**Topics**

Routing

url\_for

Templates

Wtf forms

Database -sqlAlchemy

Migrations

Crud

Paginations

Blueprints

Restx api

Restful

Flash messages

Custom error page

Cookies

Sessions

Models

Admin page

-------------------------------------

Python -m venv env

pip install -r requirement.txt

--------------------------

pip install Flask

to run flask if u r using create function 🡪 **flask run –debug**

**python server.py**

from flask import Flask

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

@app.route("/")

def hello():

    return "hello world"

>export FLASK\_APP=flask\_blog.py

>flaskrun –debug

when we need to see html files / run server

#if we run with python eg: python flask\_blog.py

if \_\_name\_\_=="\_\_main\_\_":

    app.run(debug=True)

to write code inside html templates : **jinja2**

posts is created in file (flask\_revision)

 home .html page

 {% for post in posts %}

    <h1>{{ post.author}}</h1>

    <p>title {{ post.title}}</p>

 {% endfor %}

migrate using migrate package :

from flask\_migrate import Migrate

add this after models :

Migrate(app,db)

**---- Flask SQLAlchemy ----**

>>pip install Flask Flask-SQLAlchemy

**>> pip install psycopg2-binary**

Create a file ‘app.py’ 🡪 and import the following:

import os

from flask import Flask, render\_template, request, url\_for, redirect

from flask\_sqlalchemy import SQLAlchemy

add path for SQLite database:

basedir = os.path.abspath(os.path.dirname(\_\_file\_\_)) #current file dir

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

app.config['SECRET\_KEY']='1234'

app.config['SQLALCHEMY\_DATABASE\_URI'] ='sqlite:///' + os.path.join(basedir, 'database.db')

app.config['SQLALCHEMY\_TRACK\_MODIFICATIONS'] = False

db = SQLAlchemy(app)

creating a database :

>> export FLASK\_APP=app

>> flask shell

>>from app import db, <model name >

>>db.create\_all()

..

Add the values in table ..

..

db.session.add(variable)

To add multiple values :

>>> db.session.add\_all([owner1,owner2,owner3])

db.session.commit()

>> to display all records

User.query.all()

Display all records – using routes

Make sure u have created a template folder n html template in it

@app.route('/')

def index():

students = Student.query.all()

return render\_template('index.html', students=students)

display with id

@app.route('/<int:student\_id>/')

def student(student\_id):

student = Student.query.get\_or\_404(student\_id)

return render\_template('student.html', student=student)

for get , post , update, delete

route should be

@app.route('/create/', methods=('GET', 'POST'))

def create():

if request.method == 'POST':

firstname = request.form['firstname']

lastname = request.form['lastname']

email = request.form['email']

age = int(request.form['age'])

bio = request.form['bio']

student = Student(firstname=firstname,

lastname=lastname,

email=email,

age=age,

bio=bio)

db.session.add(student)

db.session.commit()

return redirect(url\_for('index'))

return render\_template('create.html')

update

@app.route('/<int:student\_id>/edit/', methods=('GET', 'POST'))

def edit(student\_id):

student = Student.query.get\_or\_404(student\_id)

if request.method == 'POST':

firstname = request.form['firstname']

lastname = request.form['lastname']

email = request.form['email']

age = int(request.form['age'])

bio = request.form['bio']

student.firstname = firstname

student.lastname = lastname

student.email = email

student.age = age

student.bio = bio

db.session.add(student)

db.session.commit()

return redirect(url\_for('index'))

return render\_template('edit.html', student=student)

delete :

@app.post('/<int:student\_id>/delete/')

def delete(student\_id):

student = Student.query.get\_or\_404(student\_id)

db.session.delete(student)

db.session.commit()

return redirect(url\_for('index'))

