Project's creation from scratch;

Decorator Pattern in Python, GraphQL and Flask.

(the smart ifs)

Ines Ivanova

Table of content

Create venv

The creation is really easy assuming that you have already installed python (version > 3.4).
First, choose a location where you want to create it. Create a new folder with an appropriate name. Open your terminal there and write:

```
python -m venv {here is the absolute path to your folder}
```

NB! Please note that there should **NOT** be any spaces in your path.

To check if the creation is successful, you should be able to see these three folders in your main folder – Include, Lib, Scripts.

So, what is the point of creating a virtual environment, if we do not activate it, right?

With these three steps, you will be able to activate it and return to the root folder of our future project.

```
cd Scripts
activate
cd ..
```

Install packages

Now we will have to add some packages:

```
pip install Flask Flask-Migrate Flask-GraphQL Flask-SQLAlchemy graphene
```

Please observe if every package has been installed successfully. That's it! Let's start with the real fun!

You can output your requirements in a file now because we all know that we will forget it later.

```
pip freeze > requirements.txt
```

Structure the application

If you are using **PyCharm** open the folder that we have just created, and inside it, create a new python package called **blog** (Python package is a folder with *__init__.py* file in it).

Right next to it, create a python file called *blog_app.py*. In the blog package, we will have to create two more sub-packages called **models** and **utils**. In **models**, create three python files called **contact.py**, **user.py,** and **post.py.** In **utils**, create **enums.py.** Right next to **models** and **utils** packages create two python files **blog_post.py** and **config.py.**

If you execute the following command on the terminal, you will see the following tree:

```
tree /F
```

You may want to visualize it without the cache (use **PowerShell** for this).

```
tree /F | Where-Object {$_ -notlike "*.pyc"} |  Where-Object {$_ -notlike "*__pycache__"}
```

```
|   blog_post.py
|   config.py
|   __init__.py
|
├───models
|   |   contact.py
|   |   post.py
|   |   user.py
|   |
|
├───utils
|   |   enums.py
|   |   __init__.py
|   |
|
```

Configure the project

In **config.py**, we will need to define our **db** connection.

```
import os

basedir = os.path.abspath(os.path.dirname(__file__))


class Config(object):
    SQLALCHEMY_DATABASE_URI = 'sqlite:///' + os.path.join(basedir, 'blog.db')
    SQLALCHEMY_TRACK_MODIFICATIONS = False
```

In **blog_post.py**, we will create our application, migration, and **db** managers.

```
from flask import Flask
from blog.config import Config
from flask_sqlalchemy import SQLAlchemy
from flask_migrate import Migrate

app = Flask(__name__)
app.config.from_object(Config)
db = SQLAlchemy(app)
migrate = Migrate(app, db)
db.init_app(app)
```

After that, we will need to go to **blog_app.py**

```
from blog.blog_post import app
app.debug = True


if __name__ == '__main__':
    app.run()
```

Create models and relations

If you prefer to see the code directly, you can refer to this commit - here (do not worry – I removed the __pychache__ after that 😊)

First, let's define our user model in **models/user**

We do not observe anything interesting.

```
from blog.blog_post import db


class User(db.Model):
    __tablename__  = 'users'

    pk = db.Column(db.Integer, primary_key=True)
    title = db.Column(db.String(10))
    first_name = db.Column(db.String(20), nullable=False)
    last_name = db.Column(db.String(20), nullable=False)

    def __repr__(self):
        return f"<User {self.pk} {self.first_name} {self.last_name}>"
```

