# Authentication, Schemas, Validation

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 implement authentication and authorization in Flask, using JWT.

## Authentication. JWT

JSON Web Tokens (or JWTs) provide a means of transmitting information from the client to the server in a stateless, secure way.

On the server, JWTs are generated by signing user information via a secret key, which are then securely stored on the client. This form of auth works well with modern, single page applications. For more on this, along with the pros and cons of using JWTs vs. session and cookie-based auth, please review [the following article](https://auth0.com/blog/cookies-vs-tokens-definitive-guide/).

The auth workflow works as follows:

* Client provides email and password, which is sent to the server
* Server then verifies that email and password are correct and responds with an auth token
* Client stores the token and sends it along with all subsequent requests to the API
* Server decodes the token and validates it
* This cycle repeats until the token expires or is revoked. In the latter case, the server issues a new token.

The tokens themselves are divided into three parts:

* Header
* Payload
* Signature

you can read more about each part from [here](https://jwt.io/introduction/).

To work with JSON Web Tokens in our app:

pip install pyjwt

The pyjwt documentation you can find [here](https://pyjwt.readthedocs.io/en/stable/)

We will need two functions – for encoding and for decoding the jwt.

The first one will help us to create a token. Here is the part where we have to set the token's expiration time and the so-called 'sub' (stands for subject). In most cases, the sub is the id of the user. Depending on the application security's requirements, the token can expire in 5 minutes or five days. Of course, you can set it never to expire, but this is not a good idea from a security perspective. You can decide where to define this function. If you are using class-based models (as we do now in our app), the User model would be a good place to define this function:

**def** encode\_token(self):  
 **try**:  
 payload = {  
 **'exp'**: datetime.utcnow() + timedelta(days=2),  
 **'sub'**: self.id  
 }  
 **return** jwt.encode(  
 payload,  
 key=config(**'SECRET\_KEY'**),  
 algorithm=**'HS256'** )  
 **except** Exception **as** e:  
 **raise** e

In the payload, we define the **exp** key and the **sub** key. We use the user’s id for the subject, and we set the exp date for two days. You can also define this in the .env file, so developers do not need to change it and commit it to production by mistake.

Now the signup resource should be changed a little:

**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'**)  
 user = User(\*\*data)  
 db.session.add(user)  
 db.session.commit()  
 token = user.encode\_token()  
 **return** {**"token"**: token}, **201**  
 **return** 400

Now if you make a request, the response in Postman should be:

{

    "token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJleHAiOjE2MzQwMzczMDcsInN1YiI6N30.z2hHbM6i2Xno7pft8A4pWAgkiDGN1Fbtk4t6qZfwW8E"

}

(The token value will be different for you)

If you face any difficulties, you can access the project repo and compare.

## Protected routes and Authorization

Let's assume we have a business logic that involves only authenticated users to access the clothes. Since we are doing REST API we could utilize a package called flask\_httpauth . It will help us define the restriction for our app's login and will allow us access the global user.

pip install flask\_httpauth

**from** flask\_httpauth **import** HTTPTokenAuth

auth = HTTPTokenAuth(scheme=**'Bearer'**)

@auth.verify\_token  
**def** verify\_token(token):  
 **try**:  
 user\_id = User.decode\_token(token)  
 **return** User.query.filter\_by(id=user\_id).first()  
 **except** Exception:  
 **return** 400

This @auth.verify\_token allows us to extend the token authentication against our custom implementation.

Now we can use the decorator in the newly created clothes resource. Do not forget to add the resource in the api object:

**class** ClothesRouter(Resource):  
 @auth.login\_required  
 **def** get(self):  
 current\_user = auth.current\_user()  
 clothes = Clothes.query.all()  
 **return** {**"data"**: clothes}, 200

Current user is not used in this case, but line just shows us that if we need the user object, we can access it from auth.current\_user(). That wouldn’t be possible if we haven’t implemented this logic into the verify\_token function to return the user object.

The final step would be the authorization. For this purpose, we need to add a role column in our User model:

**class** UserRolesEnum(enum.Enum):  
 super\_admin = **"super admin"** admin = **"admin"** user = **"user"  
  
  
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)  
 role = db.Column(  
 db.Enum(UserRolesEnum),  
 server\_default=UserRolesEnum.user.name,  
 nullable=**False** )

….

