To Begin Use Flask, We Must Install It: **pip install flask**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

To Run The Flask Application:

* Create New File Called: **app.py**
* Create app-Variable Like:
  + Code Here: **app = Flask(\_\_name\_\_)**
* Run From Terminal: **flask run**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

The Flask Can Be Imported Like: from flask import Flask

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

To Get The Data From Request In Traditional Way:

from flask import request

Then We Get The Data AS Dict:

request.get\_json()

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

To Create Dynamic Routes That Accept Params:

@store\_blueprint.route('/store/<int:store\_id>')

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

To Add Flask-smorest To Flask App:

* First We Run: *pip install flask-smorest*
* Then We Import It For Each Resource-Python-File:
  + **Note**: Here We Create resources-Folder For Better Organization.
  + We Write Inside The Resource File:
    - from flask\_smorest import Blueprint
  + Then We Define Our Blueprint-Object That Define The Routes For Each Resource
    - item\_blueprint = Blueprint("Item", \_\_name\_\_, description= "The Item's Requests")
      * **Note**: The Description Is Important For Swagger Documentation
* Then For Creating Resource Class, We Need MethodView From Flask-Module:
  + from flask.views import MethodView
* Then We Define Our Class That Represent The Resource And Inherit From *MethodView-Class*
* **Note**: Foreach Resource We Have These Methods: **get**, **post**, **put**, **delete**, **patch**

@item\_blueprint.route('/item')

class ItemListResource(MethodView):

    def get(self):

        return ItemModel.get\_all\_items\_in\_db()

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

For Defining Our Swagger Configurations For Flask APP:

    app =Flask(\_\_name\_\_)

    app.config['PROPAGATE\_EXCEPTIONS'] = True # This is for debugging only

    app.config['API\_TITLE'] = "Jafar App"

    app.config['API\_VERSION'] = "v1"

    app.config['OPENAPI\_VERSION'] = '3.0.3'

    app.config['OPENAPI\_URL\_PREFIX'] = '/'

    app.config['OPENAPI\_REDOC\_URL'] = 'https://cdn.jsdelivr.net/npm/redoc@next/bundles/redoc.standalone.js'

    app.config['OPENAPI\_SWAGGER\_UI\_PATH'] = '/docs' # This is The Endpoint Where The Swagger UI Found

    app.config['OPENAPI\_SWAGGER\_UI\_URL'] = 'https://cdn.jsdelivr.net/npm/swagger-ui-dist/'

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

For Defining The Schema Of Our ***Request And Response***, When Using **flask-smorest**, We Can Use marshmallow-module:

**First**, We Import *Marshmallow-Module* With Required Data:

from marshmallow import Schema, fields

**Second**, We Define Our Classes That Represent The Shape Of Request And Response:

**Note**: Here We Use dump\_only That Means: This Field Will Populate Only In Response, And Not Required For Request.

**Note**: The required-attr Means The Request Must Contain This/These Field/Fields.

class PlainItemSchema(Schema):

    id = fields.Str(dump\_only=True)

    name = fields.Str(required=True)

    price = fields.Float(required=True)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

To Add The Shape Of Request To Our URL-Endpoint:

**First**, We Use Blueprints, And Method View To Define Our Resource.

**Second**, We Use Blueprint-object To Define The Argument Of Request Shape

**Note**: Here We Can Set Any Name For Request Shape in post-Method, Not Enough payload.

**Note**: Here If We Have Request Param, Then The Request Shape Will Be First In args, Then The Request-Param.

@item\_blueprint.route('/item')

class ItemListResource(MethodView):

    @item\_blueprint.arguments(schema=ItemSchema)

    def  post(self, payload):

        item = ItemModel(\*\*payload)

        # In Real scenario we must check the store id with

        # The Item Name, not just only the item name.

        if ItemModel.get\_item\_by\_name(item.name):

            abort(400, description="Duplicate Item Name")

        item.add\_item\_data\_to\_db()

        return item

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

For Adding The Shape Of Response With Marshmallow We Can Use: *@blueprint\_object.response(….)*

@item\_blueprint.route('/item/<int:item\_id>')

class ItemResource(MethodView):

    @item\_blueprint.response(status\_code=200, schema=ItemSchema)

    def get(self, item\_id):

        item\_data = ItemModel.get\_item\_by\_id(item\_id)

        if not item\_data:

            abort(404, {"message": f"No Item Found For ID {item\_id}"})

        return item\_data

    # The Order is Important for decorator.

    @item\_blueprint.arguments(schema=ItemUpdateSchema)

    @item\_blueprint.response(status\_code=200, schema=ItemSchema)

    def put(self, payload, item\_id):

        item = ItemModel.get\_item\_by\_id(item\_id=item\_id)

        item.update\_item\_in\_db(\*\*payload)

        return item

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Note**: If We Have List Of Objects Instead Of One Object, We Can Use Attribute: many=True

@item\_blueprint.route('/item')

class ItemListResource(MethodView):

    @item\_blueprint.response(status\_code=200,

                    schema=ItemSchema(many=True))

    def get(self):

        return ItemModel.get\_all\_items\_in\_db()

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

To Create Schema That Are Made Of Nested Of Specific Another Schema, We Can Use *fields.Nested*:

from marshmallow import Schema, fields

class PlainItemSchema(Schema):

    id = fields.Str(dump\_only=True)

    name = fields.Str(required=True)

    price = fields.Float(required=True)

class PlainStoreSchema(Schema):

    id = fields.Str(dump\_only=True)

    name = fields.Str(required=True)

class ItemSchema(PlainItemSchema):

    store\_id = fields.Int(required=True, load\_only=True)