Let's define the models in **contact.py**

```python
from blog.blog_post import db
from blog.utils.enums import ContactType


class Email(db.Model):
    __tablename__  = "emails"

    pk = db.Column(db.Integer, primary_key=True)
    type = db.Column(
        db.Enum(ContactType),
        default=ContactType.personal,
        nullable=False
    )
    email = db.Column(db.String, nullable=False, unique=True)
    is_primary = db.Column(db.Boolean, nullable=False)
    user = db.Column(db.Integer, db.ForeignKey('users.pk'), nullable=False)

    def __repr__(self):
        return f'<Email {self.pk} {self.email}>'


class Phone(db.Model):
    __tablename__  = "phones"

    pk = db.Column(db.Integer, primary_key=True)
    type = db.Column(
        db.Enum(ContactType),
        default=ContactType.personal,
        nullable=False
    )
    country_code = db.Column(db.String(7), nullable=False)
    number = db.Column(db.String(20), nullable=False)
    extension = db.Column(db.String(7))
    is_primary = db.Column(db.Boolean, nullable=False)
    user = db.Column(db.Integer, db.ForeignKey('users.pk'), nullable=False)

    def __repr__(self):
        return f'<Phone {self.pk} {self.country_code} ' \
               f'{self.number} {self.extension}>'
```

Note that here we have imported **ContactType,** but we haven’t created yet. Go to **utils/enums** and create the following **enum**.

```python
import enum


class ContactType(enum.Enum):
    personal = "personal"
    work = "work"
```

This **enum** will help us define something as “choice field” for the contact type. Also, here we have defined one-to-many relationship user->phones, user->emails. I know that it seems unnecessary to have so many fields for email and phone but trust me, we are going to use them later for sure.

And the last one will be **models/post**.

```
from blog.blog_post import db


class Post(db.Model):
    __tablename__ = "posts"

    pk = db.Column(db.Integer, primary_key=True)
    title = db.Column(db.String(160), nullable=False)
    content = db.Column(db.Text, nullable=False)
    user = db.Column(db.Integer, db.ForeignKey('users.pk'), nullable=False)

    def __repr__(self):
        return f"<Post {self.pk} {self.Title}>"
```

Last but not least – after we have created our models, we should add imports at the end of **blog_post.app**.

```
from flask import Flask
from blog.config import Config
from flask_sqlalchemy import SQLAlchemy
from flask_migrate import Migrate

app = Flask(__name__)
app.config.from_object(Config)
db = SQLAlchemy(app)
migrate = Migrate(app, db)
db.init_app(app)

from blog.models.contact import Email, Phone
from blog.models.user import User
from blog.models.post import Post
```

Create db and migrations.

There are a couple of commands we need to run. First, we will need to set the flask app file. From the root of the project:

```
set FLASK_APP=blog_app.py
```

After that, we need to initialize the **db** and migrate.

```
flask db init
```

```
flask db migrate -m "Create base models for application."
```

```
db upgrade
```

Assure everything is set up correctly by running:

```
python blog_app.py
```

Setup GraphQl logic

Let’s start by creating a sub-package in a **blog** called **graph.** We will create 4 files **input.py, objects.py, schemas.py,** and **util.py**. If you want you can see the changes directly in **GitHub** from this commit**.**

```
cd blog/graph
```

From powershell:

```
tree /F | Where-Object {$_ -notlike "*.pyc"} |  Where-Object {$_ -notlike "*__pycache__"}
```

```
|   input.py

|   objects.py

|   schemas.py

|   util.py

|   __init__.py
```

We will start with a definition of out input/output objects: In input.py

```python
import graphene


class Iphone(graphene.InputObjectType):
    type = graphene.String()
    country_code = graphene.String(required=True)
    number = graphene.Int(required=True)
    extension = graphene.Int()
    is_primary = graphene.Boolean(default=True)


class IEmail(graphene.InputObjectType):
    type = graphene.String()
    email = graphene.String(required=True)
    is_primary = graphene.Boolean(default=True)


class IUser(graphene.InputObjectType):
    title = graphene.String()
    first_name = graphene.String(required=True)
    last_name = graphene.String(required=True)
    phone = Iphone()
    email = IEmail()
```

```python
import graphene


class Phone(graphene.ObjectType):
    type = graphene.String()
    country_code = graphene.String()
    number = graphene.Int()
    extension = graphene.Int()
    is_primary = graphene.Boolean()


class Email(graphene.ObjectType):
    type = graphene.String()
    email = graphene.String()
    is_primary = graphene.Boolean()


class User(graphene.ObjectType):
    pk = graphene.Int()
    title = graphene.String()
    first_name = graphene.String()
    last_name = graphene.String()
    phones = graphene.List(Phone)
    emails = graphene.List(Email)
```