Do not forget to make the migrations and upgrade

flask db migrate

flask db upgrade

*\*if you see an error related to that the enum is not created you may need to do some adjustments manually in the the respective migration folder:*

def upgrade():  
 *# ### commands auto generated by Alembic - please adjust! ###* userrolesenum = postgresql.ENUM('super\_admin', 'admin', 'user', name='userrolesenum')  
 userrolesenum.create(op.get\_bind())  
 op.add\_column('user', sa.Column('role', sa.Enum('super\_admin', 'admin', 'user', name='userrolesenum'), nullable=False, server\_default='user'))  
 *# ### end Alembic commands ###*

We can now define a simple decorator to check if the user has the needed role to proceed with the request:

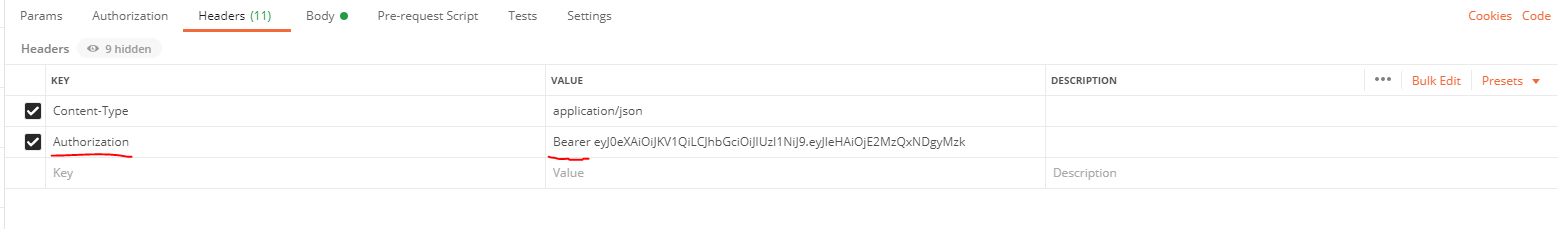
**def** permission\_required(permission):  
 **def** decorator(f):   
 **def** decorated\_function(\*args, \*\*kwargs):  
 user = auth.current\_user()  
 **if not** user.role == permission:  
 abort(403)  
 **return** f(\*args, \*\*kwargs)  
 **return** decorated\_function  
 **return** decorator

To take effect, this decorator should be placed on the function we want to. In our case, this will again be on the GET

**class** ClothesRouter(Resource):  
 @auth.login\_required  
 @permission\_required(UserRolesEnum.admin)  
 **def** get(self):  
 current\_user = auth.current\_user()  
 clothes = Clothes.query.all()  
 **return** {**"data"**: clothes}, 200

With this we conclude the topic for authentication and authorization. From now on you can extend these as much as you need to meet your application needs in future.

When you try the request in Postman, do not forget to include Authorization header in the headers tab:



## Small refactor

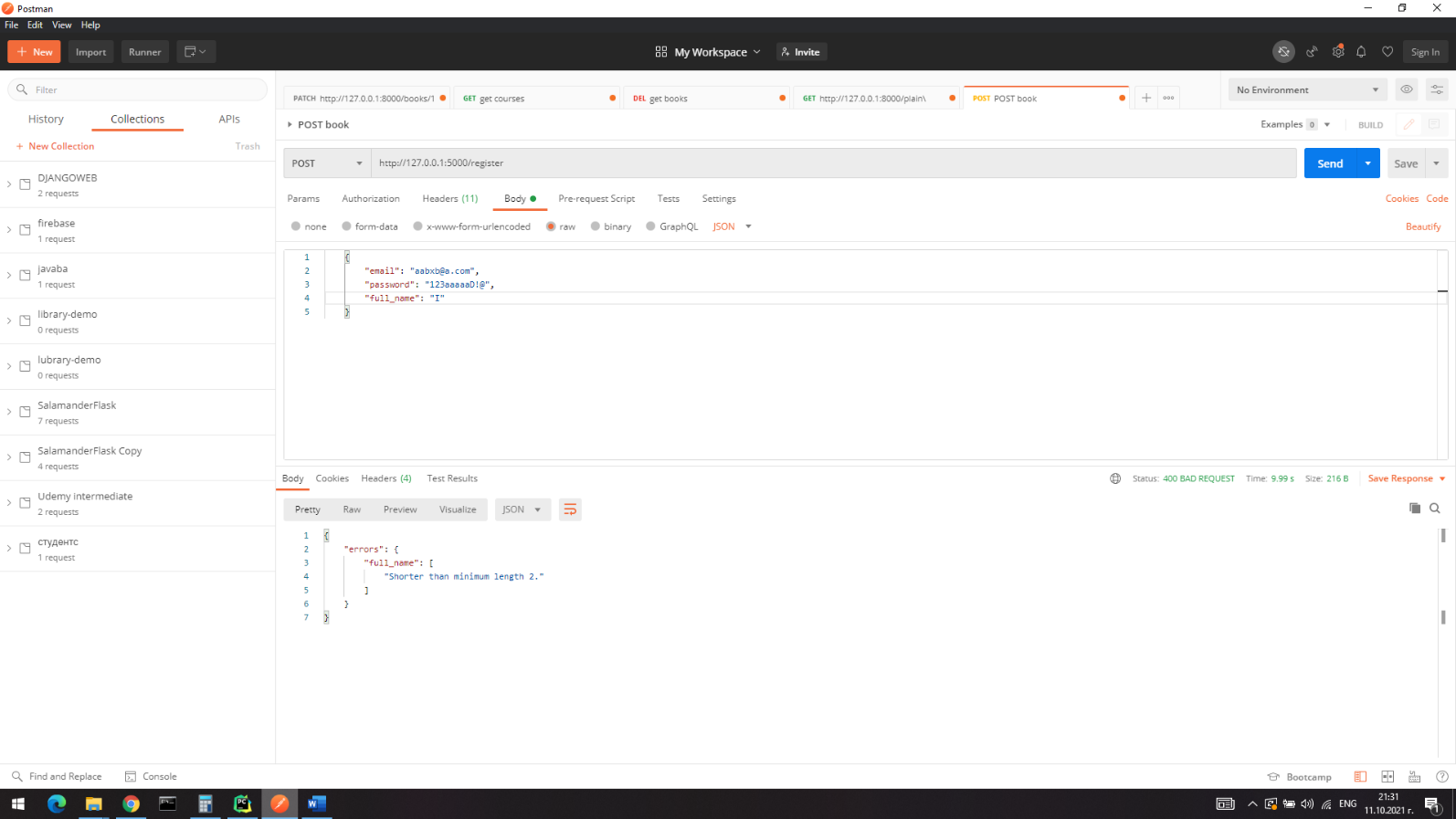
Remember the schemas we defined? We could now do it even better and remove the if from the function and create a decorator which will be reusable and could be extended more easily without breaking the SOLID principles:

**def** validate\_schema(schema\_name):  
 **def** decorator(f):  
 **def** decorated\_function(\*args, \*\*kwargs):  
 schema = schema\_name()  
 errors = schema.validate(request.get\_json())  
 **if** errors:  
 abort(400, errors=errors)  
 **return** f(\*args, \*\*kwargs)  
 **return** decorated\_function  
 **return** decorator

And the usage will be:

**class** SignUp(Resource):  
 @validate\_schema(UserSignInSchema)  
 **def** post(self):  
 data = request.get\_json()  
 data[**"password"**] = generate\_password\_hash(data[**'password'**], method=**'sha256'**)  
 user = User(\*\*data)  
 db.session.add(user)  
 db.session.commit()  
 token = user.encode\_token()  
 **return** {**"token"**: token}, 201

And now if you try to use it with incorrect data, and error will be thrown without even going to the post request:



In conclusion, we can say that we have done a pretty solid job, but our code is a mess. There are a lot of decorators, models, enumerators, and schemas in one place, and even though the app has only three routes, it is already hard to read.

Do not worry! In the following lecture, we will learn how to structure the Flask app, implement SOLID and MVC principles, and, most importantly, make the code more readable and reusable.