Admin user using both `flags` and `prompt` :

Use *-u* or *–username* and then the username beside it, if you do not specify it, you will see a prompt saying *Enter username:*:

```
python thunder.py create_user -u network-engineer

# or

python thunder.py create_user --username network-engineer
```

Use *-e* or *–email* and then the email beside it, if you do not specify it, you will see a prompt saying *Enter email:*:

```
python thunder.py create_user -e network-engineer@datacenter.com

# or

python thunder.py create_user --email network-engineer@datacenter.com
```

Use *-p* or *–password* and then the password beside it, if you do not specify it, you will see a prompt saying *Enter password:*:

```
python thunder.py create_user -p my-secret-pass

# or

python thunder.py create_user --password my-secret-pass
```

1.6 Custome auth

Custom authentication for users

In this chapter we are going to see how we can write a custom authentication for users which will replace the default route for our *auth* pages and instead of rendering the *wtforms* views, we are to use html form. To do so, sakyum already have html form for that available in the **[admin_register.html, admin_login.html, admin_change_password.html]**, now what remain for us is to create an app (*custom_auth*) just like the way we create the exam app:

```
python thunder.py create_app -a custom_auth
```

after that, paste the following in the *custom_auth* views.py file:

First we are to replace the top import with the following:

```
import re
import os
import secrets
from werkzeug.utils import secure_filename
from flask import Blueprint, render_template, request, redirect, url_for, flash, send_
↪from_directory, send_file
from sakyum.utils import footer_style, template_dir, static_dir
from flask_login import login_user, current_user, logout_user, fresh_login_required,␣
↪login_required
from sakyum.contrib import db, bcrypt
from sakyum.auth.models import User
# from .models import <app_models>
# from .forms import <model_form>


UPLOAD_FOLDER = os.environ.get('FLASK_UPLOAD_FOLDER')
ORIGIN_PATH = os.environ.get('FLASK_ORIGIN_PATH')
ALLOWED_EXTENSIONS = os.environ.get('FLASK_ALLOWED_EXTENSIONS')
```

Route for register: the default route of *adminRegister* can be replace with:

```
@custom_auth.route('/admin/register/', methods=['POST', 'GET'])
@login_required
def adminRegister():
  """
  The `admin_register.html` below is located in the sakyum package (templates/default_
↪page/admin_register.html)
  """
  if request.method == 'POST':
    username  = request.form['username']
    email  = request.form['email']
```

(continues on next page)

(continued from previous page)

```
    password1 = request.form['password1']
    password2 = request.form['password2']
    # username check
    check_username = User.query.filter_by(username=username).first()
    if check_username:
      flash(f'This username `{check_username}` has been taken!', 'error')
      return redirect(url_for('custom_auth.adminRegister'))
    # email check
    check_email = User.query.filter_by(email=email).first()
    if check_email:
      flash(f'This email `{check_email}` is taken, choose a different one.', 'error')
      return redirect(url_for('custom_auth.adminRegister'))
    # checking email pattern using regex
    pattern = re.compile(r'^[a-zA-Z0-9-_]+@[a-zA-Z0-9]+\.[a-zA-Z0-9]+')
    if not re.match(pattern, email):
      flash(f'Please use a valid email', 'error')
      return redirect(url_for('custom_auth.adminRegister'))
    # password check
    if len(password1) < 6 or len(password2) < 6:
      flash('Password must be not less than 6 character', 'error')
      return redirect(url_for('custom_auth.adminRegister'))
    if password1 == password2:
      hashed_password = bcrypt.generate_password_hash(password2).decode('utf-8')
      user_obj = User(username=username, email=email, password=hashed_password)
      db.session.add(user_obj)
      db.session.commit()
      flash(f'Account for {username} has been created!', 'info')
      return redirect(url_for('custom_auth.adminLogin'))
    else:
      flash(f'The two password fields didn\'t match', 'error')
  context = {
    'head_title': 'admin register',
    'footer_style': footer_style,
  }
  return render_template('admin_register.html', context=context)
```