**------ Flask Restx ---**

* Pip install flask-restx
* In **\_\_init\_\_.py** file = create a function for app. -------- use **flask run** to run the app
* for debug mode on : make **.flaskenv** textfile n add FLASK\_DEBUG=True
* separate file **extension.py** to import sqlalchemy n db
* for swaggers :
* create new **resource.py** file , create namespace and register it in **\_\_init\_\_.py**

from flask\_restx import Resource, Namespace

eg : **ns = Namespace("api")**

@ns.route("/hello")

class Hello(Resource):

    def get(self):

        return("hello restx")

import the resouces file in init

create models for tables :-

class Course(db.Model):

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

    name=db.Column(db.String(50),unique=True)

    #sqlalchmey relationship

    students=db.relationship("Student", back\_populates="course")

(u need to create a separate models for api ) --- the fields that we want the server to accept / display

Eg:

from flask\_restx import fields

student\_model= api.model("Student",{

    "id":fields.Integer,

    "name":fields.String,

    "course":fields.Nested(course\_model)

})

to migrate :

>> flask shell

>>> from app.models import \*

>>> db.create\_all()

>>> exit()

To check sqlite tables

>>sqlite3 instance/db.sqlite3

>>.tables

>>.exit

For get :

    @ns.marshal\_list\_with(serializer\_name)

For post : use

@ns.expect(name\_of\_serializer\_model)

@ns.marshal\_with(serializer\_name) # to add single item

**--------------------------models---------------**

**-------------------- Relations / models ---------------------**

1)One to one relationship

class Parent(db.Model):

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

    name= db.Column(db.String(100))

    child=db.relationship('Child',backref='parent',uselist=False)  #uselist = list of student

class Child(db.Model):

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

    name= db.Column(db.String(100))

    parent\_id = db.Column(db.Integer, db.ForeignKey('parent.id'), unique=True)

here we get null value if 2-3 child r assign to same parent

**backref** : allows u to create relation on both the sides parent as well as child , adding new column ‘parent’ in child (in other table )

create parent and child In flask shell :

>>> parent=Parent(name='Parent3')

>>> child\_4 = Child(name='Dhan',parent=parent)

To add multiple values :

>>> db.session.add\_all([owner1,owner2,owner3])

2)One to many relationship

#one to many

class Owner(db.Model):

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

    name= db.Column(db.String(100))

    address= db.Column(db.String(100))

    pets = db.relationship('Pet', backref='owner') # by default the table name  is in lowercase in db

class Pet(db.Model):

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

    name= db.Column(db.String(100))

    age = db.Column(db.Integer)

    owner\_id= db.Column(db.Integer,db.ForeignKey('owner.id')) # by default the owner is lowercase in db

here we get the id value of the owner if 2-3 child r assign to same pet

If u restart the shell then u can get the value by :

owner1= Owner.query.filter\_by(name='candace').first()

**one to many does not contain uselist=False**

3) Many to many relationship

Create a new 3rd table , you use **secondary= user\_channel,** to refer to that table

With backref = ‘followers’ you can check the followers of a particular channel.

user\_channel = db.Table('user\_channel',

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

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

                        )

class User(db.Model):

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

    name= db.Column(db.String(100))

    following = db.relationship('Channel', secondary= user\_channel, backref='followers')

    def \_\_repr\_\_(self):

        return f'<User: {self.name}>'

class Channel(db.Model):

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

    name= db.Column(db.String(100))

    def \_\_repr\_\_(self):

        return f'<Channel: {self.name}>'

**-------flask wtf forms-------------------------**

Forms.py

from flask\_wtf import FlaskForm

from wtforms import StringField, PasswordField, SubmitField, BooleanField

from wtforms.validators import DataRequired, Length, Email, EqualTo

class RegistrationForm(FlaskForm):

    username = StringField('Username',validators=[DataRequired(), Length(min=2, max=20)])

    email = StringField('Email',validators=[DataRequired(), Email()])

    password = PasswordField('Password', validators=[DataRequired()])

    confirm\_password = PasswordField('Confirm Password',validators=[DataRequired(), EqualTo('password')])

    submit = SubmitField('Sign Up')

class LoginForm(FlaskForm):

    email = StringField('Email',validators=[DataRequired(), Email()])

    password = PasswordField('Password', validators=[DataRequired()])

    remember = BooleanField('Remember Me')

    submit = SubmitField('Login')

main.py

from forms import RegistrationForm, LoginForm

@app.route("/register",methods=['GET','POST'])

def register():

    form = RegistrationForm()

    if form.validate\_on\_submit():

        flash(f'Account created for {form.username.data}!','success')

        return redirect(url\_for('home')) #name of funct

    return render\_template('register.html', title='Register',form=form)