And now, we will finally write our first mutation for the season:

```python
import graphene

from blog.graph.input import IUser
from blog.graph.objects import User


class CreateUser(graphene.Mutation):
    class Arguments:
        user_data = IUser(required=True)

    person = graphene.Field(User)

    def mutate(self, root, info, user_data=None):
        user = User(first_name=user_data.first_name,
                    last_name=user_data.last_name)
        return CreateUser(user)
```

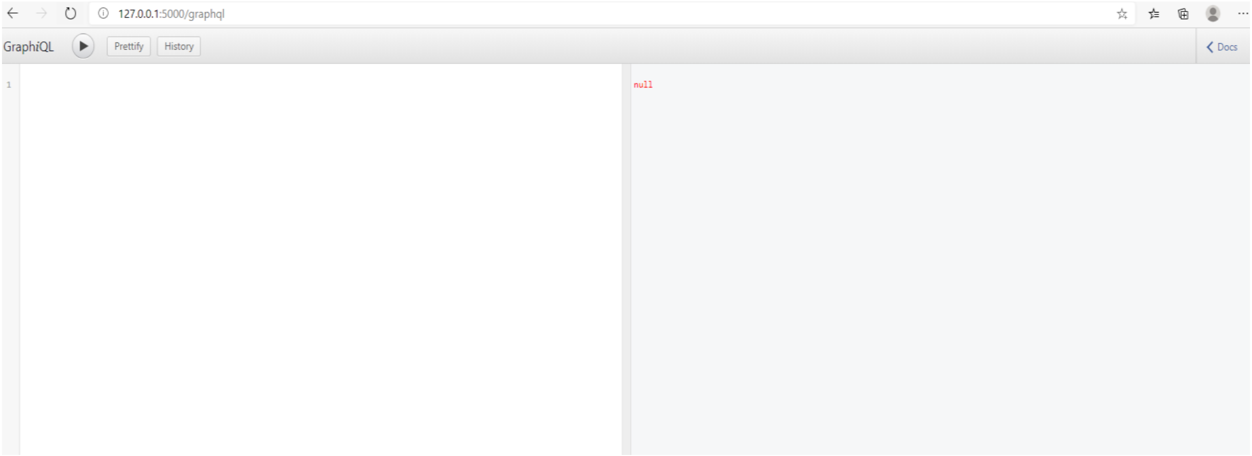
A Mutation is a special **ObjectType** that also defines an **Input**.

Arguments attributes are the arguments that the Mutation CreateUser needs for resolving: in this case, **IUser** will be the argument for the mutation.

Mutate is the function that will be applied once the mutation is being invoked. This method is just a special resolver that we can change data within. It takes the same arguments as the standard query Resolver Parameters.

Once you are ready with this part, you can try it. Start the application and go to: http://127.0.0.1:5000/graphql

You will see the **graphql** provided look:



```
null
```

You can run this:

```
mutation myFirstMutation {
  createUser(userData: {firstName:"Ines", lastName: "Ivanova"}) {
    person {
      firstName
    }
  }
}
```

And the result:

```
{
 "data": {
  "createUser": {
   "person": {
    "firstName": "Peter"
   }
  }
 }
```

Here you can feel free to add more mutations and queries. Just do not forget to add them in Mutations and Query classes in the **blog/graph/schemas.py** file.