Route for login the default route of *adminLogin* can be replace with:

```
@custom_auth.route('/admin/login/', methods=['POST', 'GET'])
def adminLogin():
  """
  The `admin_login.html` below is located in the sakyum package (templates/default_page/
↪admin_login.html)
  """
  if current_user.is_authenticated:
    return redirect(url_for('base.index'))
  if request.method == 'POST':
    username = request.form['username']
    password = request.form['password']
    user = User.query.filter_by(username=username).first()
    if user and bcrypt.check_password_hash(user.password, password):
      """
```

(continues on next page)

(continued from previous page)

```
      Parameters:
        user (object) - The user object to log in.

        remember (bool) - Whether to remember the user after their session expires.␣
↪Defaults to False.

        duration (datetime.timedelta) - The amount of time before the remember cookie␣
↪expires. If None the value set in the settings is used. Defaults to None.

        force (bool) - If the user is inactive, setting this to True will log them in␣
↪regardless. Defaults to False.

        fresh (bool) - setting this to False will log in the user with a session marked␣
↪as not "fresh". Defaults to True.
      """
      login_user(user, remember=True)
      flash('You are now logged in!', 'success')
      next_page = request.args.get('next')
      return redirect(next_page) if next_page else redirect(url_for('admin.index'))
    else:
      flash('Login Unsuccessful. Please check username and password', 'error')
  context = {
    'head_title': 'admin login',
    'footer_style': footer_style,
  }
  return render_template('admin_login.html', context=context)
```

Route for change password the default route of *adminChangePassword* can be replace with:

```
@custom_auth.route('/admin/change/password/', methods=['POST', 'GET'])
@fresh_login_required
def adminChangePassword():
  """
  The `admin_change_password.html` below is located in the sakyum package (templates/
↪default_page/admin_change_password.html)
  """
  if request.method == 'POST':
    old_password = request.form['old_password']
    password1 = request.form['password1']
    password2 = request.form['password2']
    # password check
    if len(password1) < 6 or len(password2) < 6:
      flash('Password must be not less than 6 character', 'error')
      return redirect(url_for('custom_auth.adminChangePassword'))
    user = User.query.filter_by(username=current_user.username).first()
    if user and bcrypt.check_password_hash(user.password, old_password):
      if password1 == password2:
        hashed_password = bcrypt.generate_password_hash(password2).decode('utf-8')
        user.password = hashed_password
        db.session.commit()
        flash('Your password has changed!', 'success')
        return redirect(url_for('custom_auth.adminLogin'))
```

(continues on next page)

(continued from previous page)

```
      else:
        flash('The two password fields didn\'t match', 'error')
    else:
      flash('Cross check your login credentials!', 'error')
  context = {
    'head_title': 'admin change password',
    'footer_style': footer_style,
  }
  return render_template('admin_change_password.html', context=context)
```

Route for logout the default route of *adminLogout* can be replace with:

```
@custom_auth.route('/custom_admin/logout/', methods=['POST', 'GET'])
@login_required
def adminLogout():
  logout_user()
  flash('You logged out!', 'info')
  return redirect(url_for('custom_auth.adminLogin'))
```

Route and functions for changing image and it route can be replace with:

```
def allowed_file(filename):
  return '.' in filename and filename.rsplit('.', 1)[1].lower() in ALLOWED_EXTENSIONS


@custom_auth.route('/profile_image/<path:filename>')
@login_required
def profile_image(filename):
  """
  This function help to show current user profile image, it won't download it
  like the `download_file` function below does
  """
  return send_file(UPLOAD_FOLDER + '/' + filename)


@custom_auth.route('/media/<path:filename>')
@login_required
def download_file(filename):
  """
  If we use this to show current user profile image, it won't show instead it will␣
↪download it,
  so it meant for downloading media file
  """
  return send_from_directory(UPLOAD_FOLDER, filename, as_attachment=True)


def picture_name(pic_name):
  random_hex = secrets.token_hex(8)
  _, f_ext = os.path.splitext(pic_name)
  picture_fn = random_hex + f_ext
  new_name = _ + '_' + picture_fn
  return new_name
```

(continues on next page)