@app.route("/login",methods=['GET','POST'])

def login():

    form = LoginForm()

    if form.validate\_on\_submit():

        if form.email.data == 'dhan@gmail.com' and form.password.data == 'qwe123':

            flash('you have been logged in!','success') #success is bootstrap class

            return redirect(url\_for('home')) #name of funct

        else:

            flash('login unsuccessfull','danger')

    return render\_template('login.html', title='login',form=form)

in templates use

 {{ form.hidden\_tag() }}

**----------serialisers------------Rest api**

from flask import Flask, json,jsonify,request

from marshmallow import Schema,fields

#models

class Recipe(db.Model):

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

name=db.Column(db.String(255),nullable=False)

description=db.Column(db.Text(),nullable=False)

def \_\_repr\_\_(self):

return self.name

#deco to use later

@classmethod

def get\_all(cls):

return cls.query.all()

@classmethod

def get\_by\_id(cls,id):

return cls.query.get\_or\_404(id)

def save(self):

db.session.add(self)

db.session.commit()

def delete(self):

db.session.delete(self)

db.session.commit()

#shema – serialiser

class RecipeSchema(Schema):

id=fields.Integer(required=True)

name=fields.String()

description=fields.String()

#make a class eg . n write all functions under it

Class UserManager

def signup(self, data):

        if not data:

            return {"message": "No input data provided"}

        email = data.get("email")

        user = User.get\_user(email=email).first()

        try:

            if user == None:

                hashed\_password = bcrypt.generate\_password\_hash(

                    data.get("password")

                ).decode("utf-8")

                new\_acc = User(

                    username=data.get("username"),

                    email=data.get("email"),

                    password=hashed\_password,

                    first\_name=data.get("first\_name"),

                    last\_name=data.get("last\_name"),

                )

                db.session.add(new\_acc)

                db.session.commit()

                return {"message": "success"}

            else:

                return {"message": "enter different email"}

        except Exception as e:

            logger.error("some error occured", exc\_info=e)

            return {"message": "an error occured "}

#validations –api views

class SignUp(Resource, MethodView):

    @swagger\_api.doc()

    @swagger\_api.expect(signup\_model)

    def post(self):

        data = request.get\_json()

        # schema = SignSchema()

        # person = schema.dump(data)

        try:

            data2 = SignSchema().load(data)

        except ValidationError as err:

            # logger.error(f"{err.messages}")

            return err.messages

        print(data2, "----------------")

        new\_acc = UserManager().signup(data2)

        # result=schema.load(new\_acc)

        return new\_acc

**-------flask admin----**

pip install Flask Flask-SQLAlchemy Flask-Admin

from flask\_admin import Admin

admin = Admin()

inside create function

 admin.init\_app(app)

to use models import :

from flask\_admin.contrib.sqla import ModelView

models

class User(db.Model):

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

    name = db.Column(db.String(50))

    posts = db.relationship("Post", back\_populates="user")

    def \_\_str\_\_(self):

        return self.name

class Post(db.Model):

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

    title = db.Column(db.String(100))

    body = db.Column(db.Text)

    user\_id = db.Column(db.ForeignKey("user.id"), nullable=False)

    user = db.relationship("User", back\_populates="posts")

class PostView(ModelView):

    can\_delete = False

    form\_columns = ["title", "body", "user"] #fields to display -- create

    column\_list = ["title", "body", "user"]  # display list

admin.add\_view(ModelView(User, db.session))

admin.add\_view(PostView(Post, db.session))