Adding new requirements to BL

Now we will add some logic for saving the information to the **DB**. Let's create a file in **utils** called **db_helpers.py.**

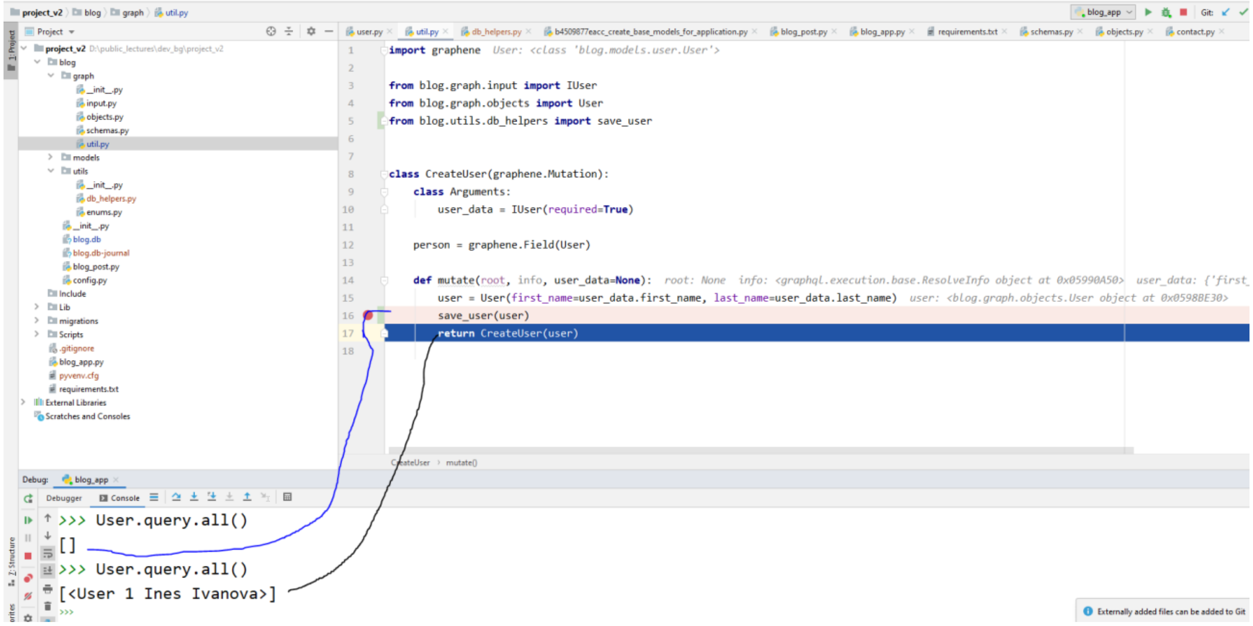
There we will create a function that will be responsible for saving the data to the **db**.

```python
from blog.blog_post import db
from blog.models.user import User


def save_user(user_data):
    user = User(first_name=user_data.first_name, last_name=user_data.last_name)
    db.session.add(user)
    db.session.commit()
```

And after that, we will invoke this function in our mutate method in **CreateUser** class:

```python
import graphene

from blog.graph.input import IUser
from blog.graph.objects import User
from blog.utils.db_helpers import save_user


class CreateUser(graphene.Mutation):
    class Arguments:
        user_data = IUser(required=True)

    person = graphene.Field(User)

    def mutate(root, info, user_data=None):
        user = User(first_name=user_data.first_name,
                    last_name=user_data.last_name)
        save_user(user)
        return CreateUser(user)
```

If we debug the application while using the python console in **PyCharm**, we can query the **User** and check if we had added a new record:



```python
import graphene  User: <class 'blog.models.user.User'>

from blog.graph.input import IUser
from blog.graph.objects import User
from blog.utils.db_helpers import save_user


class CreateUser(graphene.Mutation):
    class Arguments:
        user_data = IUser(required=True)

    person = graphene.Field(User)

    def mutate(root, info, user_data=None):  root: None  info: <graphql.execution.base.ResolveInfo object at 0x05990A50>  user_data: {'first_
        user = User(first_name=user_data.first_name, last_name=user_data.last_name)  user: <blog.graph.objects.User object at 0x0598BE30>
        save_user(user)
        return CreateUser(user)
```

```
>>> User.query.all()
[]
>>> User.query.all()
[<User 1 Ines Ivanova>]
>>>
```

