**Task Summary:**

"My STORE" is a B2B application, where it maintains an inventory for food items. All registered retailers/shop keepers, for placing order of required items via REST Webservices. As per the new requirement and more secured order transactions, here are the requirements:

Automatic process to generate TOKEN, every time user login credentials are verified.

With Order Details User has to pass the Automatic Generated Tokens as the part of Header.

**Tasks**

Required to have Approach Writeup

Information about the approach in the form of Flow Chart/Pseudo Code

Test Scenarios

Source Code in Python 3 / JAVA

**Solution:**

We are going to use flask library in Python3 for json web tokenisation (JWTs). On the server, JWTs are generated by signing user information via a secret key, which are then securely stored on the client.

Diagram

Description automatically generated

**Objectives:**

1. Discuss the benefits of using JWTs versus sessions
2. Implement user authentication with JWTs
3. Write tests to create and verify JWTs and user authentication

Tokens can expire like cookies, but we have more control. When we need to stream something, we use the token to get a signed request however the token gets sent on every request, hence we need to watch out for its size. If we store confidential info, we need to encrypt the token. JSON Web Tokens can be used in OAuth.

**Database Setup**

We are going to use Postgres database. Once the local Postgres server is running, we create two new databases from psql that share the same name as project name:

$ psql

# create database flask\_jwt\_auth;

# create database flask\_jwt\_auth\_test;

**User Model**

import datetime

class User(db.Model):

\_\_tablename\_\_ = "users"

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

email = db.Column(db.String(255), unique=True, nullable=False)

password = db.Column(db.String(255), nullable=False)

registered\_on = db.Column(db.DateTime, nullable=False)

admin = db.Column(db.Boolean, nullable=False, default=False)

def \_\_init\_\_(self, email, password, admin=False):

self.email = email

self.password = bcrypt.generate\_password\_hash(

password, app.config.get('BCRYPT\_LOG\_ROUNDS')

).decode()

self.registered\_on = datetime.datetime.now()

self.admin = admin

**Secret Key Generation:**

The secret key must be random and only accessible server-side. We use the Python interpreter to generate a key:

>>> import os

>>> os.urandom(24)

b"\xf9'\xe4p(\xa9\x12\x1a!\x94\x8d\x1c\x99l\xc7\xb7e\xc7c\x86\x02MJ\xa0"

Set the key as an environment variable:

$ export SECRET\_KEY="\xf9'\xe4p(\xa9\x12\x1a!\x94\x8d\x1c\x99l\xc7\xb7e\xc7c\x86\x02MJ\xa0"

Add this key to the SECRET\_KEY within config class

### Encode Token

Now we add the method to Generate Auth Token to the User() class:

import jwt

def encode\_auth\_token(self, user\_id):

try:

payload = {

'exp': datetime.datetime.utcnow() + datetime.timedelta(days=0, seconds=5),

'iat': datetime.datetime.utcnow(),

'sub': user\_id

}

return jwt.encode(

payload,

app.config.get('SECRET\_KEY'),

algorithm='HS256'

)

except Exception as e:

return e

### Now we test our token generator script:

import unittest

import db

import User

class TestUserModel():

def test\_encode\_auth\_token(self):

user = User(

email='test@test.com',

password='test'

)

db.session.add(user)

db.session.commit()

auth\_token = user.encode\_auth\_token(user.id)

self.assertTrue(isinstance(auth\_token, bytes))

if \_\_name\_\_ == '\_\_main\_\_':

unittest.main()

### Decode Token

We add the following method to decode a token:

def decode\_auth\_token(auth\_token):

try:

payload = jwt.decode(auth\_token, app.config.get('SECRET\_KEY'))

return payload['sub']

except jwt.ExpiredSignatureError:

return 'Signature expired. Please log in again.'

except jwt.InvalidTokenError:

return 'Invalid token. Please log in again.'

### Now we test our decoder script:

def test\_decode\_auth\_token(self):

user = User(

email='test@test.com',

password='test'

)

db.session.add(user)

db.session.commit()

auth\_token = user.encode\_auth\_token(user.id)

self.assertTrue(isinstance(auth\_token, bytes))

self.assertTrue(User.decode\_auth\_token(auth\_token) == 1)

We need to decode the auth token with every API request and verify its signature to be sure of the user’s authenticity. To verify the auth\_token, we used the same SECRET\_KEY used to encode a token.