**app function. Here**

**---------------choices---**

**Models:**

class PhoneNumber(BaseModel):

    NUMBER\_TYPE=[

    ("mobile", "Mobile Number"),

    ("mobile2", "Mobile2 Number"),

    ("home","Home Number"),

    ("office","Office Number")

    ]

    mobile = db.Column(db.String(20), unique=True, nullable=False)

    type = db.Column(ChoiceType (NUMBER\_TYPE))

    user\_id = db.Column(db.Integer, db.ForeignKey("user.id"), unique=True)

**migrations**

For choice remember to do changes in migration-version file , import sqlalchemy\_utils , add choice (number type)

Eg: migration file

import sqlalchemy\_utils

NUMBER\_TYPE=[

    ("mobile", "Mobile Number"),

    ("mobile2", "Mobile2 Number"),

    ("home","Home Number"),

    ("office","Office Number")

    ]

# revision identifiers, used by Alembic.

revision = '12aa25db0708'

down\_revision = '9940c9ec39f5'

branch\_labels = None

depends\_on = None

def upgrade():

    # ### commands auto generated by Alembic - please adjust! ###

    with op.batch\_alter\_table('phone\_number', schema=None) as batch\_op:

        batch\_op.add\_column(sa.Column('type', sqlalchemy\_utils.types.choice.ChoiceType(NUMBER\_TYPE), nullable=True))

        batch\_op.drop\_constraint('phone\_number\_home\_key', type\_='unique')

        batch\_op.drop\_constraint('phone\_number\_mobile2\_key', type\_='unique')

        batch\_op.drop\_constraint('phone\_number\_office\_key', type\_='unique')

        batch\_op.drop\_column('home')

        batch\_op.drop\_column('office')

        batch\_op.drop\_column('mobile2')

**----------------------------orm-------**

user= User.query.first() ------- first data

# Querying all users

users = User.query.all()

# Querying a specific user by username

user = User.query.filter\_by(username='john\_doe').first()

# Querying users with a specific email domain

users = User.query.filter(User.email.like('%@example.com%')).all()

# Updating a user's email

user = User.query.filter\_by(username='john\_doe').first()

user.email = 'new\_email@example.com'

db.session.commit()

# Updating a user's email

user = User.query.filter\_by(username='john\_doe').first()

user.email = 'new\_email@example.com'

db.session.commit()

(1)Write a query to get all the latest record by date.

latest\_messages = Message.query.order\_by(Message.created\_at.desc()).all()

(2)Write two queries.

(a)Retrieve first 5 records.

first\_five\_messages = Message.query.limit(5).all()

(b)Retrieve records from sixth to tenth.

next\_five\_messages = Message.query.offset(5).limit(5).all()

(3)Write queries to retrieve records with exact message like “Good morning”

whether it is case sensitive or case insensitive (write all possible queries)

# Case-sensitive

messages\_exact\_cs = Message.query.filter\_by(message\_text='Good morning').all()

# Case-insensitive

messages\_exact\_ci = Message.query.filter(Message.message\_text.ilike('Good morning')).all()

(4)Write a query to filter records in which messages are blank

messages\_blank = Message.query.filter(Message.message\_text == '').all()

(5)Write a query to retrieve records which are not contains messages like “Nice”

messages\_not\_nice=Message.query.filter(~Message.message\_text.ilike('%Nice%')).all()

(6)Write a query to filter records in which messages are starts with “Who” and created on ‘2010-05-02’ OR ‘2010-06-02’.

messages\_who\_and\_date = Message.query.filter(

(Message.message\_text.like('Who%')) &

((Message.created\_at >= '2010-05-02') & (Message.created\_at <= '2010-06-02'))

).all()

(7)Write a query to filter records with a messages\_from whose first name is “

messages\_from\_john = Message.query.join(User).filter(User.first\_name == 'John').all()

