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()
**********************************************************************
```