# 03.Schemas Validation and Password Hashing Lab

In this document, you will find helpful information on how you can integrate schemas so that you can serialize and deserialize objects easily, validate input data and shape response data.

## Setup project skeleton

Reusing our knowledge from the previous lecture, we will structure a base application. Again we will use **flask\_migrate**, **flask\_restful**, **flask\_sqlalchemy**, **psycopg2** (if needed), **psycopg2-binary** and of course **flask**. If you have created a new environment, you should install all of these packages. Also, we will implement a base clothes shop, so use your PyCharm (or pgAdmin) and create a new database called 'clothes'. The owner should be Postgres. Please, keep in mind, if you specify a different owner, you should change that in the database connection string.

On the next page you will find the application at its initial state. You can directly use this code as a skeleton so that you can start adding schemas, validations, and authentication.

Please do not forget to create .env file where you will store you DB\_USER and DB\_PASSWORD values.

Another interesting topic is the enum. The enum allows us to choose and store between predefined values. In our case, for the color and the size we want to pick between the values listed in the enums. The db column is also db.Enum and we can use the enum to place a default value.

The last thing that is a little bit different is that we added a column of type DateTime. Please note when we use **server\_default** and **onupdate** we use now from **sqlalchemy.func** not from **datetime.now**.

**import** enum  
  
**from** decouple **import** config  
**from** flask **import** Flask  
**from** flask\_migrate **import** Migrate  
**from** flask\_restful **import** Api  
**from** flask\_sqlalchemy **import** SQLAlchemy  
**from** sqlalchemy **import** func  
  
  
app = Flask(\_\_name\_\_)  
  
db\_user = config(**'DB\_USER'**)  
db\_password = config(**"DB\_PASSWORD"**)  
  
app.config[**'SQLALCHEMY\_DATABASE\_URI'**] = **f'postgresql://{**db\_user**}:{**db\_password**}@localhost:5433/clothes'**db = SQLAlchemy(app)  
api = Api(app)  
migrate = Migrate(app, db)  
  
  
**class** User(db.Model):  
 id = db.Column(db.Integer, primary\_key=**True**)  
 email = db.Column(db.String(120), nullable=**False**, unique=**True**)  
 password = db.Column(db.String(255), nullable=**False**)  
 full\_name = db.Column(db.String(255), nullable=**False**)  
 phone = db.Column(db.Text)  
 create\_on = db.Column(db.DateTime, server\_default=func.now())  
 updated\_on = db.Column(db.DateTime, onupdate=func.now())  
  
  
**class** ColorEnum(enum.Enum):  
 pink = **"pink"** black = **"black"** white = **"white"** yellow = **"yellow"  
  
  
class** SizeEnum(enum.Enum):  
 xs = **"xs"** s = **"s"** m = **"m"** l = **"l"** xl = **"xl"** xxl = **"xxl"  
  
  
class** Clothes(db.Model):  
 id = db.Column(db.Integer, primary\_key=**True**)  
 name = db.Column(db.String(255), nullable=**False**)  
 color = db.Column(  
 db.Enum(ColorEnum),  
 default=ColorEnum.white,  
 nullable=**False** )  
 size = db.Column(  
 db.Enum(SizeEnum),  
 default=SizeEnum.s,  
 nullable=**False** )  
 photo = db.Column(db.String(255), nullable=**False**)  
 create\_on = db.Column(db.DateTime, server\_default=func.now())  
 updated\_on = db.Column(db.DateTime, onupdate=func.now())  
  
  
**if** \_\_name\_\_ == **"\_\_main\_\_"**:  
 db.create\_all()  
 app.run(debug=**True**)

## Schemas

Marshmallow is an object-relational mapping library which is used to convert objects to and from Python data types. It is often used alongside SQLAlchemy, an ORM that maps database schemas to Python objects. Marshmallow is often used to deserialize Python dictionaries to SQLAlchemy models and vice versa. Let’s focus on how to use Marshmallow.  
pip install marshmallow

Schemas are nothing more than a simple class that should inherit Schema class from Marshmallow. As a regular python class they can be extended from other classes. When you write schemas, it is essential to keep in mind the base case, create a base schema for the model, and then extend it further for different purposes. For example, our User class has id, email, password, full\_name, phone, created\_at, updated\_on fields. But not all fields will be passed as an input (for example, when we create a user, we will pass only email and password) neither all fields will be passed as a response (the password field). In this case, we would like to make the as flexible as possible. We will end up with something like this:

**from** marshmallow **import** Schema, fields

**class** BaseUserSchema(Schema):  
 email = fields.Email()  
 full\_name = fields.String()  
  
  
**class** UserSignInSchema(BaseUserSchema):  
 password = fields.String())

A full documentation you can find [here](https://marshmallow.readthedocs.io/en/stable/index.html). You can always come back to it and check the verity of fields and options and apply the one that suits the best for your application case. Now we can do something like this:

**class** SignUp(Resource):  
 **def** post(self):  
 data = request.get\_json()  
 schema = UserSignInSchema()  
 errors = schema.validate(data)  
  
 **if not** errors:  
 db.session.add(User(\*\*data))  
 db.session.commit()  
 **return** 201, data

Do not forget to add the router to the api object if you want to try it in Postman

api.add\_resource(SignUp, "/register")

This looks as a right step, but the validation is missing, so now we will learn how to do it using schemas.

## Validation

Validating with Marshmallow schemas is an easy and flexible way of controlling the data that comes in with the request and the fields you want to show on the response.

The schemas come with the pre-defined types checking (if you define email as fields.Email), It will raise an error out of the box if you pass an integer, for example, and has pre-defined validation regex for a valid email which again is applied out of the box.

That is a great work, but most of the cases we do not have pre-build fields or validators and in real life we have far more complicated scenarios than that - field should or should not be required, can or can not be nullable, we are interested in min or max length of the strings, we may want to validate using regex and ect. Again for all of this marschmallaw suggest a solution. Consider these changes in the schemas:

**from** password\_strength **import** PasswordPolicy

**from** marshmallow **import** Schema, fields, validate, ValidationError

policy = PasswordPolicy.from\_names(uppercase=1, *# need min. 1 uppercase letters* numbers=1, *# need min. 1 digits* special=1, *# need min. 1 special characters* nonletters=1, *# need min. 1 non-letter characters (digits, specials, anything)*)

**def** validate\_password(value):  
 errors = policy.test(value)  
 **if** errors:  
 **raise** ValidationError(**f"Not a valid password"**)

**class** BaseUserSchema(Schema):  
 email = fields.Email(required=**True**)  
 full\_name = fields.String(required=**True**, validate=validate.Length(min=2))  
  
  
**class** UserSignInSchema(BaseUserSchema):  
 password = fields.String(required=**True**, validate=validate.And(validate.Length(min=8, max=20), validate\_password))

required=**True** means that when a data is validated against this schema it would expect to have a field with that name with actual value.

validate=validate.Length(min=2) using the validate we can pass a validator. Marshmallow schemas come with some common validators as Length, Range, Email and so on. You can find all of them [here](https://marshmallow.readthedocs.io/en/stable/marshmallow.validate.html).

validate=validate.And(validate.Length(min=8, max=20), validate\_password) using the .And we tell marshmallow that we want to apply more than on validator. In this case we validate against length and a custom function validate\_password. As you can see this function describe a param value (which will pass the password value dynamically when the validation against schema is triggered). It uses a library called **password\_strength** to validate the password. If you want to use it and do not feel like you want to write your custom function, **install it via pip**.

If you try it it will work correctly, but sadly it will store the password in plain text in the DB. We should never do that.

But you can do it even more elegant by importing **validates** from **marshmallow** and use it like this:

class BaseUserSchema(Schema):  
 email = fields.Email(required=True)  
 full\_name = fields.String(required=True)  
  
 @validates("full\_name")  
 def validate\_name(self, value):  
 try:  
 first\_name, last\_name = value.split()  
 except ValueError:  
 raise ValidationError(  
 "Full name should consists of first and last name at least"  
 )  
  
 if len(first\_name) < 3 or len(last\_name) < 3:  
 raise ValueError("Name should be at least 3 characters")