So far so good. We know that if we try to send a query without **first_name** or **last_name**, the **graphql** will throw an error for us that there are required params. But what if we want to send information for the phone or email when we create the user? Well, we will for sure need to validate the data.

Let’s create the code for this scenario. Let’s send a query for a user with email data:

```
mutation myFirstMutation {
  createUser(userData: {
    firstName: "Ines",
    lastName: "Ivanova",
    email: {
      email: "example@email.com"
    }
  }) {
    person {
      firstName
    }
  }
}
```

Of course, adding this means that we have to change the mutate method. And because we are not sure if the **user** will be sent with an email or with an email and phone, the code for the mutate method should look like something like this:

```
import graphene

from blog.blog_post import db
from blog.graph.input import IUser
from blog.graph.objects import User
from blog.utils.validation_helpers import validate_email
from blog.utils.db_helpers import save_user, save_email


class CreateUser(graphene.Mutation):
    class Arguments:
        user_data = IUser(required=True)

    person = graphene.Field(User)

    def mutate(root, info, user_data=None):
        user = User(first_name=user_data.first_name,
                    last_name=user_data.last_name)
        user_pk = save_user(user)
        if user_data.email:
            if validate_email(user_data.email.email):
                save_email(user_data.email, user_pk)
        db.session.commit()
        return CreateUser(user)
```

Let's add the new files, and see how the code has changed in the existing. First create blog/constants.py, blog/utils/db_helpers.py and blog/utils/validation_helpers.py. Respectively the code in these files should be something like this:

```
EMAIL_REGEX = '^[a-z0-9]+[\._]?[a-z0-9]+[@]\w+[.]\w{2,3}$'
```

```
from blog.blog_post import db
from blog.models.user import User
from blog.models.contact import Email


def save_user(user_data):
    user = User(first_name=user_data.first_name, last_name=user_data.last_name)
    db.session.add(user)
    db.session.flush()
    return user.pk


def save_email(email_data, user_pk):
    email = Email(email=email_data.email, user=user_pk,
is_primary=email_data.is_primary)
    db.session.add(email)
    db.session.flush()
```

See how we replaced **.commit()** with **.flush()** – if we are going to modify more than one commit, we will change the thread and will throw an error. We put the **.commit()** in the mutate method.

```
import re

from blog.constants import EMAIL_REGEX


def validate_email(email):
    if re.search(EMAIL_REGEX, email):
        return True
    return False
```

In this file with helpers we can store our validation functions. In the input file we have also changed the named argument for email **class is_primary = graphene.Boolean(default=True)** to class is **_primary = graphene**. Boolean(**default_value**=True), because this is the actual param.

Now we can make the following query:

```
mutation myFirstMutation {
 createUser(userData: {
  firstName: "Ines",
  lastName: "Ivanova",
  email: {
   email: "example@email.com"
  }
 }) {
  person {
   firstName,
   emails {
    type
    email
    isPrimary
   }
  }
 }
}
```

But another question pops up – what if the email is not valid? Well, we have to change a little bit of the mutation **again.**

Create a file utils called exceptions and create a custom exception.

```
class InvalidInput(Exception):
    pass
```

Then we can import it and use it in mutation like this:

```
def mutate(root, info, user_data=None):
    user = User(first_name=user_data.first_name, last_name=user_data.last_name)
    user_pk = save_user(user)
    if user_data.email:
        if not validate_email(user_data.email.email):
            raise InvalidInput("Email not valid")
        save_email(user_data.email, user_pk)
    db.session.commit()
    return CreateUser(user)
```

The code for this part you can find in this commit.