    # if we want to use nested schema before defining them we

    # can use lambda: lambda: StoreSchema.

    # When we use nested schema may a recursive problem happened.

    # Then we must define a plain text schema and inherit from them.

    store = fields.Nested(PlainStoreSchema(), dump\_only=True)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

If We Want To Create Schema, That Contains List Of Nested Specific Object:

class StoreSchema(PlainStoreSchema):

    items = fields.List(fields.Nested(PlainItemSchema()), dump\_only=True)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

To Use SQLAlchemy With Flask, We Run: *pip install sqlalchemy flask-sqlalchemy*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

To Create SQLAlchemy Model:

Create Any db-python -file; Ex: **db.py**

Import The Required DB: from flask\_sqlalchemy import SQLAlchemy

Create The DB-Object: db = SQLAlchemy()

Then We Create The Model-Python-File; Ex: **items\_model.py**

Then Create The Class That Inherit From **db.Model**, And Create The Columns, With Required Relationships:

from db import db

class StoreModel(db.Model):

    \_\_tablename\_\_ = 'stores'

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

    name = db.Column(db.String(80), unique=True, nullable=False, index=True)

    items = db.relationship("ItemModel", back\_populates='store', lazy='dynamic')

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

from db import db

# The Column.default and Column.onupdate keyword arguments also accept Python

# functions. These functions are invoked at the time of insert or update if no

# other value for that column is supplied, and the value returned is used for # the column’s value.

class ItemModel(db.Model):

    \_\_tablename\_\_ = 'items'

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

    name = db.Column(db.String(80), index=True, nullable=False)

    price = db.Column(db.Float(precision=2), nullable=False)

    description = db.Column(db.String)

    store\_id = db.Column(db.Integer, db.ForeignKey('stores.id'),

    nullable=False)

    store= db.relationship("StoreModel", back\_populates='items')

    \_\_table\_args\_\_ = (

        db.UniqueConstraint('name', 'store\_id', name="item\_name\_store\_id\_uc"),

    )

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

To Use Flask-SQLAlchemy With Flask-APP, We Need Some Configuration For The Flask-APP:

app.config['SQLALCHEMY\_DATABASE\_URI']= db\_url or os.getenv('DATABASE\_URL', 'sqlite:///data.db')

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

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Then We Initialize The DB With The Flask-APP:

**Note**: We Must Use Factory Pattern To Create Flask-APP With Complete Configuration

from db import db

Then For Initialization:

db.init\_app(app=app)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Then To Create The Tables For OUR Database And APP:

**Note**: This Way Valid Only For Flask-SQLAlchemy Database Object.

# Here we use flask-migrate to generate the

# db, so we don't need this lines any more.

with app.app\_context():

db.create\_all()

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

To Add The Data To DB Using SQLAlchemy;

First, We Import The Models That We Need In Resource File (StoreResource): from models import StoreModel

Then We Define The Required Method For Saving Data Inside The **Model-Class**. (StoreModel)

def add\_store\_data\_to\_db(self):

        db.session.add(self)

        db.session.commit()

Then We Populate The Data And Check Constraint With Exceptions, Then Add Data, And Return Result:

@store\_blueprint.arguments(schema=StoreSchema)

@store\_blueprint.response(status\_code=201, schema=StoreSchema)

def post(self, payload):

        store\_data = StoreModel(\*\*payload)

        if StoreModel.get\_store\_by\_name(store\_data.name):

            abort(400, description="Duplicate Store Name")

        store\_data.add\_store\_data\_to\_db()

        return store\_data

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

For Getting The Data:

@classmethod

def get\_store\_by\_name(cls, name: str):

        return cls.query.filter(StoreModel.name == name).first()

@classmethod

def get\_store\_by\_id(cls, store\_id: int):

return cls.query.get\_or\_404(store\_id, description=f"No Data Found for store With store\_id: {store\_id}")

@classmethod

def get\_all\_stores\_from\_db(cls):

        return cls.query.all()

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

To Delete Items From DB:

    def delete\_store\_from\_db(self):

        db.session.delete(self)

        db.session.commit()

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

To Update The Data Of Specific Model, We Can Use *self.attr=value* Of Model Class, That Inherit From db.Model-SQLAlchemy Class:

# The Order is Important for decorator.

    @item\_blueprint.arguments(schema=ItemUpdateSchema)

    @item\_blueprint.response(status\_code=200, schema=ItemSchema)

    def put(self, payload, item\_id):

        item = ItemModel.get\_item\_by\_id(item\_id=item\_id)

        item.update\_item\_in\_db(\*\*payload)

        return item

def update\_item\_in\_db(self, name: str, price: float):

        self.name = name

        self.price = price

        db.session.commit()

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

To Retrieve All Items From DB, We Can Use all-Method:

@classmethod

    def get\_all\_items\_in\_db(cls):

      return cls.query.all()

And For Response We Set **many=True** For Blueprint-Response:

@item\_blueprint.route('/item')

class ItemListResource(MethodView):

    @item\_blueprint.response(status\_code=200,

                    schema=ItemSchema(many=True))

    def get(self):

        return ItemModel.get\_all\_items\_in\_db()

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*