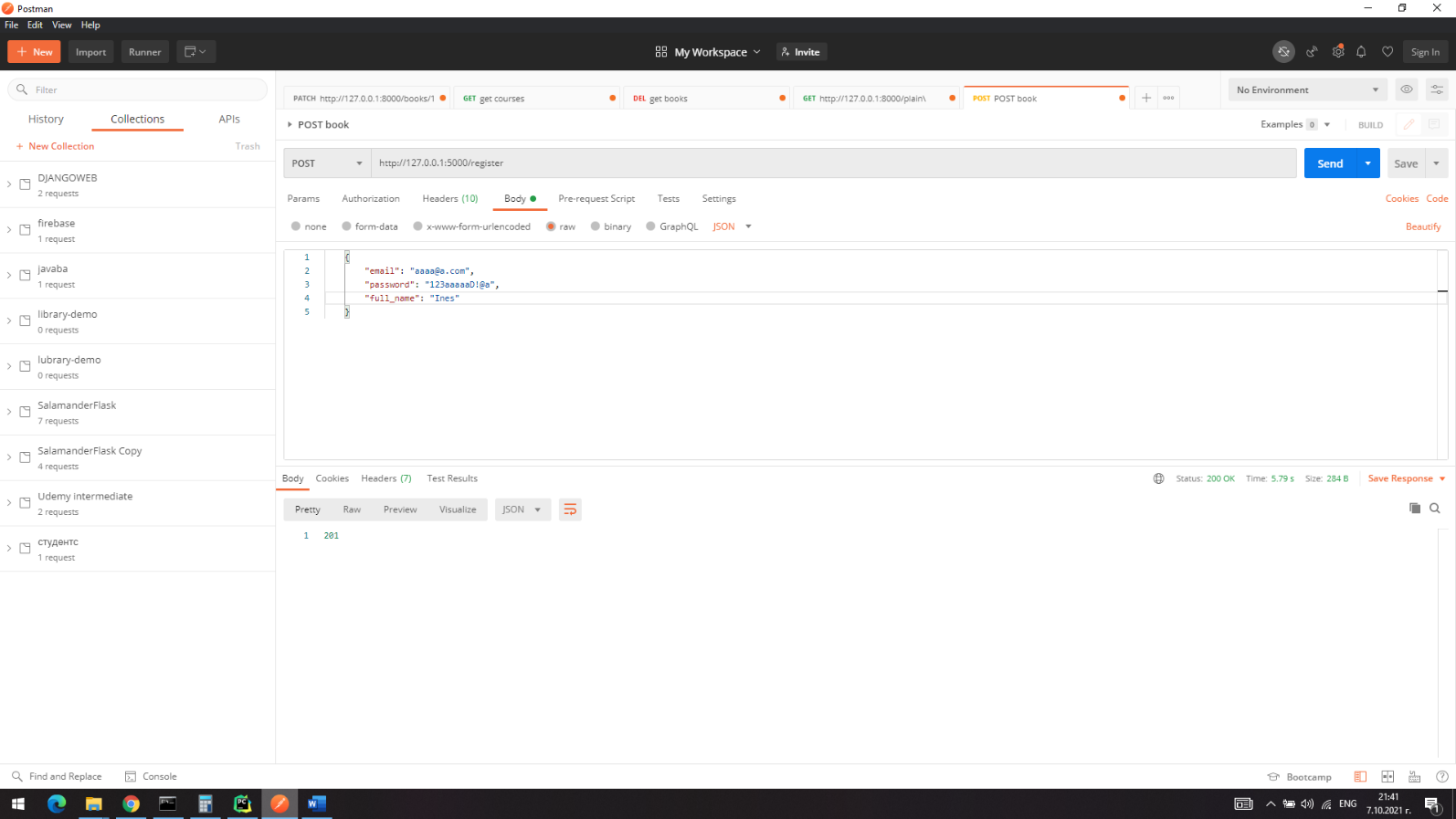
## Hashing the password

This is a short but crucial step that you do not have to neglect. You can use **generate\_password\_hash** from Werkzeug and the endpoint should be something like this:

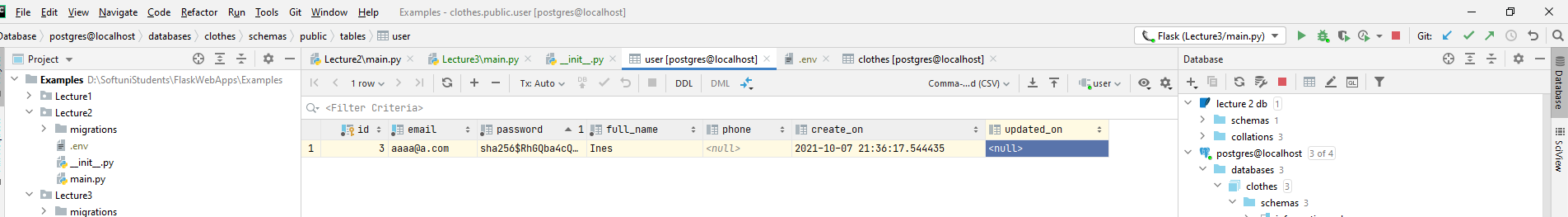
**from** werkzeug.security **import** generate\_password\_hash

**class** SignUp(Resource):  
 **def** post(self):  
 data = request.get\_json()  
 schema = UserSignInSchema()  
 errors = schema.validate(data)  
  
 **if not** errors:  
 data[**"password"**] = generate\_password\_hash(data[**'password'**], method=**'sha256'**)  
 db.session.add(User(\*\*data))  
 db.session.commit()  
 **return** 201, data  
 **return** 400

Now if you make a request with a valid data:



You will see the password is hashed:



## Bonus: Add many to many + nested response schemas

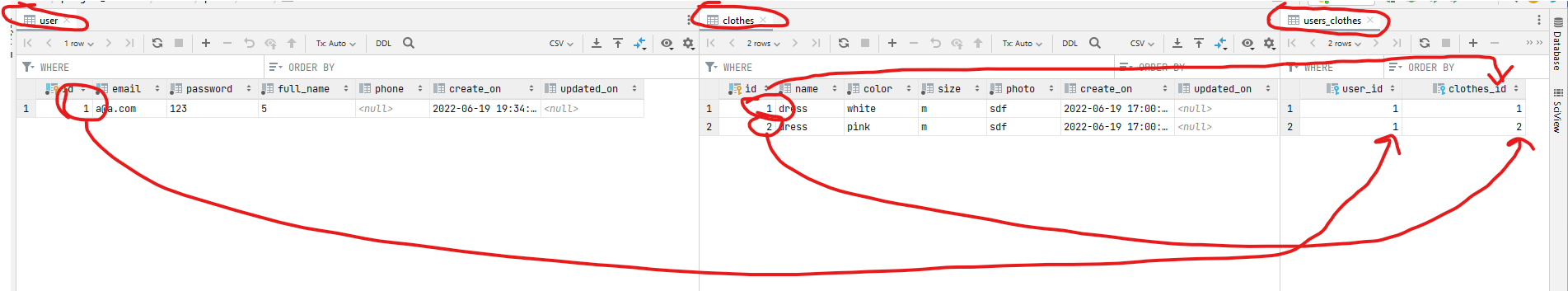
When we want to express that a user may have many clothes and clothes could be related to many users we need to define junction table which will be as simple as having two columns, both foreign keys to the user and clothes table:

users\_clothes = db.Table(  
 "users\_clothes",  
 db.Model.metadata,  
 db.Column("user\_id", db.Integer, db.ForeignKey("user.id")),  
 db.Column("clothes\_id", db.Integer, db.ForeignKey("clothes.id")),  
)

This will be a new table so we need to migrate the changes (flask db migrate -m “define any-to-many”) and then apply them by running flask db upgrade.

When you want to be able to see all clothes in the user object, you can define a db.relationship field (no changes to the database table are made here, so no migrations are needed)

class User(db.Model):  
 id = db.Column(db.Integer, primary\_key=True)  
 email = db.Column(db.String(120), nullable=False, unique=True)  
 password = db.Column(db.String(255), nullable=False)  
 full\_name = db.Column(db.String(255), nullable=False)  
 phone = db.Column(db.Text)  
 create\_on = db.Column(db.DateTime, server\_default=func.now())  
 updated\_on = db.Column(db.DateTime, onupdate=func.now())  
 **clothes = db.relationship("Clothes", secondary=users\_clothes)**