**---------GRPC----**

[Implementing gRPC In Python: A Step-by-step Guide (velotio.com)](https://www.velotio.com/engineering-blog/grpc-implementation-using-python)

[Python Microservices With gRPC – Real Python](https://realpython.com/python-microservices-grpc/)

**Command:**

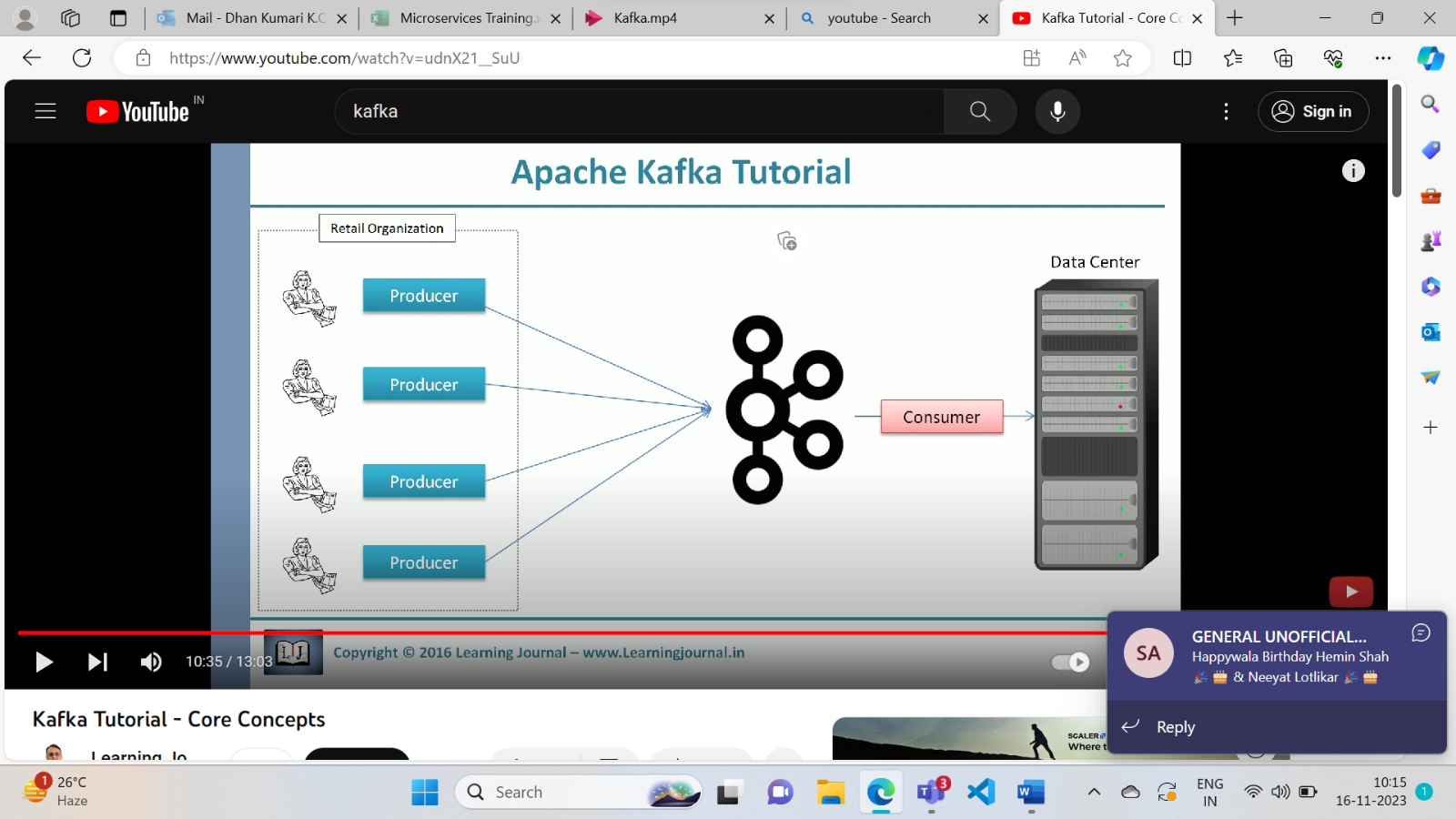
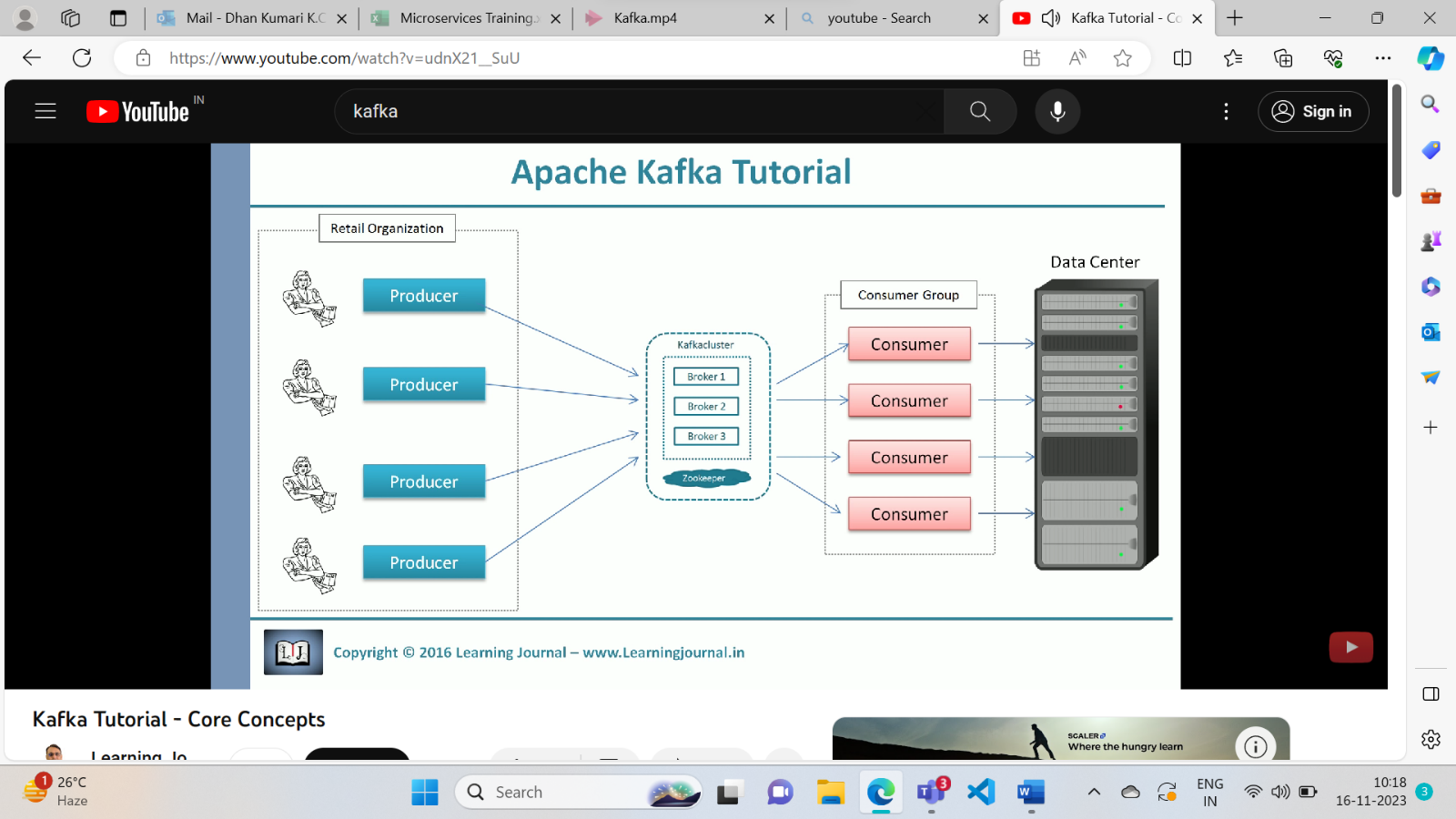
>>pip install grpcio-tools