(continued from previous page)

```
@custom_auth.route('/custom_admin/change_profile_image/', methods=['POST', 'GET'])
@login_required
def changeProfileImage():
  if request.method == 'POST':
    # check if the post request has the file part
    if 'file' not in request.files:
      flash('No file part')
      return redirect(request.url)
    file = request.files['file']
    # If the user does not select a file, the browser submits an
    # empty file without a filename.
    if file.filename == '':
      flash('No selected file')
      return redirect(request.url)
    if file and allowed_file(file.filename):
      filename = secure_filename(file.filename)
      file_name = picture_name(filename)
      file.save(os.path.join(UPLOAD_FOLDER, file_name))
      user = User.query.filter_by(username=current_user.username).first()
      if user:
        if user.user_img != 'default_img.png':
          r = str(ORIGIN_PATH) + '/media/' + user.user_img
          if os.path.exists(r):
            os.remove(r)
        user.user_img = file_name
        db.session.commit()
      flash('Your profile image has been changed!', 'success')
      return redirect(url_for('base.index')) # it will redirect to the home page
  context = {
    'head_title': 'admin change profile image',
    'footer_style': footer_style,
  }
  return render_template('admin_change_profile_image.html', context=context)
```

After all of the above, now open your project routes.py file (schoolsite/routes.py) and import your *custom_auth* blueprint:

```
from custom_auth.views import custom_auth
```

then pass it into the reg_blueprints list in other to register it by:

```
reg_blueprints = [
  blueprint.default,
  blueprint.errors,
  blueprint.auth,
  base,
  exam,
  custom_auth,
]
```

This will overwrite the default auth system for those routes. You can open the default admin page within your project

(templates/admin/index.html) and overite it with:

```
<!-- @sakyum, schoolsite (project) admin index.html page -->
{% extends 'admin/master.html' %}
{% block body %}
  <a href="/">Go to schoolsite home page</a>
  <br>
  {% if current_user.is_authenticated %}
    <a href="{{ url_for('custom_auth.adminLogout') }}">logout</a>
    <br>
    <a href="{{ url_for('custom_auth.adminChangePassword') }}">change password</a>
    <br>
    <a href="{{ url_for('custom_auth.adminRegister') }}">register</a>
    <br>
    <a href="{{ url_for('custom_auth.changeProfileImage') }}">change image</a>
  {% else %}
    <a href="{{ url_for('custom_auth.adminLogin') }}">login</a>
  {% endif %}
{% endblock body %}
```

Even the **User** model can be overwrite, but make sure to go all the files and import it from the custom_auth model instead of from sakyum. Note: `the creation of a user using the python thunder.py create_user command won't work` for the custom model.

Source code for the *custom auth* is available at official github repository of the project.

1.7 Templates / file system

Templates and static folder for each blueprint is located within (in side) the base templates/static directory, with thesame name of the bluprint, e.g let say we create an app *exam* in our project called *schoolsite* within the templates directory it will create (templates/exam) also for the static too (static/exam)

1.7.1 Customise admin page

A default directory in which you can customise admin page is created in the (templates/admin) directory with a default *index.html* file that contains some links. You can style it with different css and js file, but make sure it is extended from **{% extends 'admin/master.html' %}** and anything else wrap it within the body block **{% block body %} {% endblock body %}**

1.7.2 File system storage

The default directory where files will be saved is *media* which is in the project directory. In the media directory there is a default user profile image called *default_img.png*, and if user change profile image it will be available (saved the new image) in that directory.

1.8 Database migration

Welcome to the chapter that will talk about how to do database migration with *alembic*. By default the database that it (sakyum) come with is an sqlite database with naming convention of **default.db** located in the parent folder of your project. The main talk here is to show how we can make database migrations and stuffs like that.

Migrations are very powerful and let you change your models over time, as you develop your project, without the need to delete your database or tables and make new ones - it specializes in upgrading your database live, without losing data. More will be gist later.

Alembic is a very useful library we can use for our database migrations. when we are working with Flask Framework we need a tool which can handle the database migrations. Alembic is widely used for migrations. Alembic version *1.10.2* come with (Mako=1.2.4, MarkupSafe=2.1.2, SQLAlchemy=2.0.7, greenlet=2.0.2, typing-extensions=4.5.0) extensions, let us start how to use alembic.