You can create two records in the clothes table by hand from the GUI interface you are using for database (PyCharm, pgAdmin and etc.), a user and make a connection between the user and the clothes in the junction table:  


Then we will define the endpoint to get user information (please note the next sentences will show and explain UserOutSchema)

class UserResource(Resource):  
 def get(self, pk):  
 user = User.query.filter\_by(id=pk).first()  
 return UserOutShema().dump(user)  
  
  
api.add\_resource(SignUp, "/register/")  
api.add\_resource(UserResource, "/users/<int:pk>/")

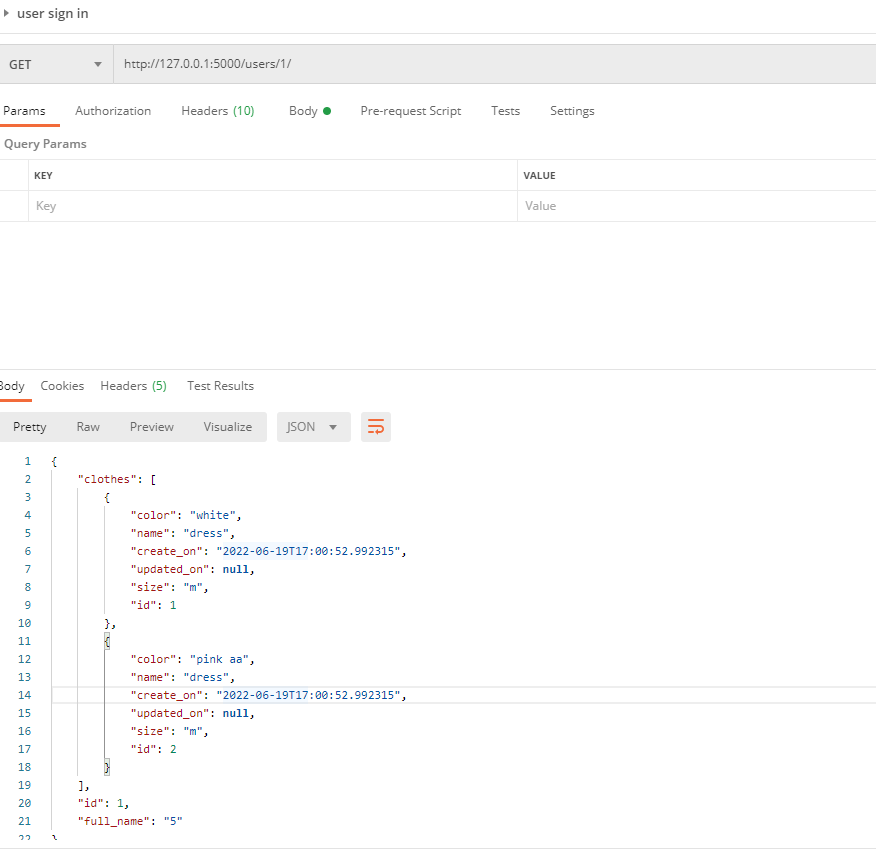
Schemas can be used to define response data as well. If you return directly the user, without a schema it will returh the object as it is, sometimes we might want to skip certain fields or add new fields, for this schemas can halp us as well. Here are the two schemas we can define:

Here we are defining the schema for the clothes (single row view), and then in the user schema we say that the clothes field will be a list of clothes, each row for clothes will be shaped according to SingleClothSchema, and also every row for the user will be shaped with UserOutSchema. For the Enum field we need to install:  
pip install marshmallow-enum

from marshmallow\_enum import EnumField  
from marshmallow import Schema, fields, ValidationError, validates

class SingleClothSchema(Schema):  
 id = fields.Integer()  
 name = fields.String()  
 color = EnumField(ColorEnum, by\_value=True)  
 size = EnumField(SizeEnum, by\_value=True)  
 create\_on = fields.DateTime()  
 updated\_on = fields.DateTime()  
  
  
class UserOutShema(Schema):  
 id = fields.Integer()  
 full\_name = fields.String()  
 clothes = fields.List(fields.Nested(SingleClothSchema), many=True)

If you test the endpoint with postman you should receive something like this:



Go and try it on your own: what will happen if you remove a field from the schema? What if you add a field in the schema which does not exist on the database model? What if you make that field required?