>>python -m grpc\_tools.protoc -I protos --python\_out=. --grpc\_python\_out=. protos/greet.proto

(it will auto generate files )

Run server, run client

**-------Kafka -----**

* it acts like a broker(middle person )
* distributed streaming platform, works on cluster of computers .
* Stores data in partition – the partitioned data might be stored in diff. computer.
* To locate a message u should know – **topic name – partition number – offset num** ( sequence id of message)
* 
* 
* -- using Kafka method
* Create a topic . eg sales etc.
* Producers(responsible for publishing (writing) data to Kafka topics.)
* n consumers( They read and process messages from topics) r formed
* Kafka cluster takes place (diff. brokers r formed )-- the message is partitioned –-- consumer group is formed – each portioned topic is read by particular consumer
* One partitioned message is read by only 2 consumers to avoid overreading.

Code :

* Create a topic eg. Email\_notification, sales etc.
* Create docker container.
* Function to saves emails n all , there will be a kafka topic which was created in it . so we trigger it .
* (define the topic)
* call async. Function ----- call the function take the raw (json ) data n pass it to python
* write docker command( **faust -A <file\_name> woker -1 info**)
* (In whichever service u creating , u need to create faust file, **faust is like a messaging service** )
* (faust sends the messages from queue to kafka one by one and kafka executes it )
* **faust -A faust\_notification worker -l info**  # Running Faust Server in Notification service  
  $ chmod og+X /home /home/(username)

eg:

**Start Kafka: install**

**Create topic : (terminal)**

kafka-topics.sh --create --bootstrap-server localhost:9092 --replication-factor 1 --partitions 1 --topic

my\_topic

**Producer (Sender):**

Write program to produce n send message to the topic :

from kafka import KafkaProducer

producer = KafkaProducer(bootstrap\_servers='localhost:9092')

topic = ‘my\_topic’

for i in range(10):

message = f'Message {i}'

producer.send(topic, value=message.encode('utf-8'))

producer.close()

**Consumer (Receiver):**

Write a program or script to consume messages from the topic.

from kafka import KafkaConsumer

consumer = KafkaConsumer('my\_topic', bootstrap\_servers='localhost:9092', group\_id='my\_group')

for message in consumer:

print(f"Received message: {message.value.decode('utf-8')}")

consumer.close()

**Run Producer and Consumer:**

* Execute the producer script to send messages to the topic.
* Execute the consumer script to receive and process messages.

**Eg of faust and kafka :**

from faust import App, Topic

app = App('real-time-analytics', broker='kafka://localhost:9092')

# Define Kafka topics

page\_views\_topic = Topic('page\_views', value\_type=dict)

purchases\_topic = Topic('purchases', value\_type=dict)

registrations\_topic = Topic('user\_registrations', value\_type=dict)

# Faust agents for processing events

@app.agent(page\_views\_topic)

async def process\_page\_views(views):

async for view in views:

# Perform real-time analytics for page views

print(f"Analyzing page view: {view}")

@app.agent(purchases\_topic)

async def process\_purchases(purchases):

async for purchase in purchases:

# Perform real-time analytics for purchases

print(f"Analyzing purchase: {purchase}")

@app.agent(registrations\_topic)

async def process\_registrations(registrations):

async for registration in registrations:

# Perform real-time analytics for user registrations

print(f"Analyzing user registration: {registration}")

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

app.main()

----------------------------------------------------------------------------------------------------------------------------------------------------------

**docker – kafka**

install docker

>create n run the server :

docker-compose -f docker-compose.yml up -d

>to check if kafka is running or no :

docker run hello-world

>to open docker terminal:

docker exec -it kafka /bin/sh

**>**create topic :

/opt/kafka\_2.13-2.8.1/bin/kafka-topics.sh --create --topic hello\_world --bootstrap-server localhost:9092 --replication-factor 1 --partitions 1