First we need to initialize the alembic to our working project directory (parent) directory, by running the following command:

```
alembic init alembic
```

After running this command you will see some files and folders are created in your project directory they are *alembic* and *alembic.ini* the tree structure of the *alembic* directory is:

```
alembic
├── env.py
├── README
├── script.py.mako
└── versions

1 directory, 3 files
```

Notice there will be no version files in your versions directory *(alembic/versions)* because we haven't made any migrations yet. Now to use alembic we need to do certain changes in these files. First, change the *sqlalchemy.url* in your *alembic.ini* file, and give it your reletive *default.db* path, ours look like:

```
# for the default.db file
sqlalchemy.url = sqlite:////home/usman/Desktop/schoolsite/default.db

# for mysql (if you are using mysql database)
sqlalchemy.url = mysql+mysqldb://root:root@localhost:3306/database_name

# for postgresql (if youare using postgres database)
sqlalchemy.url = postgresql://user:user@localhost/test
```

After giving your database url, open a file that it generate in the alembic directory **alembic/env.py** find a variable called *target_metadata = None*, above it import your app models and the **db** instance of your application and replace the value of *None* with *db.Model.metadata* like in the below snippets:

```
from exam.models import ExamQuestionModel, ExamChoiceModel
from sakyum.contrib import db
target_metadata = db.Model.metadata
```

For Autogenerating Multiple MetaData collections, you can pass a list of models instead e.g:

```
from myapp.mymodel1 import Model1Base
from myapp.mymodel2 import Model2Base
target_metadata = [Model1Base.metadata, Model2Base.metadata]
```

Lastly make the migrations (Create a Migration Script) by runnig the following command:

```
alembic revision --autogenerate -m "Added tables"
```

Before, in the *alembic/versions* directory there is nothing inside, but now after running the above command, alembic generate our first migration commit file in versions folder *(alembic/versions)*, you can see the version file now in the versions folder, for simplicity the structure look like:

```
alembic
├── env.py
├── __pycache__
│   └── env.cpython-310.pyc
├── README
├── script.py.mako
└── versions
    ├── ac25f12f55b0_added_tables.py
    └── __pycache__
        └── ac25f12f55b0_added_tables.cpython-310.pyc

3 directories, 6 files
```

Every commit we did, it will generate the migration file in the *(alembic/versions)* directory.

Once this file generates we are ready for database migration. To migrate we are to run:

```
alembic upgrade head
```

Once you run the above command your tables will be generated in your database. This is how to use alembic for your database, there are many you can do so, hit to the alembic website for more clarification.

Each time the database models change, repeat the migrate and upgrade commands.

1.8.1 Hint

- To make migrations (Create a Migration Script):

  alembic revision –autogenerate -m “Added table”

- To migrate (Running our Migration):

  alembic upgrade head

- Getting Information:

  alembic current

  alembic history –verbose

- Downgrading, We can illustrate a downgrade back to nothing, by calling alembic downgrade back to the beginning, which in Alembic is called base:

  alembic downgrade base

Source code for the *database migration* is available at official github repository of the project.

1.9 Mod wsgi

Documentations of this page is under development (very soon) it will be available to public stay update, Thank you

1.10 Deployment

Documentations of this page is under development (very soon) it will be available to public stay update, Thank you

1.11 Sakyum on docker

This repo contains code to spin up a boilerplate sakyum project with Docker Compose. To run sakyum with docker compose, first pull it by:

```
docker pull usmanmusa/sakyum
```

Next you are to clone the github repo of the project in other to get the *docker-compose.yml* by:

```
git clone https://github.com/usmanmusa1920/sakyum.git
```

Now cd into the project folder you just clone to spin up the services using the command:

```
cd sakyum/example/sakyum_demo
```

To spinup the services, run the command:

```
docker-compose up
```

you can use the command below instead of the above, in other to see how it build the image:

```
docker-compose up --build
```

Once the services build up, you can visit it at *http://0.0.0.0:5000*, also you can login with these credentials, where username is: *backend-developer* and the password is: *123456*

Bonus usage

Inspect volume:

```
docker volume ls
```

and:

```
docker volume inspect <volume name>
```

Prune unused volumes:

```
docker volume prune
```

View networks:

```
docker network ls
```

Bring services down:

```
docker-compose down
```

Open a bash session in a running container:

```
docker exec -it <container ID> sh
```

Source code for the *database migration* is available at official github repository of the project.

CHAPTER

TWO

USEFUL LINKS:

- Repository
- PYPI Release