Violation of SOLID

A new requirement is coming up – we should be able to receive phone as well and validate it. And where do the single responsibility and open/closed principle go when we do that in the mutate method? ☹ Of course, we can continue putting some code, but this will never end. There should be a better way? Well, there is.

Along with other approaches that we can apply, we should pay specific attention to the **decorator pattern** and adding some abstraction on top of it.

Time to refactor!

Solution with Decorator Pattern

We all know that we can put a decorator on top of the mutate and solve the problem for the email. Let's turn our **validate_email** to decorator. It happens like this:

```
import re

from blog.constants import EMAIL_REGEX
from blog.utils.exceptions import InvalidInput


def validate_email(function):
    def wrapper(*args, **kwargs):
        email_data = kwargs.get('user_data').get('email')
        email = email_data.get('email')
        if email and not re.search(EMAIL_REGEX, email):
            raise InvalidInput(f"Invalid email {email}")
        function(*args, **kwargs)
    return wrapper
```

And now our mutation function will look a bit cleaner:

```
@validate_email
def mutate(root, info, user_data=None):
    user = User(first_name=user_data.first_name, last_name=user_data.last_name)
    user_pk = save_user(user)
    save_email(user_data.email, user_pk)
    db.session.commit()
    return CreateUser(user)
```

Note that we are going to pass directly the user data, so we will need to change **save_email** so that it fits the new needs:

```
def save_email(user_data, user_pk):
    email_data = user_data.get("email")
    if email_data:
        email = Email(email=email_data.email, user=user_pk,
is_primary=email_data.is_primary)
        db.session.add(email)
        db.session.flush()
```

Adding an if, but much, much smarter than the previous we have had in the mutation.

The code for this changes is in this commit,

Take abstraction even further – the power of *args and **kwargs

But why stop here? And instead of adding decorators one above the other, the next time we need to add some validation (for example, the phone) we can make it even more beautiful – let's create a decorator that will take validation functions!

In **validation_helpers**, create a function called validate:

```
def validate(*validation_functions):
    def decorator(decorated_function):
        def wrapper(*args, **kwargs):
            for validation_func in validation_functions:
                if not callable(validation_func):
                    raise TypeError("The object in the decorator should be a
function reference")
                validation_func(**kwargs)
            decorated_function(*args, **kwargs)
        return wrapper
    return decorator
```

The purpose of this function – it has to be used as it is shown below in our mutate method (you have to import it first):

```
@validate(validate_email,)
def mutate(root, info, user_data=None):
    user = User(first_name=user_data.first_name, last_name=user_data.last_name)
    user_pk = save_user(user)
    save_email(user_data.email, user_pk)
    db.session.commit()
    return CreateUser(user)
```

As you can see, now we have a decorator which takes validation functions as arguments. Our **validate** function takes ***validation_functions**, which will be all the functions that you want to be included in the validation process. The **decorator** (nested function in **validate**) takes **decorated_function** which is basically our mutate method and the **wrapper** takes *__args__ and **__kwargs__, where **args** are the params (**root** and **info** from mutate method) and **__kwargs__ is actually our **user_data**.

It iterates through every function reference and checks if the object is actually a function. Then it invokes the validation function with **user_data**. And if everything is correct, it returns us to the mutate method at the end.

How awesome is that?

We have a little bit more refactoring to do in **validate_email**.
We will need to change it as follows:

```
def validate_email(**kwargs):
    email_data = kwargs.get('user_data').get('email')
    email = email_data.get('email')
    if email and not re.search(EMAIL_REGEX, email):
        raise InvalidInput(f"Invalid email {email}")
```

And now we are actually ready to just extend the functionality without breaking SOLID. You can create as many validation functions by **validate_email** example and add them to the validate decorator above the mutation.

The code for this you can find here.
Thank you for your attention! 😊