/opt/kafka\_2.13-2.8.1/bin/kafka-topics.sh --create --topic send\_greetings --bootstrap-server localhost:9092 --replication-factor 1 --partitions 1

>to run **producer** :

/opt/kafka\_2.13-2.8.1/bin/kafka-console-producer.sh --topic hello\_world --bootstrap-server localhost:9092

>To run **consumer** (new docker terminal ):

/opt/kafka\_2.13-2.8.1/bin/kafka-console-consumer.sh --topic hello\_world --from-beginning --bootstrap-server localhost:9092

/opt/kafka\_2.13-2.8.1/bin/kafka-console-consumer.sh --topic send\_greetings --from-beginning --bootstrap-server localhost:9092

faust -A main worker -l info (main – file name)

**kafka-python :**

**(normal data)**

import faust

app = faust.App("demo-streaming", broker="localhost:9092")

topic = app.topic("hello\_world", value\_type=str, value\_serializer="raw")

@app.agent(topic)

async def processor(stream):

async for message in stream:

print(f"Received {message}")

>>to run :faust -A app worker -l info

**Complex data type:**

import faust

app = faust.App("demo-streaming", broker="localhost:9092")

class Greeting(faust.Record, serializer="json"):

from\_name: str

to\_name: str

input\_topic = app.topic("hello\_world", value\_type=Greeting)

output\_topic = app.topic("send\_greetings", value\_type=str, value\_serializer="raw")

@app.agent(input\_topic)

async def processor(stream):

async for message in stream:

output\_tf = f"Greeting from {message.from\_name} to {message.to\_name}"

print(output\_tf)

await output\_topic.send(value=output\_tf)

> json : {"from\_name":"Dhan","to\_name":"candace"}

import faust

#create app=app name, location of broker

app= faust.App('demo-streaming', broker='localhost:9092')

#for complex input data

class Greeting(faust.Record,serializer='json'):

    from\_name:str

    to\_name:str

#create topic, read data from 'hello\_world'

#topic = app.topic('hello\_world', value\_type=str, value\_serializer='raw')

input\_topic = app.topic('hello\_world', value\_type=Greeting)

output\_topic = app.topic('send\_greetings', value\_type=str, value\_serializer='raw')

#processing input

@app.agent(input\_topic) #deco to define async stream processor

async def processor(stream): #stream consumes data from topic

    async for message in stream: #consumer

        #print(f'Received {message}')

        output\_tf =f'Greeting from {message.from\_name} to {message.to\_name}'

        print(output\_tf)

        await output\_topic.send(value=output\_tf) #producing

to run : faust -A file\_name worker -l info

json : {"from\_name":"Dhan","to\_name":"candace"}

1. **async def processor(stream):**: This defines an asynchronous function named **processor**. The use of **async def** indicates that this function contains asynchronous operations. In this case, it is designed to process a Kafka stream asynchronously.
2. **async for message in stream:**: This is an asynchronous for loop. The **async for** syntax is used for iterating over an asynchronous iterator, in this case, the **stream**. The **stream** likely represents a Kafka topic, and the loop is used to asynchronously iterate over messages in that topic.

@agent.interval()

**FAUST**

<https://youtu.be/Nt96udaC5Zk?si=k5jCtv_GoPT1J0Yh>

eg: e – com website

* kafka topics : orders and order-processing
* flask app acts like a producer , by generating and sending order messages to the **orders** Kafka topic whenever a customer places an order.
* Another instance of the Flask application acts as a consumer by subscribing to the **orders** Kafka topic, processing incoming orders, and sending the processed order to the **order\_processing** topic.
* (When a customer places an order, the order details are sent as a message to the **orders** Kafka topic by the first Flask application.
* The second Flask application (consumer) subscribes to the **orders** topic, processes the incoming orders, and sends the processed orders to the **order\_processing** topic)